Further Programming

COSC2288

Week 5 – Exception Handling and Unit Test

Dipto Pratyaksa

dipto.pratyaksa@rmit.edu.au

Basd on materials by:

A/Prof Andy Song

Dr. Mengmeng Ge

RMIT UNIVERSITY

Week 5

- Exceptions
- Handling Exceptions
- Recovering from Exceptions
- Exception Hierarchy
- Propagation of Exceptions
- The clause finally
- Writing our own exceptions
- JUnit

What is an exception?

There are three kinds of errors (in all programming languages):

- **Syntax/Compilation Errors**: detected at compile time

```
double pay = 0.0          // missing semicolon
```

- **Execution/Runtime Errors**: Appear when the program runs

- **Logic Errors:** program compiles and runs, but the results are not what they should be.

```
double avg = n1+n2+n3 / 3.0;      //should have a bracket
```

- **Exception is the mechanism handling Runtime Errors**
- If exception is not handled, program terminates abnormally

What if there is an error during the run?

```
import java.util.*;
public class Exception1{
    public static void main(String[] args){
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter your height (m) : ");
        double ht = sc.nextDouble();
        System.out.print("Your ht in (cm) = " + ht*100);
    }
}
```

Console

```
<terminated> Exception1 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_121.jdk/Contents/Home/bin/java (7 May 2018 11:19:59 am)
Enter your height (m) : abc
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextDouble(Scanner.java:2413)
        at week9.Exception1.main(Exception1.java:7)
```

Exception type

Exception trace

Exception

- Exception occurs for a number of reasons:
  - `ArrayIndexOutOfBoundsException`

```
int[] arr  = new int[5];
arr[5] = 1; // index should be between 0-4
```

  - Wrong type of input : `InputMismatchException` (Example in the previous slide)
  - Attempting to open a non-existent file: `FileNotFoundException` (we will see it today!)
  - Dividing by zero: `ArithmeticException`

… and many more exceptions …

Handling Exceptions

Some exceptions must be handled, otherwise we cannot compile.

This type of exception is called *checked* Exception.

For example, all exceptions related to file open/read/write – `IOException`

```
import java.io.*;
public class Exception1{
    public static void main(String[] args) {
        FileReader fw = new FileReader ("dest.txt");
    }
}
```

Cannot compile: Unhandled Exception type IOException

Handling Exceptions

Which type of exception have we seen so far?

Other exceptions are called *unchecked* exceptions –

we can compile, but if the exception occurs the program stops

So it is best to handle the possible exceptions

For example, `InputMismatchException` while using Scanner

Ways to handle exception

Declare that the program is not prepared to handle the exception

Handle by the program

– **catch exception**

Not prepared to handle an Exception

```
import java.io.*;

public class Week11_2 {
    public static void main(String[] args)
                                    throws IOException{
        FileReader input = new FileReader("NoSuchFile");
        int x = Integer.parseInt("a");
        System.out.println( 5/0 );

        int[] arr  = new int[5];
        arr[4] = 1; // index should be between 0-4
    }
}
```

This method is not prepared to handle it!

throws an **IOException**

FileNotFoundException

NumberFormatException

ArithmeticException

ArrayIndexOutOfBoundsException

throws

The **`throws`** clause specifies that

(1) Exceptions are not going to be handled in the method where they are generated, and

(2) That they should be thrown to the calling method.

```
public static void main(String[] args) {
    FileInputStream input = new FileInputStream("file.txt");
}
```

The above code cannot compile because there is a chance that **`FileInputStream()`** throws a **`FileNotFoundException`**, while its caller '**`main()`**' does not have "throws".

Another useful example

An alternative of Scanner is `BufferedReader`.

We didn't use BufferedReader before as it throws a checked exception and we must handle checked exceptions.

```
import java.io.*;
public class BufReadExample {
   public static void main (String[] args)
                                    throws IOException {
      BufferedReader stdin = new BufferedReader
         (new InputStreamReader (System.in));
      String string1; int num1;
      System.out.println ("Input an integer");

      string1 = stdin.readLine();
      num1 = Integer.parseInt (string1);

      System.out.print("The num is: " + num1);
}}
```

This method is not prepared to handle it!

may throw an IOException

may throw a NumberFormatException

Why is there no throws clause for NumberFormatException?

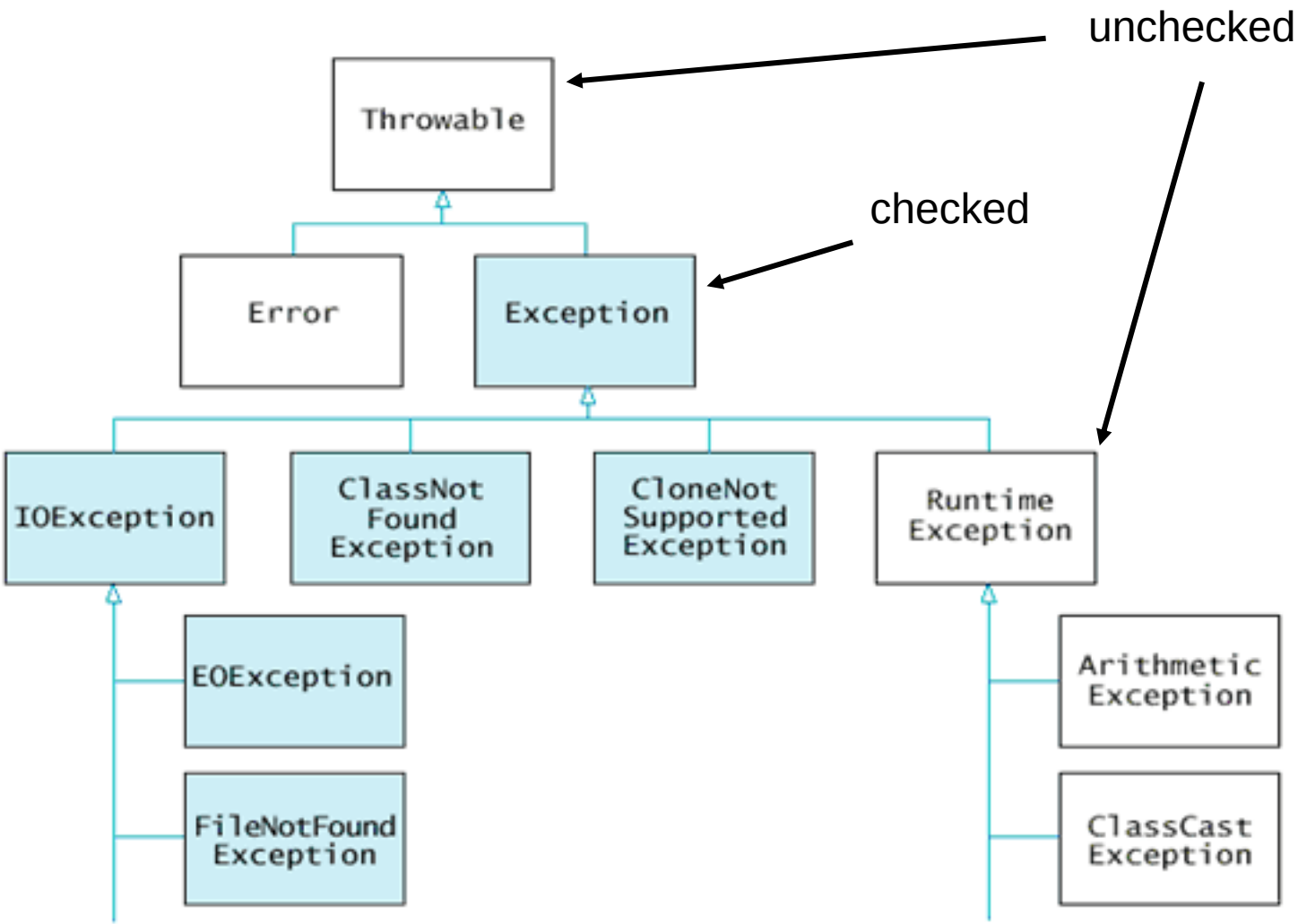
Last program does not handle the exceptions thrown…

As the main method does not handle the exception the program aborts when the exception occurs …

Considered Poor programming practice

Next program catches both type of exceptions but does not attempt to recover from them
– before that, we need to see exception hierarchy

Exception classes Hierarchy



Checked vs. Unchecked Exceptions

An exception is either **Checked** or **Unchecked**.

Checked exception

- a checked exception either must be ***caught*** by a method, or must be listed in the ***throws*** clause of any method that may throw or propagate it.
- that means a checked exception is either handled locally or thrown to others to be dealt externally.
- the compiler will issue an error if a checked exception is not handled appropriately.

Unchecked exception

- An unchecked exception does not require explicit handling.
- It could be processed like a checked exception.
- The only unchecked exceptions are objects of type RuntimeException or any of its descendants.

Catch or Declare Rule

If **a checked exception** may be thrown within a method, then the method must deal with it in one of two ways:

- handle the possible exception by a ***try-catch*** block within the method body.
- declare this possibility of exceptions as a part of method definition by a ***throws*** clause.

If the method declares a possible checked exception, then the caller of this method needs to

- handle the possible exception by a ***try-catch*** block. Or
- also declare the exception and let whoever uses this caller worry about the handling.

try - catch

```
import java.util.*;
class ExceptionHandle {
   public static void main (String[] args)  {

      Scanner stdin = new Scanner(System.in);
      String string1;
      int num1 = 0;

       try  {
         System.out.println ("Input an int number");
         string1 = stdin.nextLine();
         num1 = Integer.parseInt (string1);
       }
       catch (Exception e) {
          System.err.println("Exiting ..");
          System.err.println(e);
       }
       System.out.println("The num is: " + num1);
   }
}
```

The statements that may cause Exception

Catches any exception that is of type Exception or **its subclasses**

Refinement 1 … Catches the exceptions separately

Subclasses of Exception

```
try {
    System.out.println ("Input an integer number");
    string1 = stdin.nextLine();
    num1 = Integer.parseInt (string1);
  }
catch (NumberFormatException nfe) { // wrong input
    System.err.println("Invalid input format. "
                       + "Exiting ...”);
    System.err.println(nfe);

  }
 catch (Exception ioe) {
    System.err.println("Problem in input.  Exiting... ");
    System.err.println(ioe);
 }
```

Whenever an exception occurs, the program goes directly to the first matching ‘catch’ part (if any).

when `string1 = nextLine()` causes an exception, will the next line be executed?

Still unable to recover from the error ! Next Refinement attempts it.

Refinement 2 … Repeat until user enters valid numbers

Repeat until both the number is valid

```
boolean valid = false;
  do  {
      try  {
          System.out.println ("Input an integer number");
          string1 = stdin.readLine();
          num1 = Integer.parseInt (string1);

          valid = true;
      }
      catch (NumberFormatException nfe) {
          System.err.println("Invalid input: try again");
          System.err.println(nfe);
      }
      catch (Exception ioe) {
          System.err.println("Problem in input.  Exiting ..");
          System.err.println(ioe);
      }
  } while (!valid);
```

Initially set to false

All okay up to now - (no exceptions thrown) set valid to true

Repeats loop until valid is set to true

Ordering of catch – matters!

Handling the Scanner InputMismatchException

```
import java.util.*;
public class Week11_7{
  public static void convert(){
      Scanner sc = new Scanner(System.in);
      boolean done = false;
      do {
         try {
            System.out.print("Enter your height (m) : ");
            double ht = sc.nextDouble();
            System.out.println("Your ht in (cm) = " + ht*100);
            done = true;
         }
         catch(InputMismatchException ex){
            System.out.println("Invalid input. Enter again : ");
            sc.nextLine();
         }
      } while (!done);
   }
   public static void main(String args[])
   {  convert();
 } }
```

Week 11 — -bash — 54×13

```
192-168-1-110:Week 11 andysong$ java Week11_7
Enter your height (m) : abc
Invalid input. Enter again :
Enter your height (m) : 1.x
Invalid input. Enter again :
Enter your height (m) : 1.2.3
Invalid input. Enter again :
Enter your height (m) : 1.45
Your ht in (cm) = 145.0
192-168-1-110:Week 11 andysong$
```

Invalid inputs handled

Order of Exception Clauses

A **`try..catch`** block can have multiple **`catch`** statements.

Most specific exception types are the first **`catch`** statements.

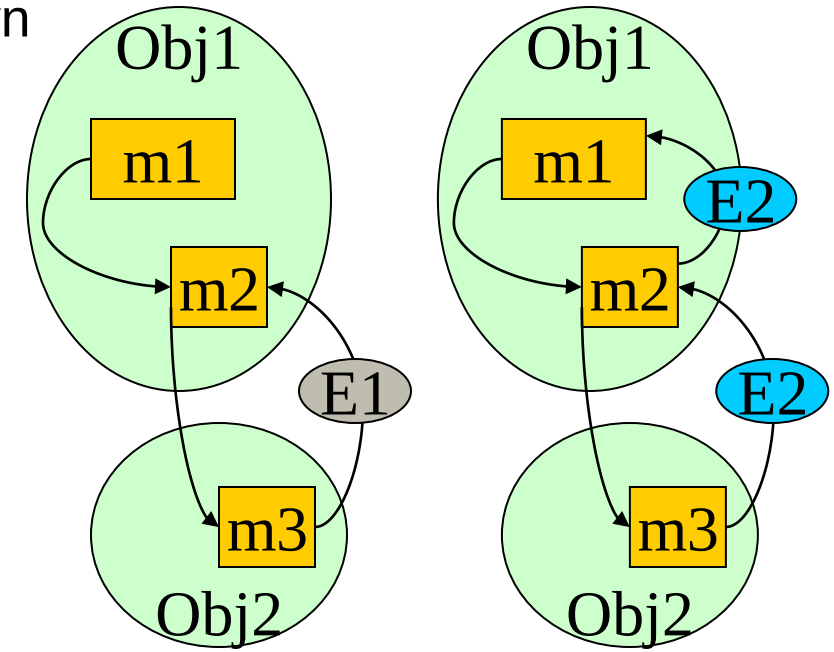
More generic types are placed at last.

```
try {
    // source code here with
    // many possible exceptions
}
catch (NumberFormatException e){
    // handle this exception
}
catch (FileNotFoundException e){
    // handle this exception
}
catch (Exception e){
    // handle this exception
}
```

Propagation of Exceptions

Q. What happens If an exception is thrown and there is no catch clause within the method (m3 of object2?)

A. The exception propagates back to the caller of the method



As method m3 propagates exceptions objects of type E1 and E2 it must add the clause `throws E1, E2`

As method m2 propagates exceptions objects of type E2 it must add the clause `throws E2`

Method m2 catches the exception E1 thrown inside m3

Method m1 catches the exception E2 thrown inside m3

Accounts Example

As a constructor cannot return a value, it may throw an exception.

- For example the Account constructor may throw an exception if initial balance passed is negative.

- SAccount may throw an exception if min-amount > initial-balance

If insufficient balance – withdraw method can throw an exception

```
public Account(String accID, String accountName,
               double amount) throws Exception {
      if (amount < 0) throw new
                  Exception("Balance cannot be negative!!");

      this.accID    = accID;
      name          = accountName;
      balance       = amount;

   }
```

Writing our Own Exceptions

The code below defines our own exception class which is thrown when a user attempts to withdraw a `-ve amount` or when `amount` exceeds balance.

This class has *`instance variables`* for storing information about the error

```
class WithdrawException extends Exception
{
   private String reason;
   private double maxAvailable; //max. withdrawable
   public String getReason() {      return reason; }
   public double maxAvailable() { return maxAvailable;}
   public WithdrawException(String reason) {this.reason = reason;}

   public WithdrawException(String reason,  double maxAvailable) {
      this.reason = reason;
      this.maxAvailable = maxAvailable;
    }
}
```

Why write our own exceptions ... why not just show error messages?

`System.out.println` shows the error message in monitor (standard output device)

What if there is no monitor?

– server machines, or problem with monitor connection…

The catch part will be executed regardless – you can use `System.err` to write in a system log file

```
catch (Exception e) {
        System.err.println("Invalid input..");
```

Our `Account` class withdraw can now be redefined to throw this exception whenever any condition is detected.

Notice the exception object is storing an *error message* and the *maximum amount* that can be withdrawn.

This information may be used to recover from the error.

```
// if insufficient funds WithdrawException will be thrown
public boolean withdraw(double amount) throws WithdrawException
{
   if (amount < 0 )
      throw new  WithdrawException("Negative amount not allowed");
   if (balance <  amount)
      throw new WithdrawException("Amount Not Available", balance);
   balance = balance - amount;
   return true;
}
```

The finally construct

You may want to take some action whether or not exception is thrown (such as closing a file).

The finally is used to handle this situation.

In the code below statements B and D may not be reached but statement C will always be executed.

```
try{   A; // may throw ExType1 or ExType2
       B; //;
}
catch(ExType1 e){
    handling e;
}
finally{
   C; // finalStatements;
}

D ;
```

The finally construct …

```
public class Week11_10{
   public static void main(String args[]){

      try{
         int x = Integer.parseInt("x");
         System.out.println("I see no problem!");
      }
      catch(Exception e){
         System.out.println("Something is wrong!");
         System.err.println(e.getMessage());
      }
      finally{
         System.out.println("I am always on ...");
      }

      System.out.println("-- The End --");
   }
}
```

Rethrowing exceptions

When an exception occurs the enclosing method exits immediately, unless the exception is caught.

If we need to perform some tasks before exiting, we can catch it and then *rethrow* it again.

```
try
{   …....
}
catch(Exception e)
{
    //perform operations before exits;
    throw e;
}
```

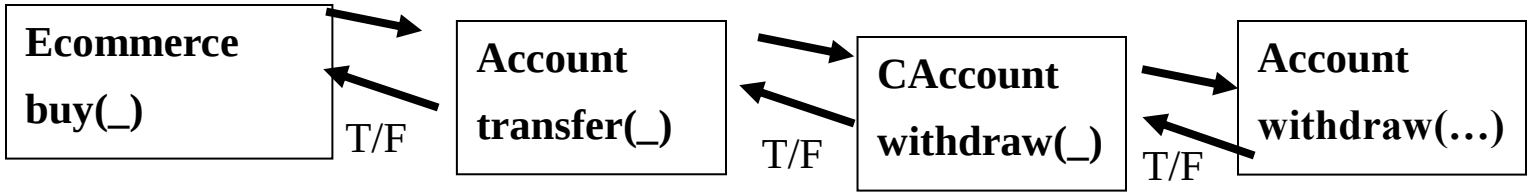
The finally construct …

```
public class Week11_11{
   public static void main(String args[]) throws Exception{

      try{
         int x = Integer.parseInt("x");
         System.out.println("I see no problem!");
      }
      catch(Exception e){
         System.out.println("Something is wrong!");
         System.err.println(e.getMessage());
         throw e;
      }
      finally{
         System.out.println("I am always on ...");
      }

      System.out.println("-- The End --");
   }
}
```

Without exceptions: method can return T/F to notify error




```
public void buy(……){   // Ecommerce class
  do { // may prompt user to enter different values
        …
        if (c1.transfer(c2,amount) == true) {
```

```
  public boolean transfer (……){   //  Account class
        if (withdraw(..) == true) {
```

```
public boolean withdraw (……){   //  CAccount class
      if (super.withdraw(..) == true) {
```

```
public boolean withdraw (……){   //  Account class
      if (balance > amt )     {
            …
            return true;    }
      else return false;
```

Using the Exception mechanism

```
public void buy(……){   // handles exception
 do {   // may prompt user to enter different values
        try { ...
                c1.transfer(c2,100)
        catch(WithdrawException we) {   ……….   }
public void transfer (……) throws WithDrawException
{
         withdraw();
public void withdraw (……) throws WithDrawException
{               super.withdraw();

public void withdraw (……) throws WithDrawException
{
        if (balance > amt )   {
                …
        }
        else throw new WithdrawException(…);
```

Propagating the WithDrawException

More example

```
public class Week11_12{
    public static void A() throws Exception{
        B();
    }

    public static void B() throws Exception{
        C();
    }

    public static void C() throws Exception{
        D();
    }

    public static void D() throws Exception{
        int x = Integer.parseInt("x");
        throw new Exception("Terrible!");
    }

    public static void main(String args[]) throws Exception{
        A();
    }
}
```

RMIT Classification: Trusted

Tips for Writing Unit Tests

- Test thoroughly
  - Normal cases
  - Corner cases
    - Boundary values (e.g., maximum, minimum, values just inside or outside boundaries)

RMIT Classification: Trusted

Tips for Writing Unit Tests (cont.)

- Test only one code unit at a time
- Each use case is tested in one test method
  - A method which has two parameters and returns a value after doing some processing
  - Different use cases might be:
    - First parameter can be null; it throws invalid argument exception.
    - Second parameter can be null; it throws invalid argument exception.
    - Both can be null; it throws invalid argument exception.
    - Both are valid; it returns valid pre-determined output.

RMIT Classification: Trusted

Tips for Writing Unit Tests (cont.)

- Make sure tests are independent and can run in any order
  - @Before and @After to set up pre-requisites (JUnit 4)

RMIT Classification: Trusted

Tips for Writing Unit Tests (cont.)

- Use the most appropriate assertion methods
  - Compare data of primitive types
    - assertTrue (expected == actual) vs assertEquals (expected, actual)
    - assertEquals() gives a useful default error message on failure, like "expected: X; but was Y", but assertTrue() does not.

RMIT Classification: Trusted

Tips for Writing Unit Tests (cont.)

- Do not write tests for getters and setters
- Do not print any statement in your tests
- Name your test clearly and consistently
- Add comments if necessary