Nesneye Yönelik Programlama BLM2012

Öğr. Grv. Furkan ÇAKMAK

Ders Tanıtım Formu ve Konular

BLM2012 Nesneye Yönelik Programlama Hafta 7

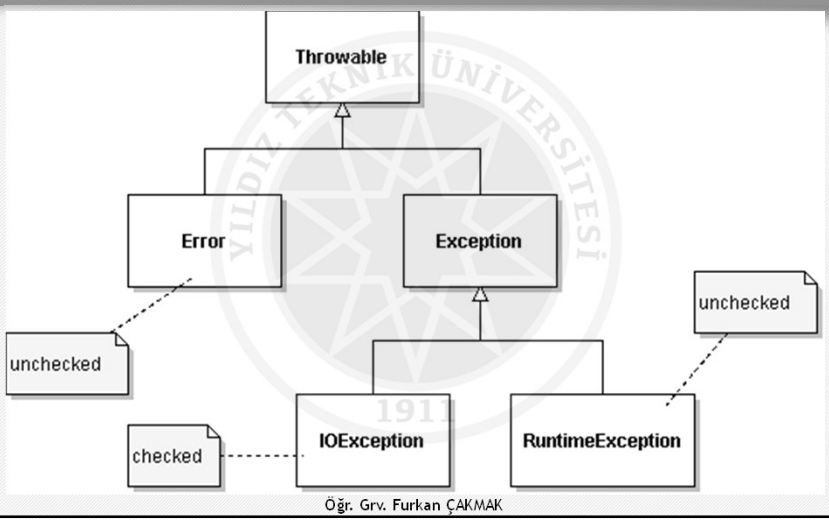
Hafta	Tarih	Konular
1	01.03.2022	Dersin ve Java Dilinin Genel Tanıtımı, Sınıflar, Nesneler, Üyeler, Final ve Static Kavramları
2	08.03.2022	UML Sınıf Şemaları, Kurucular ve Sonlandırıcılar, Denetim Akışı, Nesneleri Oluşturulması
3	15.03.2022	Kurucuların ve Metotların Çoklu Tanımlanması, İlkeller, String ve Math Sınıfları
4	22.03.2022	Sahiplik ve Kullanma İlişkileri, Tek Yönlü ve İki Yönlü Sahiplik Kavramları
5	29.03.2022	Kalıtım, Metotların Yeniden Tanımlanması ve Çoklu Metot Tanımlamadan Farkı
6	05.04.2022	NYP'da Özel Konular: Abstract Classes, Interfaces, Enum Sınıfları
7	12.04.2022	Exception Handling, Unit Test
8	21.04.2022	1. Ara Sınav (10:00-12:00)
9	26.04.2022	Temel Veri Yapılarının Jenerik Sınıflar Eşliğinde Kullanımı (Liste ve Eşleme Yapıları).
10	03.05.2022	Ramazan Bayramı
11	10.05.2022	Dosyalar ve Akışlar ile Çalışmak (Serileştirme ve Ters İşlemi)
12	17.05.2022	Tip dönüşümü, Enum Sınıfları, İç Sınıflar
13	24.05.2022	2. Ara Sınav
14	31.05.2022	Paralel Programlamaya Giriş

EXCEPTION HANDLING

- Some sources of error are:
  - Bugs in JVM
  - Wrong input by the user
  - Buggy code written by us
  - Acts of God
    - A lone and humble programmer cannot control:
      - every aspect of Internet traffic,
      - file access rights,
      - etc.
    - But we should be aware of them and deal with them!
- There are multiple ways of dealing with errors.
  - Boolean returns
  - Form components with error checking mechanisms
  - Exception handling.
- Exception handling is a form of error trapping.

EXCEPTION HANDLING

BLM2012
Nesneye
Yönelik
Programlama
Hafta 7



EXCEPTION HANDLING

- java.lang.Error:
  - indicates serious problems that a reasonable application should not try to catch
    - Internal JVM bugs, etc.
    - java.lang.UnsupportedClassVersionError: Can happen when you move your code between different versions of Eclipse/IDE.
- java.lang.RuntimeException:
  - This is mostly caused by our buggy code
    - java.lang.NullPointerException: We have tried to use an uninitialized object
    - java.lang.IndexOutOfBoundsException: We have tried to access a non-existent member of an array.
    - etc.
- java.io.IOException:
  - Something went wrong during a file operation or a network operation.
  - These operations are always risky, so we must have an alternate plan in case of something goes wrong.
    - If having an alternate plan is a must, than the exception is determined as checked.

EXCEPTION HANDLING

BLM2012
Nesneye
Yönelik
Programlama
Hafta 7

- Handling checked exceptions is done by coding a try - catch block.

```
try {
    /* error-prone methods */;
}
catch( AnException e ) {
    /* Dealing with error */
}
```

- A programmer may opt to not handle a checked exception.
  - However, someone will eventually handle it!

```
aMethod(…) throws AnException {
    /* error-prone methods */
}
```

Öğr. Grv. Furkan ÇAKMAK

EXCEPTION HANDLING

- It is possible to handle multiple exceptions as well:

```
try {
    /* error-prone methods */;
}
catch( AnException e ) {
    /* Dealing with error */
}
catch( AnotherException e ) {
    /* Dealing with error */
}
```


EXCEPTION HANDLING

- What should I do in a catch block?
  - Inform the user about the error with the e.printStackTrace( ) method.
  - Log this error
- If this is a very serious error, you may release some resources and make a "clean exit" in the finally block.
  - Scopes of the try block and the finally block are different. Therefore you cannot access the temporary variables/objects defined in the try block from the finally block. Plan your "clean exit" accordingly.
  - The finally block executes whether an exception is thrown or not.

```
try {
    /* error-prone methods */;
}
catch( AnException e ) {
    /* Dealing with error */
}
catch( AnotherException e ) {
    /* Dealing with error */
}
finally {
    /* make a clean exit */
}
```

EXCEPTION HANDLING


```
public class ExceptionExample01 {
    MyScreenRenderer graphics;
    MyCADfile myFile;
    //Other methods of this class are omitted
    public void parseMyCADfile( String fileName ) {
        try {
         graphics = new MyScreenRenderer();
         myFile = openFile( fileName );
         MyFigure figs[ ] = myFile.readFromFile( );
         drawFigures( figs );
         myFile.close();
        }
        catch( IOException e ) {
         System.out.println("An IO exception has occurred"+
          " while opening or reading from file "+fileName+":"
          + e.toString( ) );
         e.printStackTrace( );
         System.exit(1); //Multithreaded, allows finally to be run
        }
        finally {
         graphics.releaseSources();
        }
    }
}
```


EXCEPTION HANDLING

- You can create your own Exception classes by :
  - inheriting from IOException if you want your exception to be a checked one,
  - inheriting from RuntimeException if you want an unchecked one.

```
public class MyFileFormatException extends IOException {
    public MyFileFormatException( ) {
        super( );
    } //was required in JDK versions older than 5
    public MyFileFormatException( String errorMessage ) {
        super( errorMessage );
        /* Other things to do (optional) */
    } //necessary for informing the user and/or programmer
}
```

EXCEPTION HANDLING

- Throwing an exception:
  - If something terrible may happen during your code, you can throw an exception

```
public class AProgram {
    public void processFile ( ) throws MyFileFormatException{
        some_statements();
        if( an_unexpected_situation )
            throw new MyFileFormatException("... happened");
    }
}
```


EXCEPTION HANDLING - EXAMPLE


```
package nyp09;
import java.io.IOException;
/* @home: Check the needed additions
 * if we had extended this exception
 * from java.lang.RuntimeException
 */
@SuppressWarnings("serial")
public class ImpossibleInfo extends IOException {
       public ImpossibleInfo( String errorMessage ) {
           super(errorMessage);
       }
   }
}
```

EXCEPTION HANDLING – EXAMPLE CON'T


```
package nyp09;
public class Person {
    private String name;
    private int age;

    public Person( String name ) { this.name = name; }
    public String getName( ) { return name; }
    public int getAge( ) { return age; }
    public String toString() {
        return getName() + " " + getAge( );
    }
    public void setAge( int age ) throws ImpossibleInfo {
        if( age < 0 || age > 150 )
           throw new ImpossibleInfo("Impossible age: "+age);
        this.age = age;
    }
}
```


EXCEPTION HANDLING – EXAMPLE CON'T

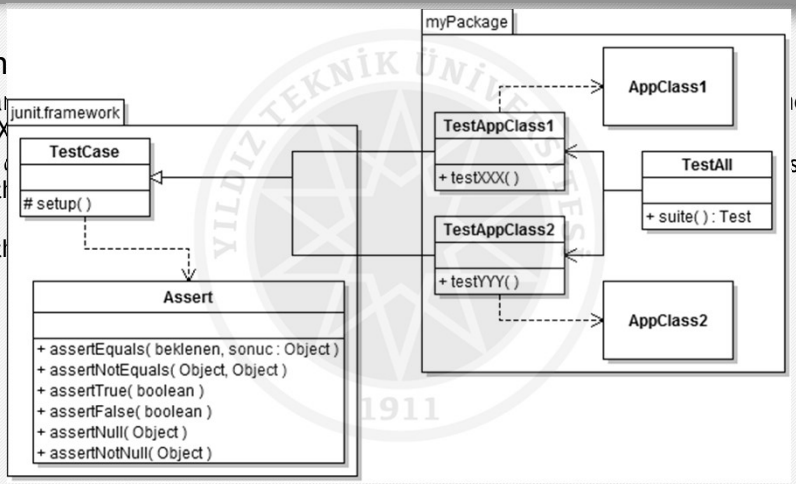

```
package nyp09;
import java.util.*;
public class TestExceptions {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter person's name: ");
        String name = in.nextLine();
        Person insan = new Person(name);
        try {
            System.out.print("Enter age: ");
            int age = in.nextInt();
            insan.setAge(age);
            System.out.println(insan);
        }
        catch (ImpossibleInfo e) {
            e.printStackTrace();
        }
        finally {
            in.close();
        }
    }
}
```

UNIT TESTING – ABOUT UNIT TESTING WITH TOOL SUPPORT

- We have used a separate class having a main method to test our code so far.
  - We have tested the responsibilities of our smallest coding unit, namely the classes we have coded.
  - This is called **'Unit Testing'** in literature.
- Notice that we had to use extensive if-else cases to determine whether a test case was successful or not.
  - Then we had to analyze all the printouts why and where a test case has failed.
- We need to be able to design, execute and analyze the results of our tests.
  - Having a tool for this purpose helps a lot.
- A widely used unit testing tool named **jUnit** can help us.

UNIT TESTING WITH JUNIT

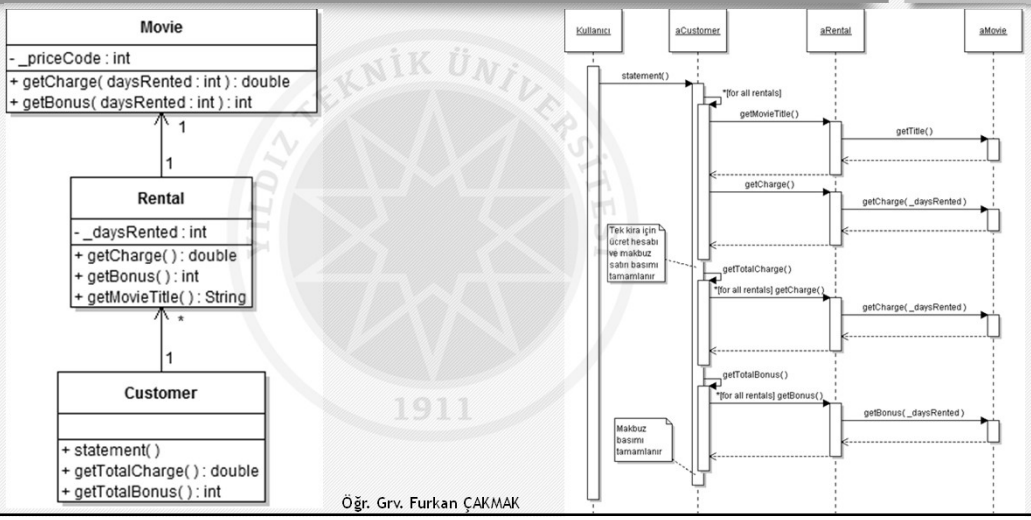
- Preparin
  - Prepa ethods such as testXX
    - You stAll.suite( )
    - The meth



UNIT TESTING

- Preparing test cases with jUnit version 4.X:
  - In addition to preserving backwards compatibility with v3, jUnit v4 adds annotation support:
    - Test classes are no longer needed to inherit from the TestCase class.
    - Test case methods' names no longer need to start with the test word, it is enough to annotate them by using the @test annotation.
      - The setup method is annotated by @before.
      - `@Before`
      - `public void setUp() { /*Preparations*/ }`
      - `@Test`
      - `public void testSomething() { /*Do test*/ }`
    - Exception support is now possible, too: You can test whether a necessary exception is thrown or not, without halting the tests.
      - `@Test(expected=SomeException.class)`
      - `public void testTheException() throws Exception {`
      - `doSomethingThatCreatesTheException();`
      - `}`

UNIT TESTING –
Let's code unit test cases for a simple movie rental example



UNIT TESTING -
Let's code unit test cases for a simple movie rental example


```
package unitTestingByFowler;
import junit.framework.TestCase;
public class TestCustomer extends TestCase {
   private Customer yunus;
   private Movie matrix, monster, surrogate, terminator;
   protected void setUp() {
     yunus = new Customer("Yunus Emre Selçuk");
     matrix = new Movie("The Matrix",Movie.REGULAR);
     monster = new Movie("Monsters, Inc.",Movie.CHILDRENS);
     surrogate = new Movie("Surrogates", Movie.NEW_RELEASE);
     terminator = new Movie("Terminator Salvation",Movie.NEW_RELEASE);
   }
   public void testGetName( ) {
     String sonuc = yunus.getName( );
     assertEquals("Yunus Emre Selçuk",sonuc);
   }
   public void testStatementWhenEmpty( ) {
     String sonuc = yunus.statement();
     String beklenen = "Rental Record for Yunus Emre Selçuk\n";
     beklenen += "Amount owed is 0.0\n";
     beklenen += "You earned 0 frequent renter points";
     assertEquals(beklenen, sonuc);
   }
```

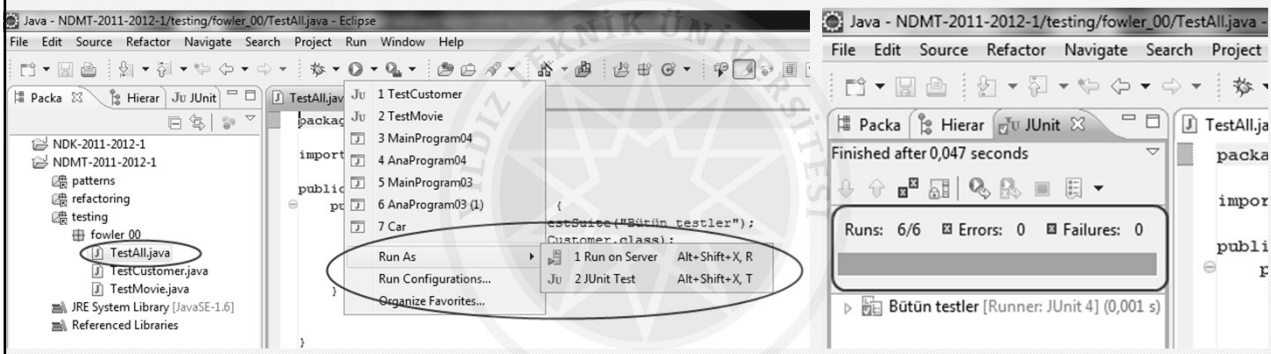

UNIT TESTING -
Let's code unit test cases for a simple movie rental example


```
public void testStatementWithMoviesLongRent( ) {
  yunus.addRental( new Rental(matrix, 3) );
  yunus.addRental( new Rental(monster, 4) );
  yunus.addRental( new Rental(surrogate, 2) );
  String sonuc = yunus.statement();
  String beklenen = "Rental Record for Yunus Emre Selçuk\n";
  beklenen += "\tThe Matrix\t3.5\n";
  beklenen += "\tMonsters, Inc.\t3.0\n";
  beklenen += "\tSurrogates\t6.0\n";
  beklenen += "Amount owed is 12.5\n";
  beklenen += "You earned 4 frequent renter points";
  assertEquals(beklenen, sonuc);
}
public void testStatementWithMoviesShortRent( ) {
  yunus.addRental( new Rental(matrix, 2) );
  yunus.addRental( new Rental(monster, 3) );
  yunus.addRental( new Rental(surrogate, 1) );
  String sonuc = yunus.statement();
  String beklenen = "Rental Record for Yunus Emre Selçuk\n";
  beklenen += "\tThe Matrix\t2.0\n";
  beklenen += "\tMonsters, Inc.\t1.5\n";
  beklenen += "\tSurrogates\t3.0\n";
  beklenen += "Amount owed is 6.5\n";
  beklenen += "You earned 3 frequent renter points";
  assertEquals(beklenen, sonuc);
}
```

UNIT TESTING -
Let's code unit test cases for a simple movie rental example


```
 public void testNewReleaseRentalBonus( ) {
     yunus.addRental( new Rental(surrogate, 2) );
     yunus.addRental( new Rental(terminator, 1) );
     String sonuc = yunus.statement();
     String beklenen = "Rental Record for Yunus Emre Selçuk\n";
     beklenen += "\tSurrogates\t6.0\n";
     beklenen += "\tTerminator Salvation\t3.0\n";
     beklenen += "Amount owed is 9.0\n";
     beklenen += "You earned 3 frequent renter points";
     assertEquals(beklenen, sonuc);
    }
}
package unitTestingByFowler;
import junit.framework.*;
public class TestAll {
   public static Test suite( ) {
      TestSuite suite = new TestSuite("Bütün testler");
      suite.addTestSuite(TestCustomer.class);
      suite.addTestSuite(TestMovie.class);
      return suite;
   }
}
```


UNIT TESTING -
Let's code unit test cases for a simple movie rental example



UNIT TEST with JUNIT

- Disadvantages of manual testing:
  - Our test code is one overly long function that is harder to examine.
    - We may forget what we are testing about.
  - We may need to manually search any "Problem" string in a long and verbose output text.
  - Our problem cases so far did not tell what especially gone wrong.
- We can alleviate these problem by automated test execution and evaluation.
  - Our test code becomes more modular
  - We have a green bar instead! Moreover, assertEquals compares its two parameters and highlight the first difference between them, especially they are String instances.

Sabırla Dinlediğiniz İçin Teşekkürler

BLM2012
Nesneye
Yönelik
Programlama
Hafta 7