INFO1113 / COMP9003
Object-Oriented Programming

Lecture 8

THE UNIVERSITY OF SYDNEY

Contents developed by Tyson Thomas

Acknowledgement of Country

I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

Copyright Warning

Topics: Part A

- **Exceptions**
- **Using exceptions**
- **Enums**
- **Using enums**

Exceptions

Exceptions evolve from the state of the machine or program execution being considered invalid.

- Dividing by 0
- Accessing memory that does not belong to your process
- Null pointer

There are a number of aspects we need to consider with the usage of exceptions.

Exceptions

Different types of except states and severity.

- Checked Exception

This ensures you have handled the exception at compile time, it identifies a state that the programmer must handle.

- Runtime Exception (Unchecked Exception)

It is a state that **should** occur during runtime and you cannot handle nicely prior. Unless you are expecting the code to raise an exception.

- Error (State that cannot be handled)

Errors in Java can be handled by a try-catch block (but it is considered bad practice to catch them) and typically invoked when the state of the program is considered unrecoverable.

Exceptions

Exceptions and errors should be **thrown** when the **precondition** of the method has been violated.

Refer to Chapter 9.1 Basic Exception Handling, p. 706-719 (Java, An Introduction to Problem Solving & Programming, Savitch & Mock)

[1]https://shipilev.net/blog/2014/exceptional-performance/

What’s a precondition?

Exceptions

Pre-condition

A precondition is input that must be within its bounds for it to execute correctly.

For example, if the method expects only a positive integer, any negative integer breaks the constraint afforded by the method.

We may want to invoke an exception **NegativeIntegerException**, **InvalidIntegerException** or something more specific to the problem domain.

Exceptions

Exception classes do not differ from any other classes besides extending from either **Exception**, **RuntimeException** and **Error.**

Syntax:

[public] class <u>ExceptionName</u> extends Exception

[public] class <u>ExceptionName</u> extends RuntimeException

[public] class <u>ErrorName</u> extends Error

Let's quickly revise exceptions briefly

Checked Exception

Let’s examine the following

```java
public static void imGonnaCrash() throws Exception {
  throw new Exception("Definitely crashing!");
}
```

Checked Exception

Let's examine the following

```java
public static void imGonnaCrash() throws Exception {
  throw new Exception("Definitely crashing!");
}
```

Where we throw the exception, typically this is in some kind of if statement.

Since the method can throw a **checked exception** we are required to handle it when we call it.

Checked Exception

Let’s examine the following

```
public static void imGonnaCrash() throws Exception {
    throw new Exception("Definitely crashing!");
}
```

Within our main method we **cannot proceed** with the following.

```
public static void main(String[] args) {
    imGonnaCrash();
}
```

Checked Exception

Let’s examine the following

```
public static void imGonnaCrash() throws Exception {
   throw new Exception("Definitely crashing!");
}
```

We are **forced** to catch it by the compiler;.

```
 public static void main(String[] args) {
   try {
     imGonnaCrash();
   } catch(Exception e) {
     e.printStackTrace();
   }
}
```

Runtime Exception

Let’s examine the following

```
public static void imGonnaCrash() {
    throw new RuntimeException("Definitely crashing!");
}
```

Where the compiler will **not** force the programmer to handle a **RuntimeException.**

```
public static void main(String[] args) {
    imGonnaCrash();
}
```

But at what point should we use them?

Example

Let's examine the following

```
public class Monitor {

 private double refreshRate;
 public final double MAX_REFRESH_RATE;

   public Monitor(double defaultRate, double max) {
      MAX_REFRESH_RATE = max;
      refreshRate = defaultRate;
   }
   public double setRefreshRate(double hz) {
      refreshRate = hz;
      return refreshRate;
   }
}
```

In the following problem we are designing a system to set the **refresh rate** on a **monitor.**

We have implemented a simple method to set the refresh rate of the monitor object.

As part of this problem, the refresh rate should never be above **MAX_REFRESH_RATE**.

However we can see that in the current implementation we can easily **break** this rule.

This is where the pre-condition is violated and our method does nothing about it.

So what should we do?

Example

```
class InvalidRefreshRateException extends Exception {
   public InvalidRefreshRateException() {
      super("Unsupported refresh rate value");
   }
}
public class Monitor {

   private double refreshRate;
   public final double MAX_REFRESH_RATE;

   public Monitor(double defaultRate, double max) {
      MAX_REFRESH_RATE = max;
      refreshRate = defaultRate;
   }
   public double setRefreshRate(double hz) {
      refreshRate = hz;
      return refreshRate;
   }
}
```

This is where we would implement an exception to show where the pre-condition has been violated.

Example

```java
class InvalidRefreshRateException extends Exception {
    public InvalidRefreshRateException() {
        super("Unsupported refresh rate value");
    }
}
public class Monitor {

    private double refreshRate;
    public final double MAX_REFRESH_RATE;

    public Monitor(double defaultRate, double max) {
        MAX_REFRESH_RATE = max;
        refreshRate = defaultRate;
    }
 public double setRefreshRate(double hz) throws InvalidRefreshRateException {
        refreshRate = hz;
        return refreshRate;
    }
}
```

This is where we would implement an exception to show where the pre-condition has been violated.

We will mark the method to throw an **InvalidRefreshRateException**.

Example

```
class InvalidRefreshRateException extends Exception {
   public InvalidRefreshRateException() {
      super("Unsupported refresh rate value");
   }
}
public class Monitor {

   private double refreshRate;
   public final double MAX_REFRESH_RATE;

   public Monitor(double defaultRate, double max) {
      MAX_REFRESH_RATE = max;
      refreshRate = defaultRate;
   }
public double setRefreshRate(double hz) throws InvalidRefreshRateException {
      if(hz < 0 || hz > MAX_REFRESH_RATE)
         throw new InvalidRefreshRateException();
      else
         refreshRate = hz;
      return refreshRate;
}
```

We add the logic to check that **refreshRate** can never be < 0 or > **MAX_REFRESH_RATE**.

Using exceptions

Enums

The java language provides a construct for enumerated types.

Enums are a set of defined instances of the same type. An enum within java allows a finite set of instances to be constructed.

We are unable to create new unique instance (cannot use the **new** keyword) of an enum type.

Refer to Java Language Specification 8.9 , (https://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.9)

Enums

We may use enums in situations where the number of instances are finite or manageable within a sequence of instances.

- A deck of playing cards (52 cards, 4 suits, 13 different ranks)
- Telephone State (Busy, Offline, Awaiting, Dialing)
- Laptop State (On, Sleeping, Off)
- Days of the week (Monday, Tuesday, Wednesday, …)
- Months of a year (January, February, March, April, …)
- Direction (Left, Right, Up, Down)

Enums

Enum is defined similar to a class but there are two variants of the construction.

Syntax:

[public] **enum** <u>EnumName</u>

Enums

Enum is defined similar to a class but there are two variants of the construction.

Syntax:

[public] **enum** EnumName

Variant 1:

```
enum Suit {
  Hearts,
  Diamonds,
  Spades,
  Clubs;
}
```

Enums

Enum is defined similar to a class but there are two variants of the construction.

Syntax:

[public] **enum** <u>EnumName</u>

Variant 1:

```
enum Suit {
  Hearts,
  Diamonds,
  Spades,
  Clubs;
}
```

Each instance of Suit is labelled and can be referred to using the enum identifier.

Enums

Enum is defined similar to a class but there are two variants of the construction.

Syntax:

[public] **enum** <u>EnumName</u>

Variant 1:

```
enum Suit {
  Hearts, //0
  Diamonds, //1
  Spades, //2
  Clubs; //3
}
```

Each instance has an ordinal number within the set. This also allows us to iterator through them

Enums

Enum is defined similar to a class but there are two variants of the construction.

Syntax:

[public] **enum** EnumName

Variant 2:

```
enum Suit {



    private int number;
    private String colour;

    Suit(int n, String colour) {
        this.number = n;
        this.colour = colour;
    }
    public String getColour() {
        return this.colour;
    }
}
```

Properties are defined within type. We can refer to these variables within our methods

We have specified a constructor to be used. Each instance can invoke this and setup its attributes.

Enums

Enum is defined similar to a class but there are two variants of the construction.

Syntax:

[public] **enum** EnumName

Variant 2:

```
enum Suit {
  Hearts(2, "Red"),
  Diamonds(1, "Red"),
  Spades(3, "Black"),
  Clubs(0, "Black");

  private int number;
  private String colour;

  Suit(int n, String colour) {
    this.number = n;
    this.colour = colour;
  }
  public String getColour() {
    return this.colour;
  }
}
```

We are able to initialise each instance at the start of the enum, passing parameters to it

Let’s see how we can use enums

Example

```
enum LightColour {
   Red,
   Green,
   Yellow;
}

public class TrafficLight {

   private LightColour colour;

   public TrafficLight() {
      colour = LightColour.Red; //By default it is Red.
   }

   public void change() {
      if(colour == LightColour.Red) {
         colour = LightColour.Green;
      } else if(colour == LightColour.Yellow) {
         colour = LightColour.Red;
      } else if(colour == LightColour.Green) {
         colour = LightColour.Yellow;
      }
   }
}
```

We have defined our enum type that will be used for **TrafficLight** class.

We have our property within the class and initialised it in the constructor.

Can we do it better?

Example

Of course!

```
enum LightColour {
  Red,
  Green,
  Yellow;

  public LightColour change() {
    if(this == Red) {
      return Green;
    } else if(this == Green) {
      return Yellow;
    } else if(this == Yellow) {
      return Red;
    }
  }
}

public class TrafficLight {

  private LightColour colour;

  public TrafficLight() {
    colour = LightColour.Red; //By default it is Red.
  }

  public LightColour change() {
    colour = colour.change();
    return colour;
  }
}
```

We have moved the change() method logic to the enum type. This allows us to specify it within the type instead of outside.

What else can we do with enums?

Enums

Enums don’t differ all that much from classes and we are able to utilise most class features with them.

- Implement interfaces
- Abstract methods
- Constructor overloading
- Method overloading
- Modify variables within a globally accessible instance.

Let’s go one step further with the traffic lights example

Example

Since we can define abstract methods we can force each instance to contain their own implementation.

```
enum LightColour {




   public abstract LightColour change();
}


public class TrafficLight {

   private LightColour colour;

   public TrafficLight() {
      colour = LightColour.Red; //By default it is Red.
   }

   public LightColour change() {
      colour = colour.change();
      return colour;
   }
}
```

Example

Since we can define abstract methods we can force each instance to contain their own implementation.

```
enum LightColour {
  Red{ public LightColour change() { return Green; } },
  Green{ public LightColour change() { return Yellow; } },
  Yellow{ public LightColour change() { return Red; } };

  public abstract LightColour change();
}
```

We do not need to check. Each instance has its own transition method that specifies its return type.

```
public class TrafficLight {

  private LightColour colour;

  public TrafficLight() {
    colour = LightColour.Red; //By default it is Red.
  }

  public LightColour change() {
    colour = colour.change();
    return colour;
  }
}
```

Take a break!

Topics: Part B

- **Assert Keyword**
- **JUnit**

Assert keyword

Java includes support for the **assert** keyword that allows checking for the truthiness of an expression.

The assert keyword is used in conjunction with a boolean expression.

Refer to Programming with assertions, Preconditions, Postconditions, and Class Invariants, (https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html#usage-conditions)

Assert keyword

Syntax:

assert expression

Assert keyword

Syntax:

assert <u>expression</u>

Example:

```
assert list.size() > 0

assert list.size() == 0 && writtenFiles
```

What happens if the condition is false?

Assert keyword

Assert evaluates an expression and will throw an **AssertionError** if the statement if false.

Syntax:

assert <u>expression</u>

As discussed before, since it throws an **Error** type, it will cause our application to crash.

Assert keyword

You would utilise this feature in an attempt to ensure that your program is sound. We are able to test preconditions, postconditions and anything in between.

Refer to Programming with assertions, Preconditions, Postconditions, and Class Invariants, (https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html#usage-conditions)

Exceptions

Post-condition

A post-condition is where any output from a method is considered to adhere to the requirements of the method.

Simply: What the method promises to do.

For example, A method must return the sum of numbers in a list. Failing this results in the post-condition being false.

Assert keyword

Let's take a look at the following program

```java
public class PackageInstaller {

  //<snipped>

  public void install() {
    preCheck();
    startInstallation();
    postCheck();
  }
```

The main method that will be called by our installer object is the **install()** method. This has simple list of instructions to carry out.

```java
  private void preCheck() {
      File f = new File(path);
      assert f.exists();
      assert key != null;
      assert key.verify(keyInput);
  }
}
```

Each assert potentially will prevent the installer from progressing if it fails the check

The first method being the **preCheck()** method that will need to verify if all dependencies for the installation are satisfied.

```java
private void startInstallation() {

    for(File file : files) {
      try {
        Files.copy(file);
        noFilesWritten++;
      } catch(IOException e) {}
    }
  }
```

If a precheck passes, then move to writing the files to the directory specified.

```java
private void postCheck() {
    assert noFilesWritten > 0;
    assert noFilesWritten == files.size();
}
```

We will run checks after writing the files to ensure that they have been written correctly.

JUnit

Although the compiler performs quite a number of checks for us to ensure we are using types correctly, it doesn't ensure that our program logic is infallible.

When building any meaningful software project you will need to formulate a mechanism of testing the software complies with the requirements.

For more information about JUnit, visit: https://junit.org/junit5/docs/current/user-guide/

JUnit

A common testing framework in the Java ecosystem is **JUnit**. You have written your own test classes to check if your code is performing correctly.

JUnit gives us a simple framework that allows us to mark methods as tests.

JUnit

Black Box Testing - User centric testing, without knowledge of the internals, input is given and compared to match the output of the program.

White Box Testing - This is typically where we employ some unit testing software, to help analyse the internals of the system and test them independently.

JUnit

To set up JUnit you need to acquire **junit.jar** and **hamcrest.jar** files

Within the java ecosystem **.jar** files (Java Archive) are a collection of **.class** files that we can import into our own application. It exposes a whole new set of methods.

Within **JUnit** we have access to variety of **annotations** that allow us to determine an order of execution for some of our methods.

JUnit

The annotations which we can use.

@Test

Simply, this annotates a method as a test method and will be considered part of the results.

@Before

@After

@Before allows a method to initialise any object before a test case is executed.
@After allows execution after a test case.

@BeforeClass

@AfterClass

Similar to the previous pair, we can initialise class level objects when the class is loaded.

Refer to JUnit API Documentation, (https://junit.org/junit5/docs/current/api/overview-summary.html)

JUnit

Testing a for a simple null

```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestMethods{

    @Test
    public void checkForNull() {
        Container a = new Container(null);
        assertNull(a.get());
    }
}
```

@Test, provides annotation of the method that it is a test case.

We can use the JUnit library methods to test if it is true.

JUnit

Our assert methods we have available within our JUnit.

assertTrue(boolean expression)

assertFalse(boolean expression)

They accept boolean expressions that should hold true or false (depending on what you expect the result to be)

assertEquals(expected , actual)

We can check if two **objects** are equal, there are overloaded methods for primitive types and reference types utilise the **.equals** method.

assertNull(object)

assertSame(object1 , object2)

Allows checking of references. We can check if the reference is null or we can check if both variables point to the same allocation.

Compile Test File

Once we have constructed our test case, we will need to compile it with the junit and hamcrest archives.

```
> javac -cp .:junit-4.12.jar:hamcrest-core-1.3.jar MyTestClass.java
```

This is a **classpath** flag that allows us to specify the location of other classes that we can use during compilation

We specify our java archives within this section, separated by **:**

Run Test File

To execute a JUnit class, we need to run the program differently from before.

```
> java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore MyTestClass
```

Because we are utilising classes in a java archive file, we will need to refer to that when executing our program.

Run Test File

To execute a JUnit class, we need to run the program differently from before.

```
> java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar org.junit.runner.JUnitCore MyTestClass
```

```
JUnit version 4.12
..
Time: 0.003

OK (2 tests)
```

Let’s write a test file

See you next time!

Remember to attend the Assignment Q/A session on 18th April

THE UNIVERSITY OF SYDNEY