# INFO1113 Object-Oriented Programming

## Week 8A: Exceptions and Enums

Slides developed by Tyson Thomas

# Copyright Warning

# Topics

- Exceptions (s. 4)
- Using exceptions (s. 16)
- Enums (s. 31)
- Using enums (s. 43)

# Exceptions

We will be exploring exceptions more in depth with Java. Why they are part of the design of the language as well as the advantages and disadvantages with exceptions.

Exceptions can be considered an **except** state within our program.

# 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
- Dereferencing a null pointer
- Unable to parse text
- Out of memory

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)

Is a state that **should** occur 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

What kind of exception should I choose?

- Checked Exceptions, if we know the state can be violated and it is a **common** occurrence, an exception should be generated.

- Runtime Exception, if an error **can** occur and it is unreasonable for the caller to to attempt to prevent the catching of the exception then it can be a runtime exceptions. These kinds of exceptions should be **rare** instances.

- Error, if the state if **unrecoverable** and therefore should crash.

# Exceptions

When to use exceptions?

We need to consider a few aspects within our system.

- When we write a function we may need to make a few assumptions.
  - Using a floating point variable although the range of values should be between 0.0 and 1.0.
  - Only positive values accepted.
  - If the system is in an invalid state, (Order is in the state **Paid** but no payment has occurred).

**Sounds like every method could require an exception?**

# Exceptions

**Yes,** however this would be absurd.

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

There are performance costs associated with throwing exceptions that involve **stack unwinding** and **stack trace construction**[1].

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

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

# Exceptions

As stated previously, we make assumptions with our applications and we want to ensure that these assumptions hold true by constructing a constraint that, when violated will notify the programmer of the error.

We need to also consider when we use an exception and when we **check.**

Exceptions are not a replacement for **if** statements!

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

## 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 of 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 ExceptionName extends Exception

[public] class ExceptionName extends RuntimeException

        [public] class ErrorName extends Error
```

**Let’s quickly revise exceptions briefly**

# Checked Exception

Let’s examine the following

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

# Checked Exception

Let’s examine the following

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

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 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;
    }

}
```

# 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**.

# 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

Let’s examine the following

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

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

```
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;
    }

}
```

# Example

Let’s examine the following

```
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;
    }

}
```

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

# Example

## Let’s examine the following

```
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)

**So where would this be appropriate?**

# Enums

We may run into problems 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. A C-like construction and a Java-like construction.

**Syntax:**

```
[public] enum EnumName
```

# Enums

**Enum** is defined similar to a class but there are two variants of the construction. A C-like construction and a Java-like construction.

**Syntax:**

```
[public] enum EnumName
```

**Example (C-Like):**

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

# Enums

**Enum** is defined similar to a class but there are two variants of the construction. A C-like construction and a Java-like construction.

**Syntax:**

```
[public] enum EnumName
```

**Example (C-Like):**

```
public 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. A C-like construction and a Java-like construction.

**Syntax:**

```
[public] enum EnumName
```

**Example (C-Like):**

```
public 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. A C-like construction and a Java-like construction.

**Syntax:**

```
[public] enum EnumName
```

**Example (Java-Like):**

```
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;
    }
}
```

# Enums

**Enum** is defined similar to a class but there are two variants of the construction. A C-like construction and a Java-like construction.

**Syntax:**

```
[public] enum EnumName
```

**Example (Java-Like):**

```
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

# Enums

**Enum** is defined similar to a class but there are two variants of the construction. A C-like construction and a Java-like construction.

**Syntax:**

```
[public] enum EnumName
```

**Example (Java-Like):**

```
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 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. A C-like construction and a Java-like construction.

**Syntax:**

```
[public] enum EnumName
```

**Example (Java-Like):**

```
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;
    }
}
```

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

**Let’s see how we can use enums**

# Example

Let’s examine the following

```
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;
        }
    }

}
```

# Example

Let’s examine the following

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

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

```
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;
        }
    }

}
```

# Example

Let’s examine the following

```
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 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;
    }

}
```

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

    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
    }

}
```

**See you next time!**