SWEN20003
Object Oriented Software Development

Exceptions

Semester 1, 2019

The Road So Far

- Java Foundations
- Classes and Objects
  - Encapsulation
  - Information Hiding (Privacy)
- Inheritance and Polymorphism
  - Inheritance
  - Polymorphism
  - Abstract Classes
  - Interfaces
- Modelling classes and relationships
- Generics I & II

Lecture Objectives

After this lecture you will be able to:

- Understand what **exceptions** are
- Appropriately handle **exceptions** in Java
- Define and utilise **exceptions** in Java

It is common to make mistakes (errors) when writing code.

Such errors can be categorised as:

- Syntax errors
- Semantic errors
- Runtime errors

Errors

Keyword

Syntax: Errors where what you write isn't legal code; identified by the editor/compiler.

Keyword

Semantic: Code runs to completion, but results in *incorrect* output/operation; identified through software testing (coming soon).

Keyword

Runtime: An error that causes your program to end prematurely (crash and burn); identified through execution.

Common Runtime Errors

- Dividing a number by zero.
- Accessing an element that is out of bounds of an array.
- Trying to store incompatible data elements.
- Using negative value as array size.
- Trying to convert from string data to a specific data value (e.g., converting string abc to integer value).
- File errors:
  - opening a file in read mode that does not exist or no read permission
  - Opening a file in write/update mode which has read only permission.
- Corrupting memory: - common with pointers
- Many more ...

Runtime Error - Example

```java
class NoErrorHandling{
    public static void main(String[] args){
        int a = 7, b = 0;
        System.out.println("The result is " + divide(a,b));
        System.out.println("The program reached this line");
    }
    public static  int divide(int a, int b) {
        return a/b;
    }
}
```

What happens if `b == 0`?

```
Exception in thread "main" java.lang.ArithmeticException: ...
```

Solution 1: Do nothing and hope for the best. Obviously less than ideal.

Runtime Errors

How can we protect against the error?

```java
public int divide(int a, double b) {
    if (b != 0) {
        return a/b;
    } else {
        ???
    }
}
```

```java
if (b != 0) {
    System.out.println("The result is " + divide(a,b));
} else {
    // Print error message and exit or continue
}
```

Solution 2: Explicitly guard yourself against dangerous or invalid conditions, known as *defensive programming*.

Runtime Errors

What are some downsides of solution 2?

- Need to explicitly protect against every possible error condition
- Some conditions don't have a "backup" or alternate path, they're just failures
- Not very nice to read
- Poor abstraction (bloated code)

Runtime Errors

```java
class WithExceptionHandling {
    public static void main(String[] args){
        int a=7,b=0;
        try {
            System.out.println("The result is " + divide(a,b));
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide - b is zero");
        }
        System.out.println("The program reached this line");
    }
    public static int divide(int a, int b) {
        return a/b;
    }
}
```

Solution 3: Use exceptions to catch error states, then recover from them, or gracefully end the program.

Exceptions

Keyword

Exception: An *error state* created by a *runtime error* in your code; an exception.

Keyword

Exception: An object created by Java to *represent* the error that was encountered.

Keyword

Exception Handling: Code that actively protects your program in the case of exceptions.

Exception Handling

```
public void method(...) {
    try {
        <block of code to execute,
                                which may cause an exception>
    } catch (<ExceptionClass> varName) {
        <block of code to execute to recover from exception,
                                or end the program>
    } finally {
        <block of code that executes whether an exception
                                happened or not>
    }
}
```

Exception Handling

Keyword

try: *Attempt* to execute some code that may result in an error state (exception).

Keyword

catch: Deal with the exception. This could be recovery (ask the user to input again, adjust an index) or failure (output an error message and exit).

Keyword

finally: Perform clean up (like closing files) assuming the code didn't exit.

Exception Handling

```java
class WithExceptionCatchThrowFinally{
    public static void main(String[] args){
        int a=7,b=0;
        try {
            System.out.println("The result is " + divide(a,b));
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide - b is zero");
            return;
        }
        finally {
            System.out.println("The program reached this line");
        }
    }
    public static int divide(int a, int b) {
        return a/b;
    }
}
```

Exception Handling - Chaining Exceptions

```
public void processFile(String filename) {
    try {
        ...
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

We can also *chain* catch blocks to deal with different exceptions *separately*

Assess Yourself

Write a method that has the potential to create an `ArithmeticException` and an `ArrayIndexOutOfBoundException`, and implement appropriate exception handling for these cases.

Assess Yourself

```java
public class AverageDifference {
  public static void main(String[] args) {
    int[] a = {1, 2, 3};
    int[] b = {2, 3, 4};
    try {
      System.out.println("Answer=" + averageDifference(a, b));
    } catch (ArithmeticException e) {
      System.out.println("Caught an arithmetic exception");
    } catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Caught an index exception");
    }
  }
  public static int averageDifference(int a[], int b[]) {
    int sumDifference = 0;
    for (int i = 0; i < a.length; i++) {
       sumDifference += a[i] - b[i];
    }
    return sumDifference/a.length;
  }
}
```

Generating Exceptions

Keyword

throw: Respond to an error state by creating an exception object, either already existing or one defined by you.

Keyword

throws: Indicates a method has the potential to create an exception, and can't be bothered to deal with it (Slick stupidly does this for **everything**), or that the exact response varies by application.

Generating Exceptions - Example

Problem Statement: Write a class `Person`, which has attributes name and age, initialized at creation. You must ensure that the name is not null.

```java
public class Person {
    private String name;
    private int age;
    public Person(int age, String name) {
        if (name == null) {
            throw new NullPointerException(
            "Creating person with null name");
        }
        this.age = age;
        this.name = name;
    }
    public static void main(String[] args) {
        Person p1 = new Person(10, "Sarah");
        Person p2 = new Person(12, null);
    }
}
```

Defining Exceptions

We can define our own exceptions!

- Exceptions are classes!
- Most exceptions inherit from an `Exception` class
- All exceptions should have these two constructors, but we can add whatever else we like

Problem Statement: Write a class `Circle`, which has attributes centre and radius, initialized at creation. You must ensure that the radius is greater than zero.

Defining Exceptions

Step 1: Write the exception class.

```java
import java.lang.Exception;
class InvalidRadiusException extends Exception {
    public InvalidRadiusException(double radius){
        super("Radius [" +  radius + "] is not valid");
    }
}
```

Step 2: Write the Circle class.

```java
class Circle {
  private double centreX, centreY, radius;
  public Circle (double x, double y, double r)
        throws InvalidRadiusException {
    centreX=x ; centreY=y;  radius=r;
    if (r <= 0 ) {
        throw new InvalidRadiusException(radius);
    }
  }
}
```

Defining Exceptions

Step 3: Test your class.

```java
class TestCircle {
    public static void main(String[] args)  {
        try{
            Circle c1 = new Circle(10, 10, 100);
            System.out.println("Circle 1 created");
        }
        catch(InvalidRadiusException e)
        {
            System.out.println(e.getMessage());
        }
        try{
            Circle c2 = new Circle(10, 10, -1);
            System.out.println("Circle 2 created");
        }
        catch(InvalidRadiusException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Types of Exceptions

Keyword

Unchecked: Inherit from the `Error` class. Can be safely ignored by the programmer; most (inbuilt) Java exceptions are *unchecked*, because you aren't forced to protect against them.

Keyword

Checked: Inherit from the `Exception` class. Must be handled by the programmer explicitly by the programmer in some way; the compiler gives an error if a checked exception is ignored.

Exception Handling

Catch or Declare

- All checked exceptions must be handled by
  - Enclosing code that can generate exceptions in a try-catch block
  - Declaring that a method may create an exception using the `throws` clause
- Both techniques can be used in the same method, for different exceptions

Using Exceptions

- Should be reserved for when a method encounters an *unusual* or *unexpected* case that cannot be handled easily in some other way