SWEN20003
Object Oriented Software Development

Exceptions

Shanika Karunasekera
karus@unimelb.edu.au

University of Melbourne

The Road So Far

- Java Foundations
  - A Quick Tour of Java
- Object Oriented Programming Foundations
  - Classes and Objects
  - Arrays and Strings
  - Input and Output
  - *Software Tools and Bagel*
  - Inheritance and Polymorphism
  - Interfaces and Polymorphism
- Advanced Object Oriented Programming and Software Design
  - Modelling Classes and Relationships
  - Generics
  - Collections and Maps
  - Design Patterns

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

Errors

It is common to make mistakes (errors) while developing as well as typing a program.

Such mistakes can be categorised as:

- Syntax errors
- Semantic errors
- Runtime errors

Errors

called at development time of compiler

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).

stop running

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 another type (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
- Many more ...

Runtime Error - Example

```
1  class NoErrorHandling {
2      public static void main(String[] args){
3          int n1 = 1, n2 = 0;
4          System.out.println("The result is " + divide(n1, n2));
5          System.out.println("The program reached this line");
6      }
7      public static  int divide(int n1, int n2) {
8          return n1/n2;
9      }
10 }
```

What happens if `n2 == 0`?

```
1  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?

```
1  public int divide(int n1, double n2) {
2      if (n2 != 0) {
3          return n1/n2;
4      } else {
5          ???
6      }
7  }
```

```
1  if (n2 != 0) {
2      divide(n1, n2);
3  } else {
4      // Print error message and exit or continue
5  }
```

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

→ check condition

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

```
1  class WithExceptionHandling {
2      public static void main(String[] args){
3          int n1 = 1, n2 = 0;
4
5          try {
6              System.out.println("The result is " + divide(n1, n2));
7          } catch (ArithmeticException e) {
8              System.out.println("Cannot divide - n2 is zero");
9          }
10         System.out.println("The program reached this line");
11     }
12
13     public static int divide(int n1, int n2) {
14         return n1/n2;
15     }
16 }
```

code would result an error

know it is error handling clear.

handle error

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 object

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

```
1  public void method(...) {
2      try {
3          <block of code to execute,
4                                  which may cause an exception>
5      } catch (<ExceptionClass> varName) {
6          <block of code to execute to recover from exception,
7                                  or end the program>
8      } finally {
9          <block of code that executes whether an exception
10                                 happened or not>
11     }
12 }
```

Exception Handling

Keyword

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

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 n1 = 1, n2 = 0;

        try {
            System.out.println("The result is " + divide(n1, n2));
        } catch (ArithmeticException e) {
            System.out.println("Cannot divide - n2 is zero");
        } finally {
            System.out.println("The program reached this line");
        }
    }

    public static int divide(int n1, int n2) {
        return n1/n2;
    }
}
```

Exception Handling - Chaining Exceptions

```
1  public void processFile(String filename) {
2      try {
3          ...
4      } catch (FileNotFoundException e) {
5          e.printStackTrace();
6      } catch (IOException e) {
7          e.printStackTrace();
8      }
9  }
```

only one will be executed

We can also *chain* catch blocks to deal with different exceptions *separately*. The most "specific" exception (subclasses) come first, with "broader" exceptions (superclasses) listed lower.

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

```
1  public class AverageDifference {
2    public static void main(String[] args) {
3      int[] n1 = {1, 2, 3};
4      int[] n2 = {2, 3, 4};
5
6      try {
7        System.out.println("Answer = " + averageDifference(n1, n2));
8      } catch (ArithmeticException e) {
9        System.out.println("Caught an arithmetic exception");
10     } catch (ArrayIndexOutOfBoundsException e) {
11       System.out.println("Caught an index exception");
12     }
13   }
14
15   public static int averageDifference(int n1[], int n2[]) {
16     int sumDifference = 0;
17     for (int i = 0; i < n1.length; i++) {
18        sumDifference += n1[i] - n2[i];
19     }
20     return sumDifference/n1.length;
21   }
22 }
```

if n1={1,2,3} n2={2,1,2,3}
↓
since iterate n1.length
so not a runtime error
⇒ will give an answer
but a semantic error
⇒ incorrect answer

→ length = 0

→ n2 shorter than n

Generating Exceptions

event-driven programming

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, or that the exact response varies by application.

Assess Yourself

Write a method that has the potential to create a `NullPointerException`, and throws an exception if its argument is null.

creat a placeholder but no an object

```
1  public class Person {
2      private String name;
3      private int age;
4
5      public Person(int age, String name) {
6          if (name == null) {
7              throw new NullPointerException("Creating person with null name");
8          }
9          this.age = age;
10         this.name = name;
11     }
12
13     public static void main(String[] args) {
14         Person p1 = new Person(10, "Sarah");
15         System.out.println("Created object p1");
16         Person p2 = new Person(12, null);
17         System.out.println("Created object p2");
18     }
19 }
```

→ object should not be created if name is null

↑ terminate and return exception

Assess Yourself

Can we improve the previous program so that it does not die on the exception (exit gracefully)?

```
1  public class PersonWithExcpHandling {
2      private String name;
3      private int age;
4      public PersonWithExcpHandling(int age, String name) {
5          if (name == null) {
6              throw new NullPointerException("Creating person with null name");
7          }
8          this.age = age;
9          this.name = name;
10     }
11     public static void main(String[] args) {
12         try {
13             Person p1 = new Person(10, "Sarah");
14             System.out.println("Created object p1");
15             Person p2 = new Person(12, null);
16             System.out.println("Created object p2");
17         } catch(NullPointerException e) {
18             System.out.println("Failed to create object");
19         }
20     }
21 }
```

limitation to put in one try block is if one error occurs the rest code will not run

e.getMessage()

deal with exception
run into error
↓
find proper exception
↓
create object of that type
↓
throw it

Defining Exceptions

What if we discover a new "type" of problem?

We can define our own exceptions!

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

Defining Exceptions

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

Defining Exceptions

Step 1: Write the exception class.

```
1  import java.lang.Exception;
2
3  public class InvalidRadiusException extends Exception {
4      public InvalidRadiusException() {
5          super("Radius is not valid");
6      }
7
8      public InvalidRadiusException(double radius){
9          super("Radius [" +  radius + "] is not valid");
10     }
11 }
```

message for user

Defining Exceptions - Throwing

Step 2: Write the Circle class.

```
1  public class Circle {
2      private double centreX, centreY;
3      private double radius;
4
5      public Circle (double centreX, double centreY, double radius)
6          throws InvalidRadiusException {
7          if (r <= 0 ) {
8              throw new InvalidRadiusException(radius);
9          }
10         this.centreX = centreX;
11         this.centreY = centreY;
12         this.radius = radius;
13     }
14 }
```

inform users

throw exception

declare an exception

you cannot write the code without the try block if error is defined

Defining Exceptions - Handling

Step 3: Test your class.

```
1  public class TestCircle {
2      public static void main(String[] args)  {
3          try {
4              Circle c1 = new Circle(10, 10, 100);
5              System.out.println("Circle 1 created");
6
7              Circle c2 = new Circle(10, 10, -1);
8              System.out.println("Circle 2 created");
9          } catch(InvalidRadiusException e) {
10             System.out.println(e.getMessage());
11         }
12     }
13 }
```

```
1  "Circle 1 created"
2  "Radius [-1] is not valid"
```

Defining Exceptions - Handling

Following is another way you handle the exception, but it is better to handle specific exceptions!

```
1  public class TestCircle5 {
2      public static void main(String[] args)  {
3          try{
4              Circle c1 = new Circle(10, 10, 100);
5              System.out.println("Circle 1 created");
6              Circle c2 = new Circle(10, 10, -1);
7              System.out.println("Circle 2 created");
8          } catch(Exception e) {
9              System.out.println(e.getMessage());
10         }
11     }
12 }
```

since your exception is a subclass

⇒ not recommend

```
1  "Circle 1 created"
2  "Radius [-1] is not valid"
```

Defining Exceptions - Handling

A better way to do this is:

```
1  public class TestCircle6 {
2      public static void main(String[] args)  {
3          try{
4              Circle c1 = new Circle(10, 10, 100);
5              System.out.println("Circle 1 created");
6              Circle c2 = new Circle(10, 10, -1);
7              System.out.println("Circle 2 created");
8          } catch(InvalidRadiusException e) {
9              System.out.println(e.getMessage() + " error 1");
10         } catch(Exception e){
11             System.out.println(e.getMessage() + " error 2");
12         }
13     }
14 }
```

first specific type of error

may have other exception

```
1  "Circle 1 created"
2  "Radius [-1] is not valid error 1"
```

Types of Exceptions

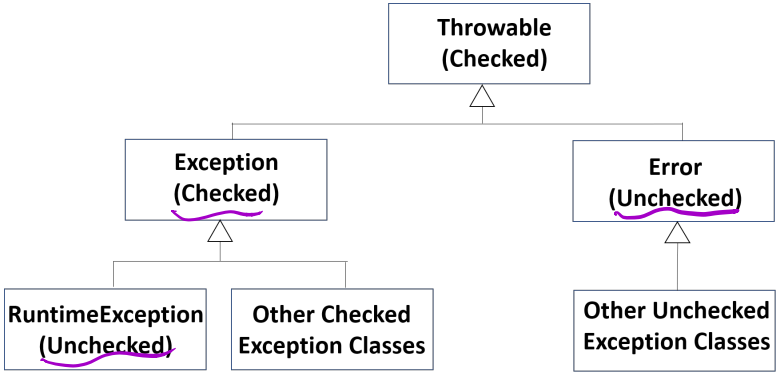
Keyword

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

Keyword

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

Checked and Unchecked Exceptions



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 → just running time error

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

public static void main(String[] args) throws ... Exception

give up running if have exception

possibility to happen

Try With

```
1  public void processFile(String filename) {
2      try (BufferedReader reader = ...) {
3          ...
4      } catch (FileNotFoundException e) {
5          e.printStackTrace();
6      } catch (IOException e) {
7          e.printStackTrace();
8      }
9  }
```

try with resources

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