Revised: 9/13/2019

Exception Handling

Michael C. Hackett

Computer Science Department

Community College of Philadelphia

Lecture Topics

- Exception Handling
  - Unchecked and Checked
  - try…catch statements
- Error Messages
  - Default Error Messages
  - Call Stack
- Uncaught Exceptions
- Handling Multiple Exceptions
- The Exception Object
- Finally Clauses
- Throwing Exceptions

Colors/Fonts

- Local Variable Names – Brown
- Primitive data types – Fuchsia
- Literals – Blue
- Keywords – Orange
- Object names – Green
- Operators/Punctuation – Black
- Field Names – Lt Blue
- Method Names – Purple
- Parameter Names – Gold
- Comments – Gray
- Package Names – Pink

Source Code – **Consolas**
Output – Courier New

Exception Handling

- An Exception is an object that is generated as the result of an error or an unexpected event.
  - When that happens, we say that an exception has been ***thrown***.

- It is the programmer's responsibility to write code that detects and handles exceptions.

- Unhandled exceptions will crash a program.

Exception Handling

- An ***exception handler*** is a section of code that gracefully responds to exceptions.
  - The process of anticipating and responding to exceptions is called ***exception handling***.

- The ***default exception handler*** deals with any unhandled exceptions.
  - It prints an error message and stops the program.

Unchecked Exceptions

- Java has two types of exceptions, checked and unchecked.
- An ***unchecked exception*** is an exception that the compiler does not check for.
  - Executing the code below will result in an ArrayIndexOutOfBoundsException.

```
char[] letters = {'a', 'b', 'c'};
System.out.println(letters[3]);
```

  - The compiler doesn't check to see if the index used is valid, yet the code will still compile.
  - The result will be an unchecked exception being thrown when the program runs.

Checked Exceptions

- A ***checked exception*** is an exception that the compiler *does* check for.
  - Checked exceptions have to be handled or the source code won't compile.

- Checked exceptions are identified by a method or constructor explicitly stating that they may throw an exception.
  - The method/constructor has a "throws" clause.

try...catch statements

- To handle possible exceptions, use a try…catch statement.

```
try {
  try block statements
}
catch(ExceptionType name) {
  catch block statements
}
```

- First, the keyword `try` indicates a block of code that will be attempted (the curly braces are required).
  - This block of code is known as a ***try block***.

try...catch statements

- The try block contains one or more statements that, when executed, can potentially throw an exception.
  - After the try block, at least one catch clause is required.

- The application will not halt if the try block throws an exception.
  - Instead, the code in the catch block will be executed.

catch clauses

- A catch clause begins with the keyword **`catch`**:

```
catch(ExceptionType name) {
   catch block statements
}
```

  - *ExceptionType* is the name/type of an exception object.
  - *name* is a variable name which will reference the exception object.
- The code that inside the catch clause is known as a ***catch block***.
  - The code in the catch block is executed if the try block throws that particular type of exception.

Handling Exceptions

- This code is designed to handle a ArrayIndexOutOfBoundsException, if it is thrown by statements in the try block.

```
try {
  values[99] = 34;
}
catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("Index does not exist in the array.");
}
```

- The Java Virtual Machine searches for a catch clause that can deal with the exception.

Default Error Messages

- Each exception object has a method named **getMessage** that can be used to retrieve the default error message for the exception.

```
try {
  value = new Integer.parseInt("abcd");
}
catch(NumberFormatException e) {
  System.out.println("An exception occurred:" + e.getMessage());
}
```

```
An exception occurred: For input string: "abcd"
```

Tracing the Call Stack

- The ***call stack*** is a list of the methods that are currently executing.

- A ***stack trace*** is a list of all the methods in the call stack.

- It can indicate:
  - The method that was executing when an exception occurred and
  - All of the methods that were called in order to execute that method.

Tracing the Call Stack

- Each exception object has a method named **printStackTrace** that will print the exception's stack trace to the console.

```
try {
    value = new Integer.parseInt("abcd");
}
catch(NumberFormatException e) {
    e.printStackTrace();
}
```

```
java.lang.NumberFormatException: For input string: "abcd"
    at sun.misc.FloatingDecimal.readJavaFormatString(FloatingDecimal.java:2043)
    at sun.misc.FloatingDecimal.parseDouble(FloatingDecimal.java:110)
    at java.lang.Double.parseDouble(Double.java:538)
    at SampleCode.TestProgram.main(b_ExceptionMessage.java:18)
```

Uncaught Exceptions

- When an exception is thrown, it cannot be ignored.
- It must be handled by the program, or by the default exception handler.
- When the code in a method throws an exception:
  - Normal execution of that method stops.
  - The JVM searches for a compatible exception handler inside the method.

Uncaught Exceptions

- If there is no exception handler inside the method:
  - Control of the program is passed to the previous method in the call stack.
  - If that method has no exception handler, then control is passed again, up the call stack, to the previous method.
- If control reaches the main method:
  - The main method must either handle the exception, or
  - The program is halted and the default exception handler handles the exception.

Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception.

- Multiple catch clauses can be written for each type of exception that could potentially be thrown.

Handling Multiple Exceptions

```
try {
   value = new Integer.parseInt(someString);
   numbers[7] = 15;
}
catch(NumberFormatException e) {
   System.out.println("Unable to parse value.");
}
catch(ArrayIndexOutOfBoundsException e) {
   System.out.println("Invalid array index.");
}
```

Handling Multiple Exceptions

```
try {
  value = new Integer.parseInt(someString);
  numbers[7] = 15;
}
catch(NumberFormatException e) {
  System.out.println("Unable to parse value.");
}
catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("Invalid array index.");
}
```

The Exception Object

- The Exception object is the parent object of many types of exceptions.
  - In one way or another, all exceptions are traced back to the Exception object.

- Catching an Exception object allows us to catch every type of exception at once.
  - Which is sometimes good and sometimes bad.

The Exception Object

```
try {
   ...
}
catch(Exception e) {
   System.out.println("An exception occurred.");
}
```

- Regardless of the code in the try block, **any** exception thrown will be handled by the catch clause.
- Sure, the program won't halt but we have no idea:
  - What caused the exception.
  - How the program should respond.

The Exception Object

- A better use of the Exception object is to make it the last catch clause.

- This will allow other catch clauses to handle specific types of exceptions.
  - The Exception catch will handle any other, possibly unforeseen, exceptions that are thrown.

The Exception Object

```
try {
  value = new Integer.parseInt(someString);
  numbers[7] = 15;
}
catch(NumberFormatException e) {
  System.out.println("Unable to parse value.");
}
catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("Invalid array index.");
}
catch(Exception e) {
  System.out.println("Unexpected exception.");
  e.printStackTrace();
}
```

finally clauses

- A try…catch statement may have an optional `finally` clause.

- If present, the finally clause must appear after all of the catch clauses.

- The ***finally block*** is one or more statements that are always executed after the try block has executed AND after any catch blocks have executed if an exception was thrown.
  - In other words, the statements in the finally block execute whether an exception occurs or not.

finally clauses

```
try {
  values[99] = 34;
}
catch(ArrayIndexOutOfBoundsException e) {
  System.out.println("Index does not exist in the array.");
}
finally {
  //Executes the code in here regardless if an exception
  //was thrown in the try block or not.
}
```

Throwing Exceptions

- If a method is unable to complete its function/purpose, the method should throw an exception.
- To throw an exception use the following statement:

```
throw new Exception("Your exception's message");
```

- The above statement throws a generic Exception object.
  - You can throw any type of exception you want (some may need to be imported):

```
throw new NumberFormatException("Your exception's message");
throw new IOException("Your exception's message");
```

- This creates a checked exception situation.
  - The exception you throw must be handled at some point up the call stack.

Throwing Exceptions

- If a method might throw an exception, the method header needs a ***throws clause***.
  - The class won't compile without it.
  - The exception type in the throws clause must match the type thrown.

```
private int validateGear(int g) throws Exception {
    if(g >= 1 && g <= 10) {
        return g; //Valid Value
    }
    throw new Exception("Gear value is invalid: " + g);
}
```

Throwing Exceptions

- Now, any method (or constructor) that calls the validateGear method must:
  - **Handle the exception itself**, or
  - Throw it up the call stack.

```
public void setCurrentGear(int currentGearIn) {
    try {
        currentGear = validateGear(currentGearIn);
    }
    catch(Exception e){
        //Maybe print an error message or set currentGear to 1
    }
}
```

Throwing Exceptions

- Now, any method (or constructor) that calls the validateGear method must:
  - Handle the exception itself, or
  - **Throw it up the call stack.**

```
public void setCurrentGear(int currentGearIn) throws Exception {
    currentGear = validateGear(currentGearIn);
}
```

Throwing Exceptions

```
public class BicycleTest {

    public static void main(String[] args) {
        Bicycle testBike = new Bicycle();
        try {
            testBike.setCurrentGear(700);
        }
        catch(Exception e){
             System.out.println("Error: " + e.getMessage());
        }
        System.out.println("testBike is in gear " + testBike.getCurrentGear());
    }

}
```

```
Error: Gear value is invalid: 700
testBike is in gear 0
```

Throwing Exceptions (Call Stack)