Altar Code Labs

Best practices and error handling

Java SE 3/4

UP ACADEMY
LEARNING FOR SUCCESS

CONTENT

1. Best Practices
    - Encapsulation
    - Access modifiers
    - Constructors
    - Java date time
2. Error Handling
    - Exceptions
    - Error handling

Best Practices

- *Encapsulation*
- *Access modifiers*
- *Constructors*
- *Java date time*

Overview

- Encapsulation means hiding object fields by making all fields private:
  - Use getter and setter methods
  - In setter methods, use code to ensure that values are valid
- Encapsulation mandates programming to the interface:
  - Data type of the field is irrelevant to the caller method
  - Class can be changed as long as interface remains same
- Encapsulation encourages good object-oriented (OO) design

public Modifier

```
public class Elevator {
   public boolean doorOpen=false;
   public int currentFloor = 1;
   public final int TOP_FLOOR = 10;
   public final int MIN_FLOOR = 1;

     ... < code omitted > ...

   public void goUp() {
      if (currentFloor == TOP_FLOOR) {
         System.out.println("Cannot go up further!");
      }
      if (currentFloor < TOP_FLOOR) {
         currentFloor++;
         System.out.println("Floor: " + currentFloor);
      }
   }
}
```

Dangers of Accessing a public Field

```
Elevator theElevator = new Elevator();

theElevator.currentFloor = 15;
```

Could cause a problem!

private Modifier

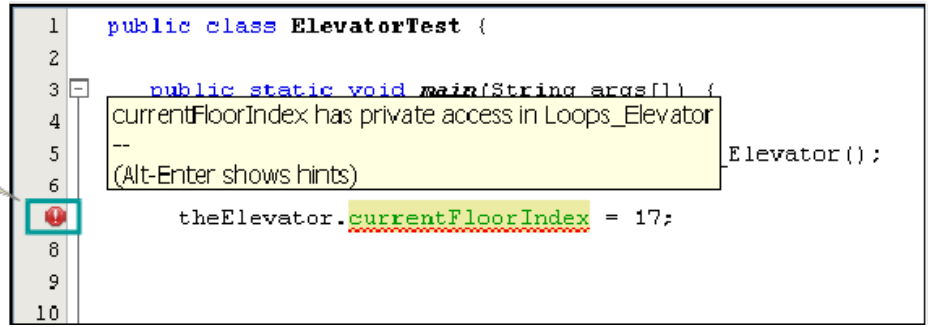
```
public class Elevator {
   private boolean doorOpen=false;
   private int currentFloor = 1;
   private final int TOP_FLOOR = 10;
   private final int MIN_FLOOR = 1;

     ... < code omitted > ...

   public void goUp() {
      if (currentFloor == TOP_FLOOR) {
         System.out.println("Cannot go up further!");
      }
      if (currentFloor < TOP_FLOOR) {
         currentFloor++;
         System.out.println("Floor: " + currentFloor);
      }
   }
}
```

None of these fields can now be accessed from another class using dot notation.

Trying to Access a private Field

```
Elevator theElevator = new Elevator();

theElevator.currentFloor = 15;  <-- not permitted
```

IDE will show an error. You can get an explanation if you place your cursor here.



Get and Set Methods

```
public class Shirt {
    private int shirtID = 0; // Default ID for the shirt
    private String description = "-description required-"; // default

    // The color codes are R=Red, B=Blue, G=Green, U=Unset
    private char colorCode = 'U';
    private double price = 0.0; // Default price for all items

    public char getColorCode() {
      return colorCode;
    }
    public void setColorCode(char newCode) {
      colorCode = newCode;
    }
    // Additional get and set methods for shirtID, description,
    // and price would follow

  } // end of class
```

Using Getter and Setter Methods

```
public class ShirtTest {
      public static void main (String[] args) {
      Shirt theShirt = new Shirt();
      char colorCode;

   // Set a valid colorCode
      theShirt.setColorCode('R');
      colorCode = theShirt.getColorCode();
      // The ShirtTest class can set and get a valid colorCode
      System.out.println("Color Code: " + colorCode);

      // Set an invalid color code
      theShirt.setColorCode('Z'); <-- not a valid color code
      colorCode = theShirt.getColorCode();
      // The ShirtTest class can set and get an invalid colorCode
      System.out.println("Color Code: " + colorCode);
}
```

Setter Method with Checking

```
public void setColorCode(char newCode) {
    switch (newCode) {
        case 'R':
        case 'G':
        case 'B':
            colorCode = newCode;
            break;
        default:
            System.out.println("Invalid colorCode. Use R, G, or B");
    }
}
```

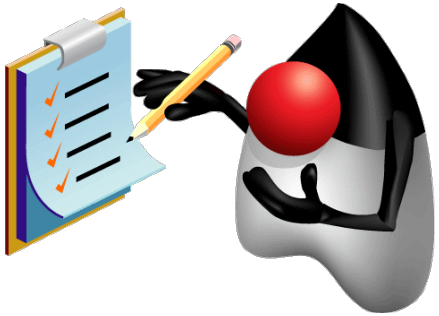
Using Setter and Getter Methods

```
public class ShirtTest {
     public static void main (String[] args) {
     Shirt theShirt = new Shirt();
     System.out.println("Color Code: " + theShirt.getColorCode());

     // Try to set an invalid color code
     Shirt1.setColorCode('Z'); ← not a valid color code
     System.out.println("Color Code: " + theShirt.getColorCode());
}
```

Output:

```
Color Code: U ← Before call to setColorCode() – shows default value
Invalid colorCode. Use R, G, or B ← call to setColorCode prints error message
Color Code: U ← colorCode not modified by invalid argument passed to setColorCode()
```

Encapsulation: Summary



Encapsulation protects data:

- By making all fields private
  - Use getter and setter methods
  - In setter methods, use code to check whether values are valid
- By mandating programming to the interface
  - Data type of the field is irrelevant to the caller method.
  - Class can be changed as long as interface remains same.
- By encouraging good OO design

Controlling Access to Members of a Class

- Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:
  - At the class level – **public** or **package-private** (no explicit modifier)
  - At the member level – **public**, **private**, **protected** or **package-private** (no explicit modifier)

Top level modifiers

- A class may be declared with the modifier ***public***, in which case that class is **visible to all classes everywhere**

- If a class has **no modifier** (the default, also known as **package-private**), it is visible **only within its own package**

Member level modifiers

The following table shows the access to members permitted by each modifier:

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Initializing a Shirt Object

```
public class ShirtTest {
     public static void main (String[] args) {
     Shirt theShirt = new Shirt();

   // Set values for the Shirt
     theShirt.setColorCode('R');
     theShirt.setDescription("Outdoors shirt");
     theShirt.price(39.99);
}
```

Constructors

- Constructors are method-like structures in a class:
  - They have the same name as the class
  - They are usually used to initialize fields in an object
  - They can receive arguments
  - They can be overloaded
  - They don't declare a return type (it is always an object of the type of the class)
- All classes have at least one constructor:
  - If there are no explicit constructors, the JVM supplies a default no-argument constructor
  - If there are one or more explicit constructors, no default constructor will be supplied

Creating Constructors

Syntax:

```
[modifiers] class ClassName {
        [modifiers] ClassName([arguments]) {
            code_block
        }
}
```

Creating Constructors

```
public class Shirt {
  private int shirtID = 0; // Default ID for the shirt
  private String description = "-description required-";
  // The color codes are R=Red, B=Blue, G=Green, U=Unset
  private char colorCode = 'U';
  private double price = 0.0; // Default price all items

  // This constructor takes one argument
  public Shirt(char colorCode ) {
      setColorCode(colorCode);
  }
```

```
public class ShirtTest {
     public static void main (String[] args) {
         Shirt theShirt = new Shirt('G');
         theShirt.display();
     }
```

Multiple Constructors

```
public class Shirt {
  ... < declarations for field omitted > ...

  // No-argument constructor
  public Shirt() {
     // You could add some default processing here
  }
  // This constructor takes one argument
  public Shirt(char colorCode ) {
      setColorCode(colorCode);
  }
  public Shirt(char colorCode, double price) {

      this(colorCode);
      setPrice(price);
  }
```

If required, must be added explicitly

Chaining the constructors

Java date time

- Date time until java 7 had several flaws, that led to the usage of external libraries like JodaTime
- Java 8 finally solved this problem bringing a clear, immutable and extensible date time API
- API naming conventions:

Prefix	Method Type	Use
`of`	static factory	Creates an instance where the factory is primarily validating the input parameters, not converting them.
`from`	static factory	Converts the input parameters to an instance of the target class, which may involve losing information from the input.
`parse`	static factory	Parses the input string to produce an instance of the target class.
`format`	instance	Uses the specified formatter to format the values in the temporal object to produce a string.
`get`	instance	Returns a part of the state of the target object.
`is`	instance	Queries the state of the target object.
`with`	instance	Returns a copy of the target object with one element changed; this is the immutable equivalent to a `set` method on a JavaBean.
`plus`	instance	Returns a copy of the target object with an amount of time added.
`minus`	instance	Returns a copy of the target object with an amount of time subtracted.
`to`	instance	Converts this object to another type.
`at`	instance	Combines this object with another.

Date time classes

- Enums (both these enums provide a number of methods, similar to the methods provided by the temporal-based classes):
  - **DayOfWeek**: MONDAY (1) through SUNDAY (7)
  - **Month**: JANUARY (1) through DECEMBER (12)
- Classes:
  - **YearMonth**: represents the month of a specific year
  - **MonthDay**: represents the day of a particular month
  - **Year**: represents a year
  - **LocalDate**: represents a year-month-day in the ISO calendar
  - **LocalTime**: represents a time of the day
  - **LocalDateTime**: represents both date and time
  - **Period**: defines an amount of time with date-based values (years, months, days)

Usage examples

```
LocalDate date = LocalDate.of(1980, Month.DECEMBER, 24);
LocalTime time = LocalTime.now();
LocalDateTime dateTime = LocalDateTime.of(date, time);
// LocalDate invalid = LocalDate.of(2015, 14, 32); => throws DateTimeException
System.out.println(date);
System.out.println(time);
System.out.println(dateTime);
System.out.println(date.getDayOfWeek());

LocalDate today = LocalDate.now();
LocalDate nextDate = date.withYear(today.getYear());
System.out.println(nextDate);
LocalDate modifiedNextDate = nextDate.minusWeeks(2060);
System.out.println(modifiedNextDate);
if(modifiedNextDate.isAfter(date)) {
    System.out.println("Not enough...");
}
else {
    System.out.println("Too much!!!");
}
Period p = Period.between(modifiedNextDate, date);
System.out.println(p);
System.out.println(nextDate.plusDays(7).minusMonths(2).plusYears(3));
System.out.println(date.plus(p));
```

```
1980-12-24
23:47:16.938
1980-12-24T23:47:16.938
WEDNESDAY
2017-12-24
1978-07-02
Too much!!!
P2Y5M22D
2020-10-31
1983-06-15
```

Parsing and Formatting

- The java.time.format.DateTimeFormatter class provides numerous predefined formatters, or you can define your own
- The parse and the format methods throw an exception if a problem occurs during the conversion process. Therefore, your parse code should catch the DateTimeParseException and your format code should catch the DateTimeException
- The format(DateTimeFormatter) method converts a temporal-based object to a string representation using the specified format
- The one-argument parse(CharSequence) method uses the ISO_LOCAL_DATE formatter. To specify a different formatter, you can use the two-argument parse(CharSequence, DateTimeFormatter) method

Parsing examples

```
String in = "20170125";
LocalDate date = LocalDate.parse(in, DateTimeFormatter.BASIC_ISO_DATE);

System.out.println(date);

String input = "04 14 2016";
try {
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("MM d yyyy");
    LocalDate newDate = LocalDate.parse(input, formatter);
    System.out.println(newDate);
}
catch (DateTimeParseException exc) {
    System.out.println(input + " is not parsable!");
}
```

```
2017-01-25
2016-04-14
```

Formatting examples

```
LocalDateTime now = LocalDateTime.now();

try {
    DateTimeFormatter format = DateTimeFormatter.ofPattern("MMM d yyyy  hh:mm a");
    System.out.println("NOW(1): " + now.format(format));
    System.out.println("NOW(2): " + now.format(DateTimeFormatter.BASIC_ISO_DATE));
    System.out.println("NOW(3): " + now.format(DateTimeFormatter.ISO_DATE));
    System.out.println("NOW(4): " + now.format(DateTimeFormatter.ISO_ORDINAL_DATE));
    System.out.println("NOW(5): " + now.format(DateTimeFormatter.ISO_LOCAL_TIME));
}
catch (DateTimeException exc) {
    System.out.println(now + " can't be formatted!");
}
```

```
NOW(1): Feb 20 2017  12:22 AM
NOW(2): 20170220
NOW(3): 2017-02-20
NOW(4): 2017-051
NOW(5): 00:22:03.221
```

Error Handling

- *Exceptions*
- *Error handling*

Reporting Exceptions

TestArray class:

```
public class TestArray {
   int[] intArray;
   public TestArray (int size) {
      intArray = new int[size];
   }
   public void addElement(int index, int value) {
     intArray[index] = value;
   }
}
```

Calling code in `main( )`:

```
TestArray myTestArray = new TestArray(5);
myTestArray.addElement(5, 23);
```

Does this code work?

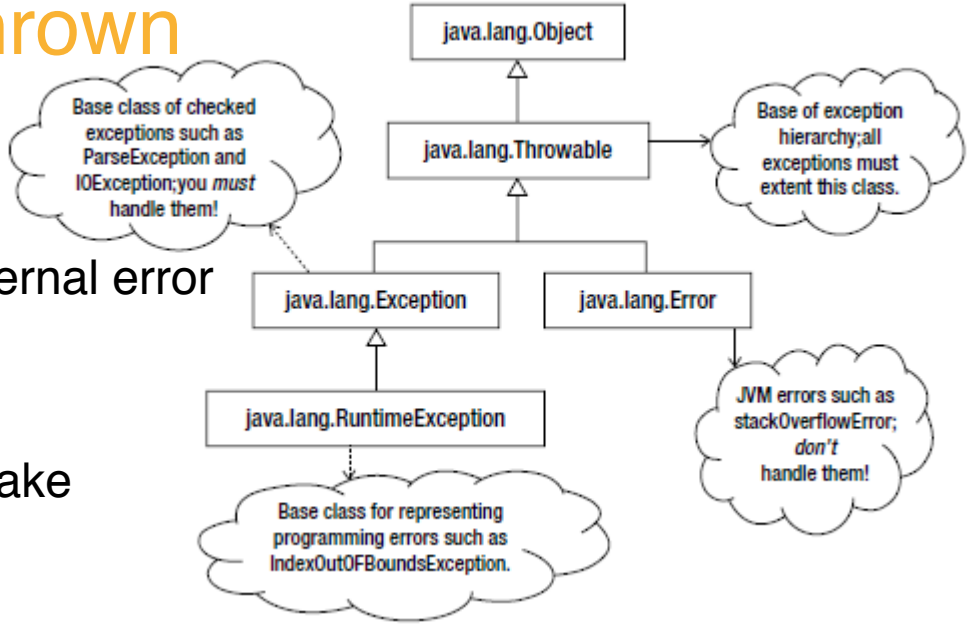
How Exceptions Are Thrown

- Normal program execution:
  - Caller method calls worker method
  - Worker method does work
  - Worker method completes work, and then execution returns to caller method.
- When an exception occurs, this sequence changes:
  - Exception is thrown and either:
    - A special Exception object is passed to a special method-like catch block in the current method

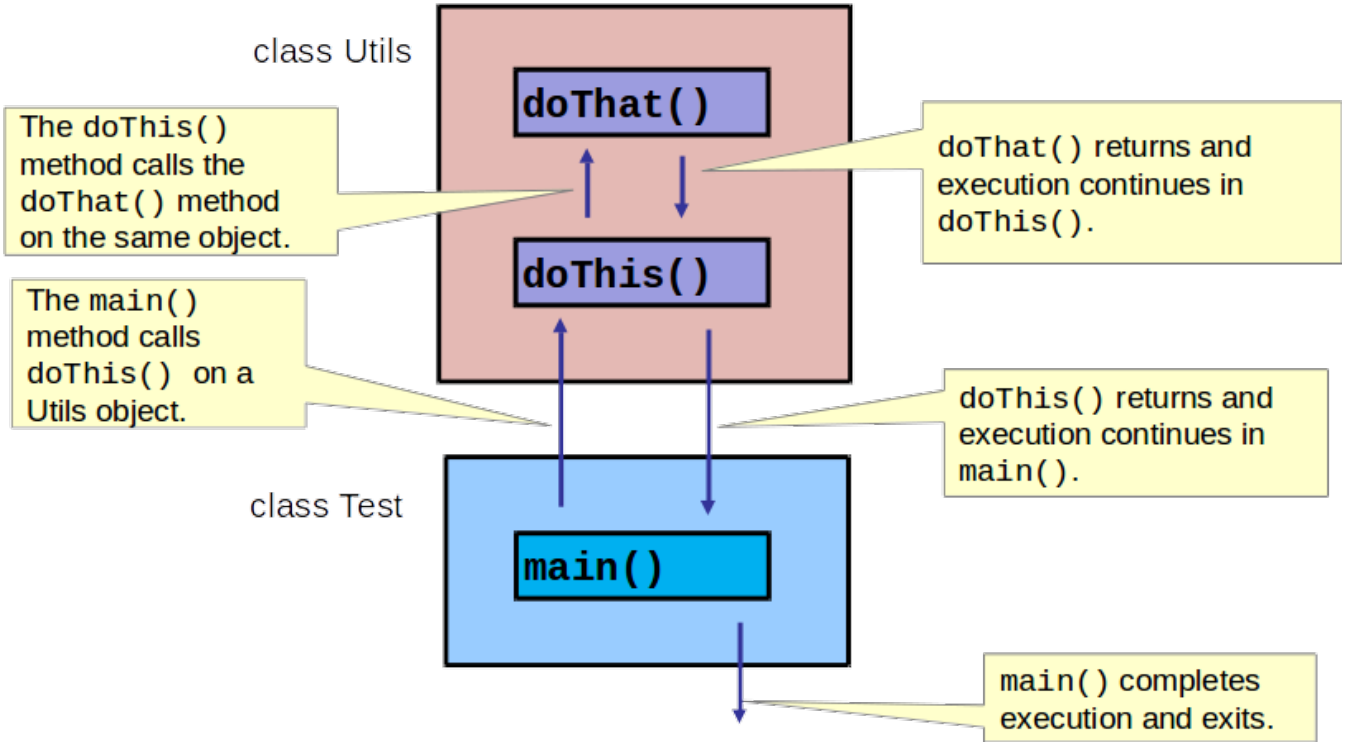
      OR
    - A special Exception object is passed to the caller method

How Exceptions Are Thrown

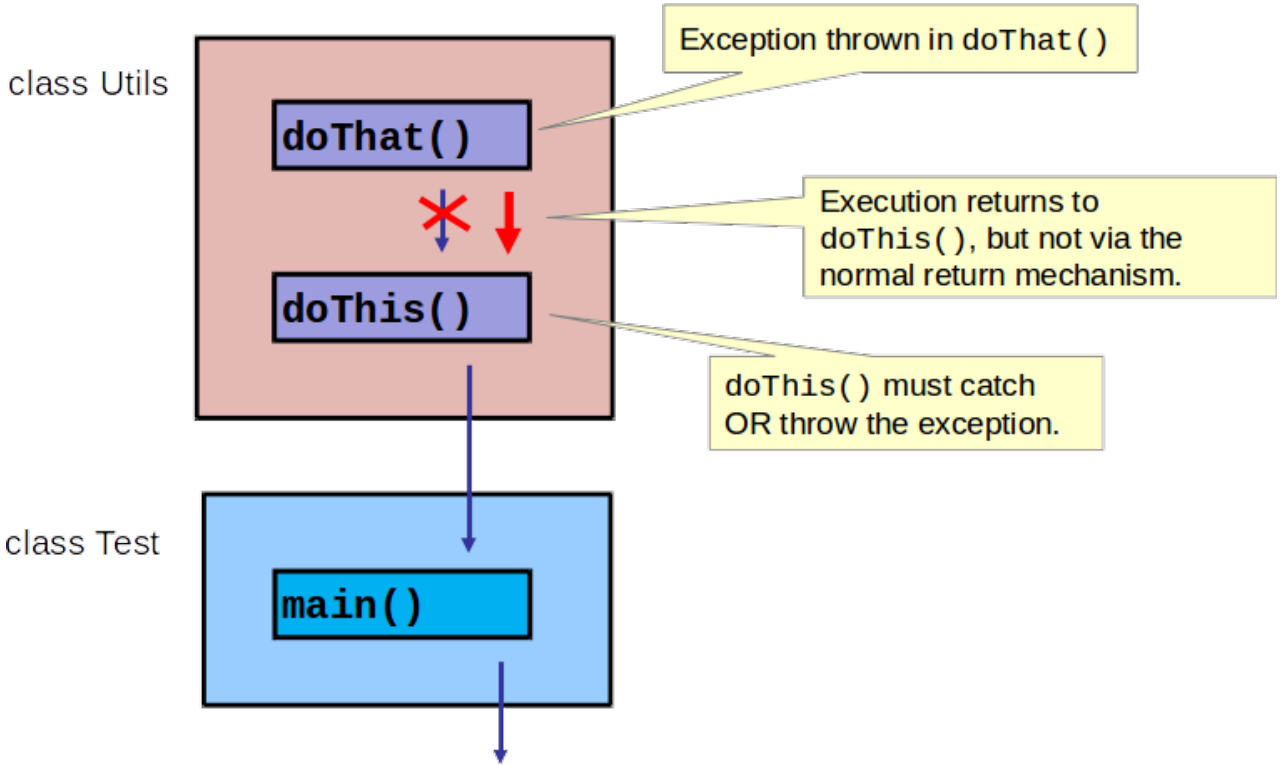
- Three main types of Throwable:
  - Error
    - Typically unrecoverable external error
    - Unchecked
  - RuntimeException
    - Typically programming mistake
    - Unchecked
  - Exception
    - Recoverable error
    - Checked (must be caught or thrown)



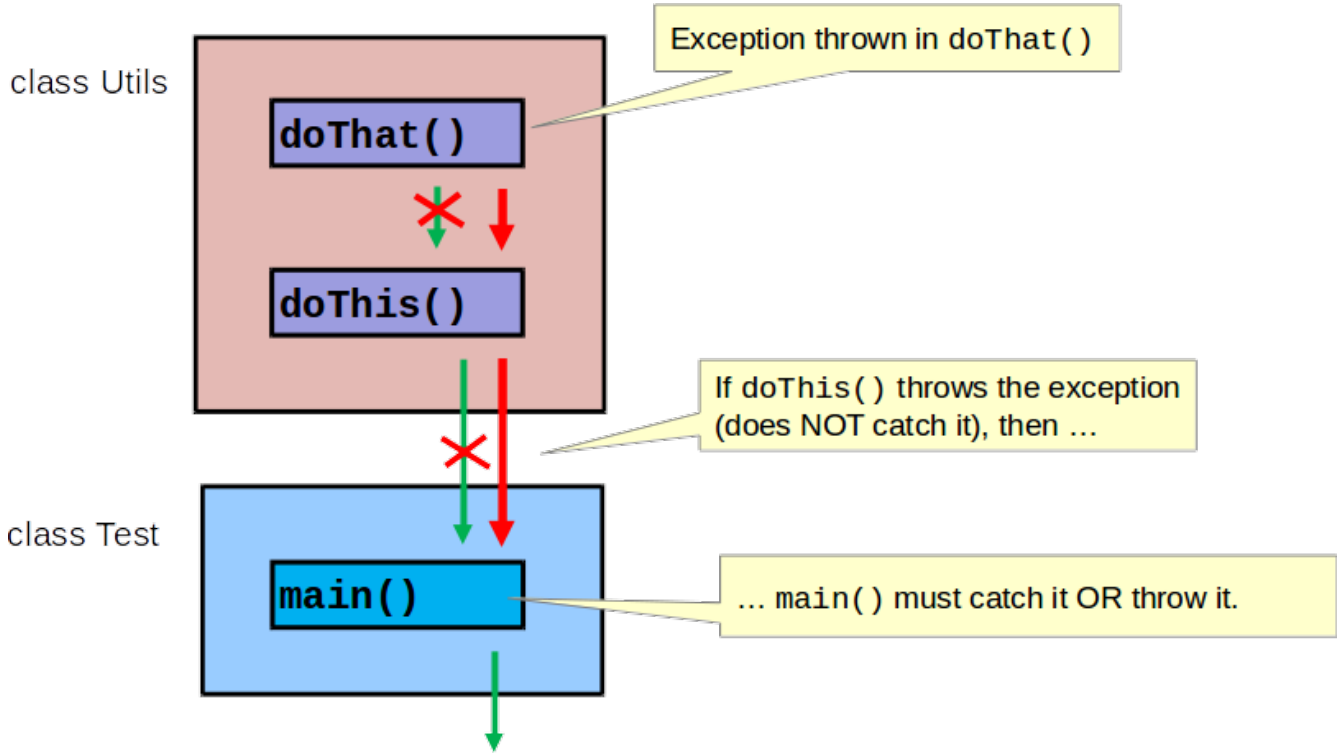
Method Stack



Throwing Throwables



Throwing Throwables



Working with Exceptions

```
10  public class Utils {
11
12      public void doThis() {
13
14          System.out.println("Arrived in doThis()");
15          doThat();
16          System.out.println("Back in doThis()");
17
18      }
19
20      public void doThat() {
21          System.out.println("In doThat()");
22      }
23  }
24
```

No exceptions thrown; nothing needs be done to deal with them.

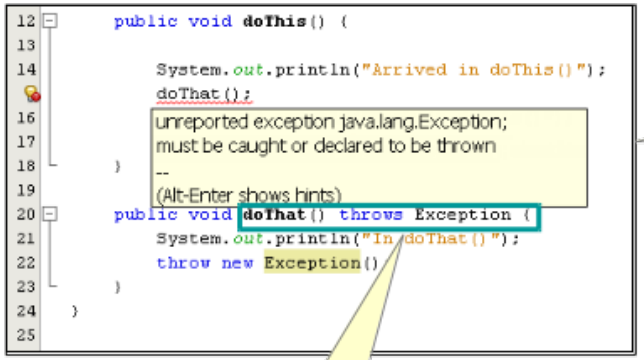
IDE uses a tooltip to give you your two options.

```
12      public void doThis() {
13
14          System.out.println("Arrived in doThis()");
15          doThat();
16          System.out.println("Back in doThis()");
17
18      }
19
20      public void doThat() {
21          System.out.println("
22          throw new Exception();
23      }
24  }
25
```

unreported exception java.lang.Exception;
must be caught or declared to be thrown
--
(Alt-Enter shows hints)

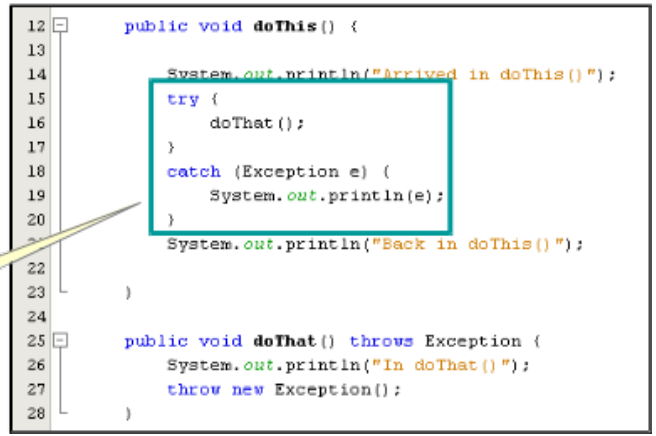
Throwing an exception within the method requires further steps.

Catching an Exception



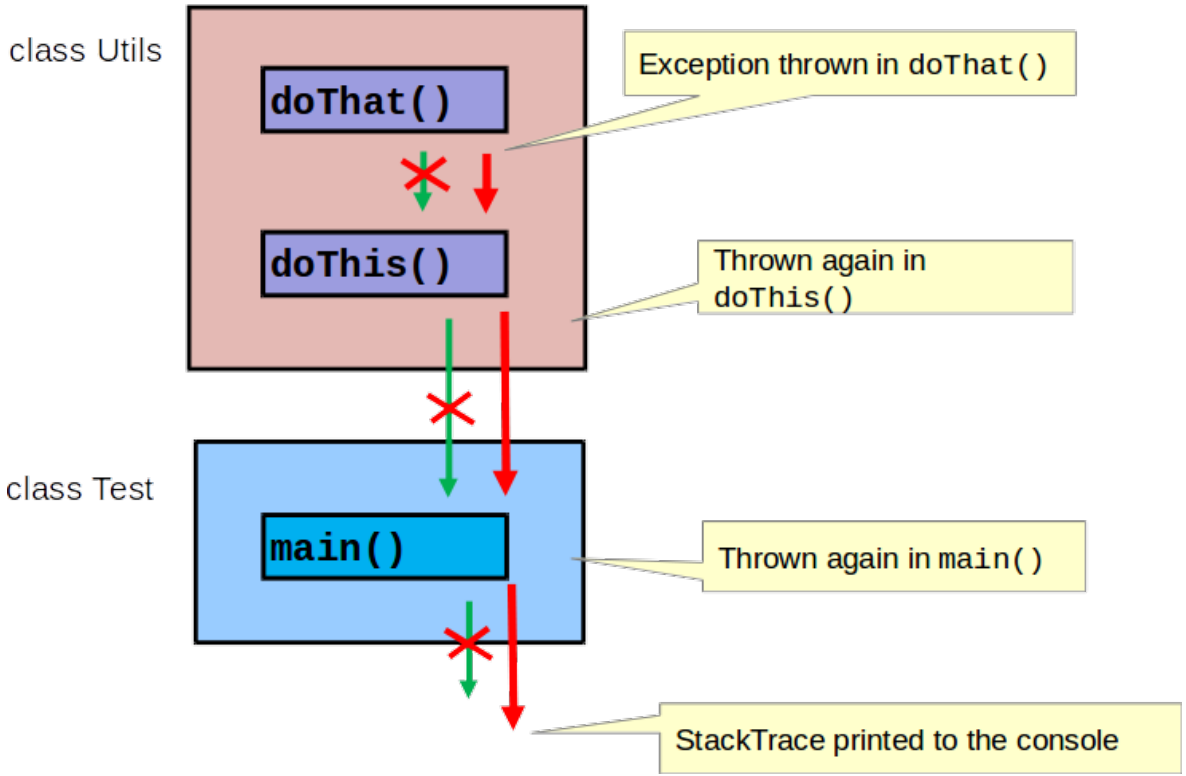
Now exception needs to be dealt with in `doThis()`.

`doThat()` now throws an exception.



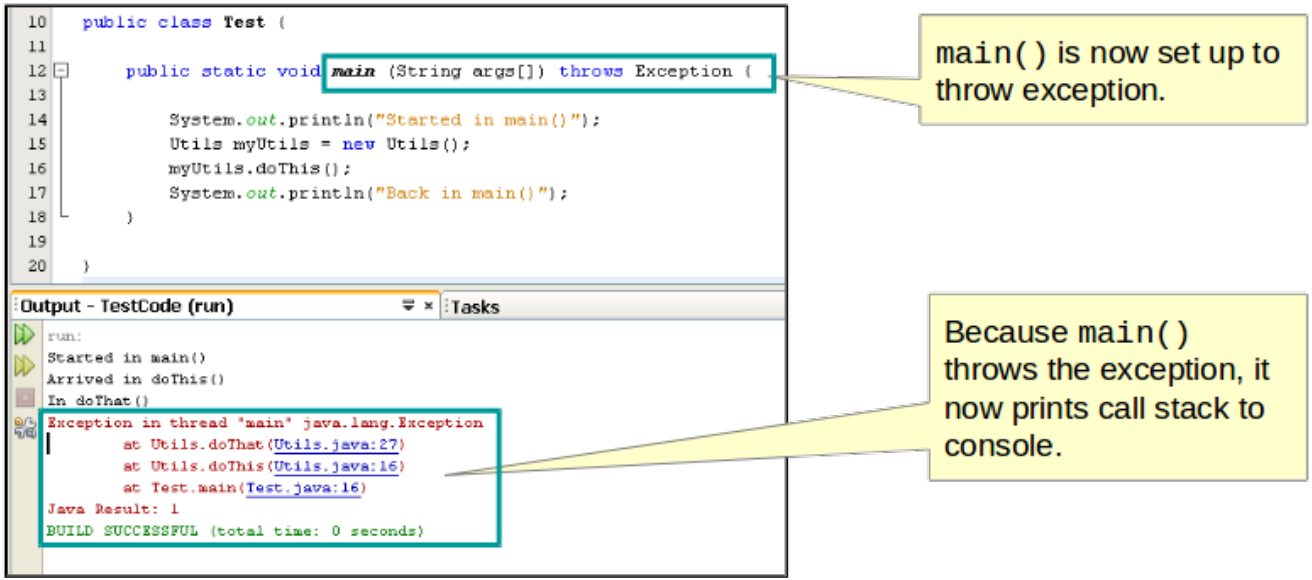
The try/catch block catches exception and handles it.

Catching an Exception



Exception Printed to Console

- Example of main() throwing exception



Summary of Exception Types

- A Throwable is a special type of Java object:
  - Only object type that is used as the argument in a catch clause
  - Only object type that can be “thrown” to the calling method
- Has two subclasses:
  - Error
    - Automatically thrown to the calling method if created
  - Exception
    - Must be explicitly thrown to the calling method
      OR
    - Caught using a try/catch block
    - Has a subclass RuntimeException that is automatically thrown to the calling method

Calling a method that throws an Exception

```
31
32      public static void testCheckedException()  {
33
34          File testFile = new File("//testFile.txt");
35
36          System.out.println("File exists: " + testFile.exists());
37          testFile.delete();
38          System.out.println("File exists: " + testFile.exists());
39
40      }
```

Constructor causes no compilation problems.

```
31
32      public static void testCheck
33
34          File testFile = new File
            testFile.createNewFile();
36
37          System.out.println("File exists: " + testFile.exists());
38          testFile.delete();
39          System.out.println("File exists: " + testFile.exists());
40
41      }
```

unreported exception java.io.IOException;
must be caught or declared to be thrown
--
(Alt-Enter shows hints)

`createNewFile()` can throw a checked exception, so testCheckedException method must throw or catch this exception.

Working with a checked Exception

Catching IOException:

```
public static void main(String args[]) {
   try {
      testCheckedException();
   }
   catch (IOException e) {
      System.out.println(e);
   }
 }

public static void testCheckedException() throws IOException {
   File testFile = new File("//testFile.txt");
   testFile.createNewFile();
   System.out.println("File exists: " + testFile.exists());

}
```

Best practices

- Catch the actual exception thrown, not the exception or Throwable superclass

- Examine the exception to find out the exact problem so you can recover cleanly

- You do not need to catch every exception:
  - A programming mistake should not be handled: it must be fixed
  - Ask yourself, “Does this exception represents behavior I want the program to recover from?"

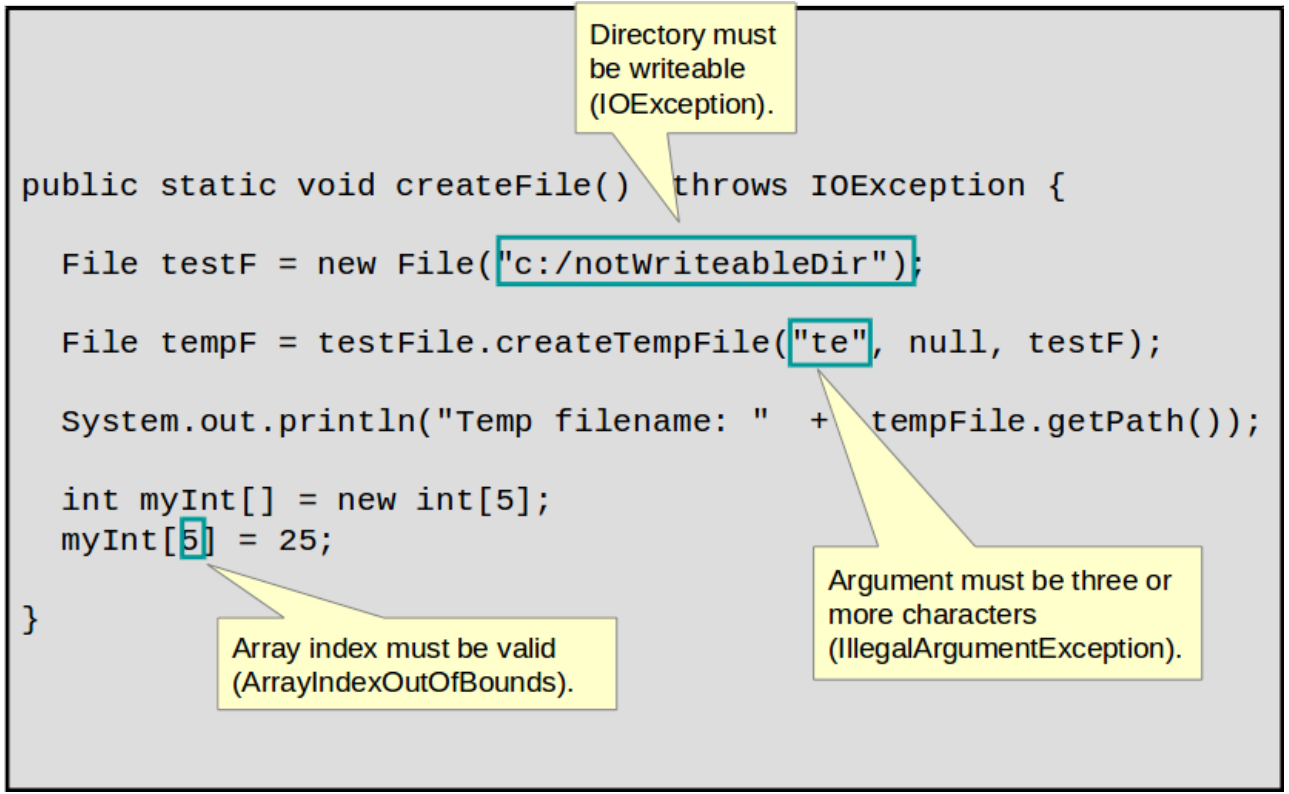
Bad practices

```
public static void main (String args[]) {
  try {
     createFile("c:/testFile.txt");
  }
  catch (Exception e) {

     System.out.println("Problem creating the file!");
  }
}

public static void createFile(String fileName) throws
   IOException {
     File f = new File(fileName);
     f.createNewFile();

     int[] intArray = new int[5];
     intArray[5] = 27;
  }
```

Catching superclass?

No processing of the Exception object?

Multiple Exceptions



Catching Remaining Exceptions

```
public static void main (String args[]) {
   try {
      createFile();
   }
   catch (IOException ioe) {
      System.out.println(ioe);
   }
   catch (IllegalArgumentException iae) {
      System.out.println(iae);
   }
   catch (Exception e) {
      System.out.println(e);
   }
}

public static void createFile()  throws IOException {
  File testF = new File("c:/writeableDir");
  File tempF = testFile.createTempFile("te", null, testF);
  System.out.println("Temp filename is " + tempFile.getPath());
  int myInt[] = new int[5];
  myInt[5] = 25;
}
```

Multiple Exceptions (since Java 7)

```
public static void main (String args[]) {
   try {
      createFile();
   }
   catch (IOException | IllegalArgumentException e) {
      System.out.println(e);
   }
   catch (Exception e) {
      System.out.println(e);
   }
}

public static void createFile()  throws IOException {
  File testFile = new File("c:/writeableDir");
  File tempFile = File.createTempFile("te", null, testFile);
  System.out.println("Temp filename is " + tempFile.getPath());
  int myInt[] = new int[5];
  myInt[5] = 25;
}
```

Try-with-Resources (since Java 7)

```
Charset charset = StandardCharsets.US_ASCII;
Path outputFilePath = Paths.get("fileName.txt");

// Open zip file and create output file with
// try-with-resources statement

try (ZipFile zf = new ZipFile("zipFileName.zip");
     BufferedWriter writer = Files.newBufferedWriter(outputFilePath, charset)) {
    for (Enumeration entries = zf.entries(); entries.hasMoreElements();) {
        // Get the entry name and write it to the output file
        String newLine = System.getProperty("line.separator");
        String zipEntryName =
                ((ZipEntry)entries.nextElement()).getName() + newLine;
        writer.write(zipEntryName, 0, zipEntryName.length());
    }
}
catch (IOException e) {
    e.printStackTrace();
    for (Throwable suppressedExceptions : e.getSuppressed()) {
        suppressedExceptions.printStackTrace();
    }
}
```

Declarations that implement interface AutoCloseable

This method of Throwable keeps the possible exceptions that might occured during the execution of the compiler generated code (close())

Close method is invoked automatically either:

- At the end of try (if no exception occurs)
- At the beginning of each catch if an exception occurs
- Simulates the call of close() of the
- ressources at finally (it is not the same thing)

Luís Ribeiro

I'm a software engineer with more than 14 years of experience in software development in Java and other technologies, software architecture, team management and project management.

My mission is to transform people's ideas into fully functional, production ready and user friendly software applications that can change the world!

luismmribeiro@gmail.com

Thank you