Object Oriented Development with Java

(CT038-3-2 and Version VC1)

Managing Error and Exceptions

Java Exception

Topic & Structure of the lesson

- Exception Handler
- Exception Class
- Handling Exception
  - Handling multiple exceptions
  - `finally` clause
  - Checked and unchecked exceptions
  - `throw` exception
- Creating (your own) exception class

Learning outcomes

- At the end of this lecture you should be able to:
  - Handle exceptions
  - Understand the difference between checked and unchecked exceptions
  - Use try-catch-finally block

Key terms you must be able to use

If you have mastered this topic, you should be able to use the following terms correctly in your assessments:

- Try
- Catch
- Throw
- Finally

Exception

- An exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions
- Exceptions are said to have been “thrown”
- It is the programmers responsibility to write code that detects and handles exceptions
- Unhandled exceptions will crash a program

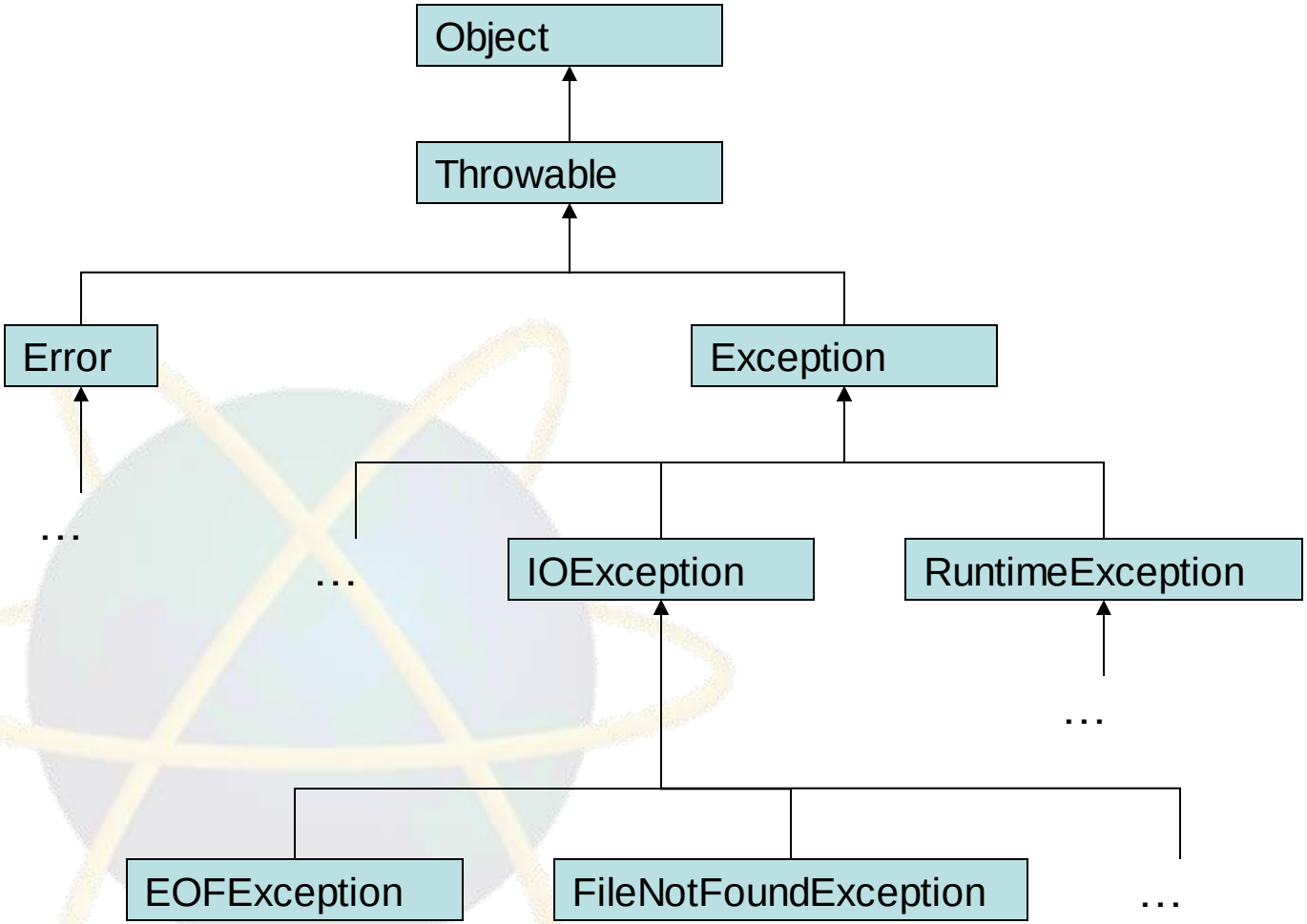
Exception Handler

- Java allows you to create exception handlers
- An *exception handler* is a section of code that gracefully responds to exceptions
- The process of intercepting and responding to exceptions is called *exception handling*

Exception Class

- Exception objects are created from classes in the Java API hierarchy of exception classes
- All of the exception classes in the hierarchy are derived from the `Throwable` class
- `Error` and `Exception` are derived from the `Throwable` class

Exception Class



Exception Class

- Classes that are derived from `Error`:
  - are for exceptions that are thrown when critical errors occur. (i.e.)
    - an internal error in the Java Virtual Machine, or
    - running out of memory.
- Applications should not try to handle these errors because they are the result of a serious condition
- Programmers should handle the exceptions that are instances of classes

Handling Exception

- To handle an exception, you use a *try* statement.

```
try
{
  //try block statements...
}
catch (ExceptionType ParameterName)
{
  //catch block statements...
}
```

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

Handling Exception

- A *try block* is:
  - one or more statements that are executed, and
  - can potentially throw an exception
- The application will not halt if the try block throws an exception
- After the try block, a `catch` clause appears

  `catch (`*`ExceptionType ParameterName`*`)`

  `ExceptionType` - name of an exception class and *`ParameterName`* - variable name that references the exception object if the code in the try block throws an exception

Example

- This code is designed to handle a `FileNotFoundException` if it is thrown

```
try
{
   File file = new File
  ("MyFile.txt");
   Scanner inputFile = new
  Scanner(file);
}
catch (FileNotFoundException e)
{
   System.out.println("File not
  found.");
}
```

Polymorphic References to Exceptions

- When handling exceptions, you can use a polymorphic reference as a parameter in the `catch` clause
- Most exceptions are derived from the `Exception` class
- A `catch` clause that uses a parameter variable of the `Exception` type is capable of catching any exception that is derived from the `Exception` class

https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html

Example

```
try
{
    number = Integer.parseInt(str);
}
catch (Exception e)
{
    System.out.println("The following
  error occurred: " + e.getMessage());
}
```

- The `Integer` class's `parseInt` method throws a `NumberFormatException` object
- The `NumberFormatException` class is derived from the `Exception` class

Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception
- A `catch` clause needs to be written for each type of exception that could potentially be thrown
- The JVM will run the first compatible `catch` clause found
- The `catch` clauses must be listed from most specific to most general

Handling Multiple Exceptions

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println(str + "is not a
  number.");
}
catch (IllegalArgumentException e)
{
   System.out.println("Bad number
  format.");
}
```

Handling Multiple Exceptions

```
try
{
   number = Integer.parseInt(str);
}
catch (NumberFormatException e)
{
   System.out.println("Bad number
  format.");
}
catch (NumberFormatException e) //
  ERROR!!!
{
  System.out.println(str + " is not a
```

Duplicated catch with same parameter type

Handling Multiple Exceptions

```
try {
   number = Integer.parseInt(str);
}
catch (IllegalArgumentException e)
{
   System.out.println("Bad number
  format.");
}
catch (NumberFormatException e) //
  ERROR!!!
{
   System.out.println(str + " is not a
  number.");
}
```

The `NumberFormatException` class is derived from the `IllegalArgumentException` class.

Handling Multiple Exceptions

- You should catch from
  - specific type to general type
- In Java SE 7 and later, a single catch block can handle more than one type of exception
- This feature can reduce code duplication and lessen the temptation to catch an overly broad exception

```
catch (IOException|SQLException ex) {
        logger.log(ex);
        throw ex;
}
```

Implicitly final

`finally` clause

- The try statement may have an optional `finally` clause
- If present, the `finally` clause must appear after all of the `catch` clauses

```
try {
   //try block statements...
}
catch (ExceptionType ParameterName)
  {
   //catch block statements...
}
finally {
   //finally block statements...
```

`finally` clause

- always executed after the try block has executed and
- after any catch blocks have executed if an exception was thrown

```
try {
   System.out.println(a[3]); //assume that array
                                size is 2
}
catch(ArrayIndexOutOfBoundsException e) {
   System.out.println("Exception thrown: " + e);
}
finally {
   System.out.println("First element: " + a[0]);
}
```

Checked and Unchecked Exception

- There are two categories of exceptions:
  - unchecked
  - checked

Checked Exception

- All exceptions that are *not* derived from `Error` or `RuntimeException` are *checked exceptions*
- Occurs during compile time
- Invalid conditions that are not within the control of the program such as invalid user input, database problems, network outages, absent files
- Eg. IOException, SQLException, ClassNotFoundException

Unchecked Exception

- *Unchecked exceptions* are program bugs (errors in program logic) at time of execution (runtime)
- derived from the `RuntimeException` class
  - `RuntimeException` serves as a superclass for exceptions that result from programming errors
  - Subclasses: IllegalArgumentException, NullPointerException, ArithmeticException or ArrayIndexOutOfBoundsException

`throws` exception

- If the code in a method can throw a checked exception, the method:
  - must handle the exception, or
  - it must have a `throws` clause listed in the method header
- The `throws` clause informs the compiler what exceptions can be thrown from a method

Example

Compiled!

```
// This method will not compile!
public void displayFile(String name)
            throws FileNotFoundException

{
   // Open the file.
   File file = new File(name);
   Scanner inputFile = new Scanner(file);
   // Read and display the file's contents.
   while (inputFile.hasNext())
   {
      System.out.println(inputFile.nextLine());
   }
   // Close the file.
   inputFile.close();
}
```

`throw` exception

- You can write code that:
  - throws one of the standard Java exceptions, or
  - an instance of a custom exception class that you have designed
- The `throw` statement is used to manually throw an exception

```
throw new
    ExceptionType(MessageString);
```

Creating Exception Class

- You can create your own exception classes by deriving them from the `Exception` class or one of its derived classes

Example

```
public class NegativeStartingBalance
                                extends Exception {
   public NegativeStartingBalance() {
      super("Error: Negative starting balance");
   }

   public NegativeStartingBalance(double amount) {
      super("Error: Negative starting balance: " +
            amount);
   }
}
```

Scenario for Customised Exception

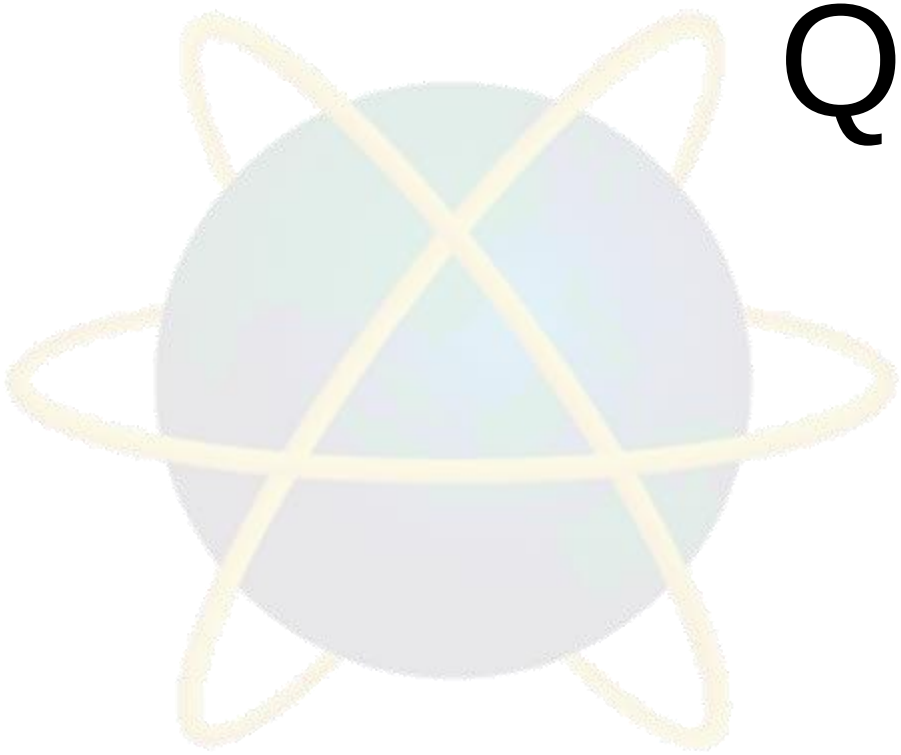
- Some examples of exceptions that can affect a bank account:
  - A negative starting balance is passed to the constructor
  - A negative interest rate is passed to the constructor
  - A negative number is passed to the deposit method
  - A negative number is passed to the withdraw method
  - The amount passed to the withdraw method exceeds the account's balance
- We can create exceptions that represent each of these error conditions

Quick Review Questions

- What is exception handling?
- How to handle exception?
- What is the different between `throws` and `throw`?
- What is checked exception?
- What is unchecked exception?
- Can I customise a specific exception type in my problem domain?

Summary of Main Teaching Points

- Different types of Exception Handler
- Exception Class
- Handling Exception
  - try
  - `finally` clause
  - catch
  - `throw` exception
- Creating (your own) exception class



Q & A

Next Session

- AWT vs Swing
- Swing components
- Creating a window
- Adding components
- Handling action events