Exception Handling

ACS-1904 LECTURE 8

Exception Handling

Exception:

- An event that occurs during program execution that disrupts the normal flow.
- When an exception occurs an exception object is created
- The object contains information about the situation and the state of the program at that time.
- The creation of an exception object, and its handoff to the Java runtime system, is called throwing an exception.

Unchecked exceptions

Two kinds of exceptions

- Checked
- Unchecked

Unchecked exceptions

Unchecked exceptions

- generally considered to be something that should never occur,

  → if one does occur the program should terminate.

  - Indicates a logic error in the program
  - e.g. `ArrayIndexOutOfBounds`

Checked exceptions

Checked exceptions

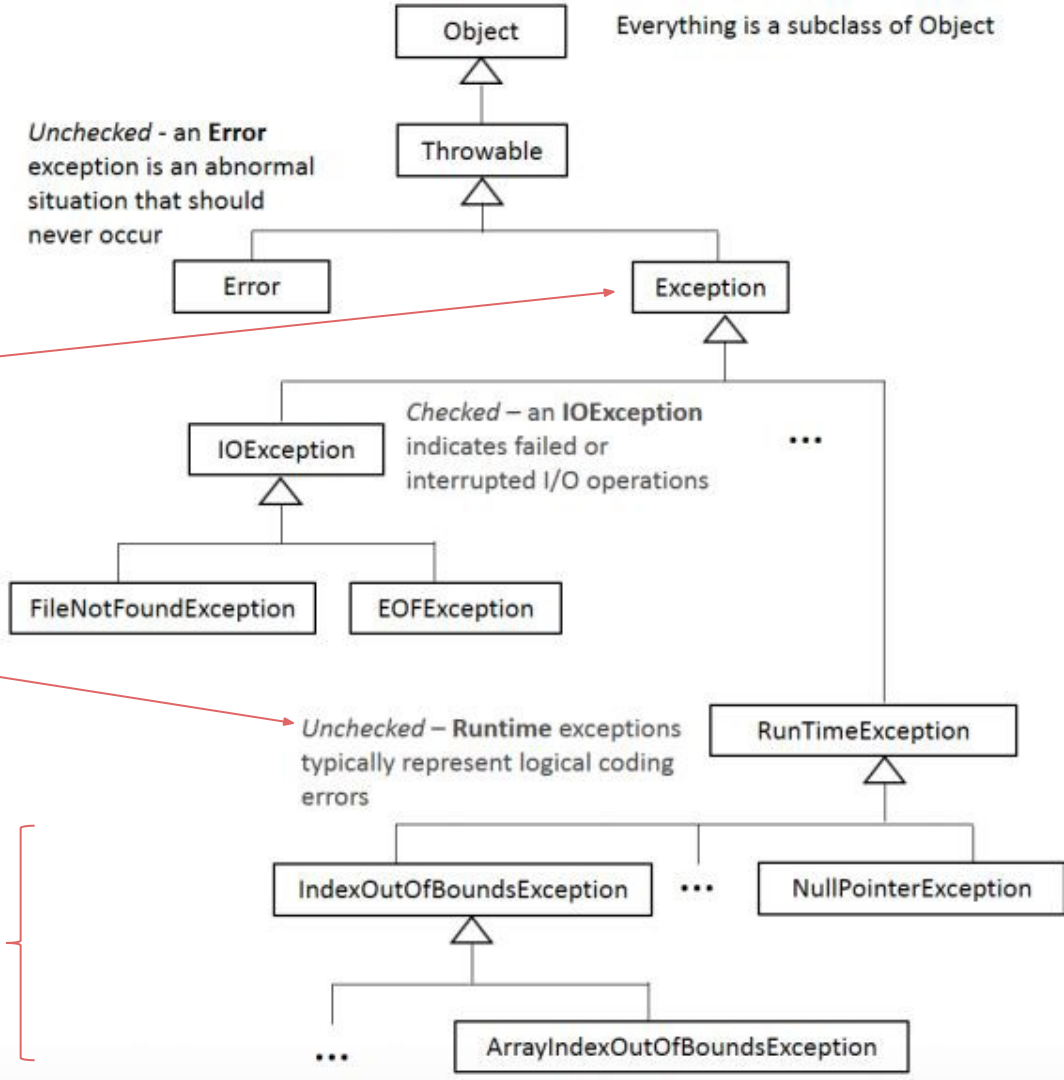
- represent situations from which a program can be expected to recover
- E.g. `EOFException`
  - End of file exception
  - Normal situation if reading a file with an unknown number of values
- E.g. `FileNotFoundException`
  - File not found exception
  - Easily happens
    - if filename is provided by user we can prompt for another name, or end gracefully

Exception subclasses

The class Exception and any subclasses that are not also subclasses of RuntimeException are checked exceptions.

Unchecked exceptions

A program should fail if any of these occur … such an exception indicates a logic error



Handling an exception

If a checked exception could occur in a program then the program must either:

include a `throws` clause for it

or

include a `try` that catches the exception.

Handling an exception: try statement

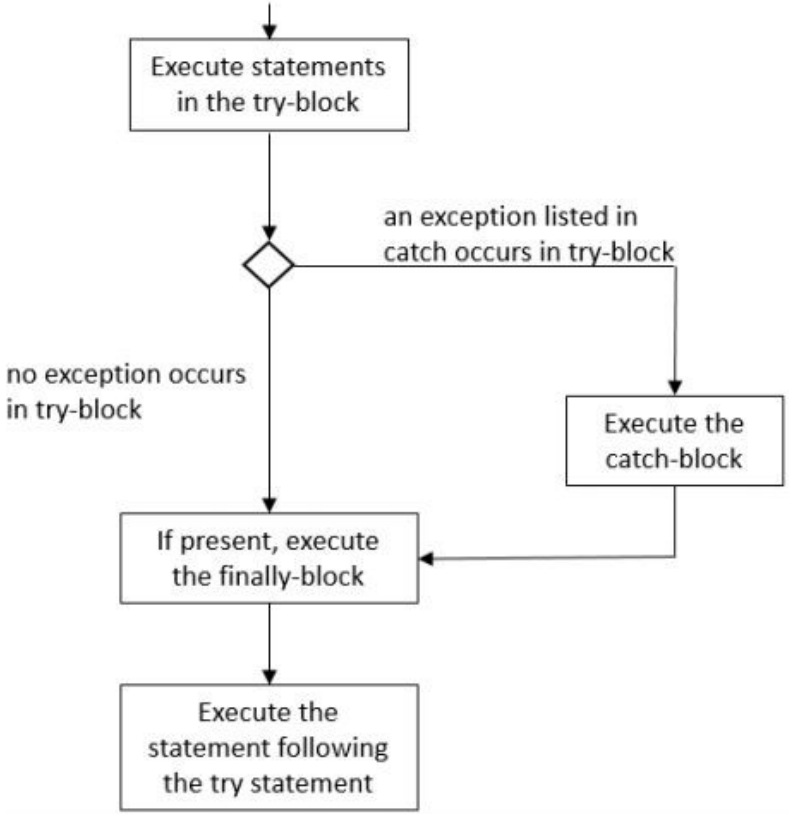
```
try{
   some Java statements
}

catch (one or more exceptions){
   some Java statements
}

finally {
   some Java statements
}
```

Handling an exception: try statement

Execution of a try statement



No try/catch for FileNotFoundException

ReadBinary.java

Recall from last class:

```
public class ReadBinary {
    public static void main(String[] args) throws IOException{
        …
        DataInputStream  is  = new DataInputStream(
            new FileInputStream("myData.ser"));

        …
```

*If an `IOException` occurs the program terminates abnormally: the exception is **thrown***

`IOException` includes `FileNotFoundException`
An `IOException` occurs if the file does not exist.
The program is terminated immediately and a call trace appears.

Handling an exception: FileNotFoundException HandleFileNotFound.java

```
public class HandleFileNotFound {
    …
        try {
            is = new DataInputStream(new FileInputStream(fileName));
            JOptionPane.showMessageDialog(null, "Success, file found");
        }
        catch (FileNotFoundException e){
            JOptionPane.showMessageDialog(null, "Error, file not found");
        }
        finally{
            JOptionPane.showMessageDialog(null, "finally clause
                executing"); }
    }
}
```

*If a `FileNotFoundException` occurs in the try-block the exception is **caught** and handled in the catch-block*

Handling an exception

- `ReadBinary.java` from chapter 7 reads exactly 5 integers
- Want to be able to read an unknown number of integers.
- If `ReadBinary` attempts to execute `readInt()` and there are no more integers to read then an `EOFException` occurs.
- Note: Java documentation states `readInt()` can raise another exception, `IOException`

Handling exceptions: example

ReadFromBinaryUntilEndOfFile.java

Main method calls

- `getFile` - uses try/catch to detect file-not-found
- `getTotal` – uses try/catch to detect end-of-file
- `closeFile` – uses try/catch to close the file

```
public static void main (String[] args){
    DataInputStream is = getFile();
    int total = 0;
    if (is!=null){
        total = getTotal(is);
        closeFile(is);
    }
    JOptionPane.showMessageDialog(null, "total = "+total);
}
```

Main method calls other methods

Handle file-not-found - `getFile`

ReadFromBinaryUntilEndOfFile.java

try/catch for file-not-found

```
public static DataInputStream getFile(){
    String fileName = JOptionPane.showInputDialog("Enter file name:");
    DataInputStream is = null;
    try {
        is = new DataInputStream(new FileInputStream(fileName));
            JOptionPane.showMessageDialog(null, "Success, file found");
    }
    catch (FileNotFoundException e){
        JOptionPane.showMessageDialog(null, "Error, file not found");
        is = null;
    }
    return is;
}
```

Handle end-of-file - `getTotal`

ReadFromBinaryUntilEndOfFile.java

Read integers until end-of-file

```
public static int getTotal(DataInputStream is){
    …
    // read integers until there are none left
    boolean endOfFile = false;
    …
    while (! endOfFile){
        try{
            i = is.readInt();
            total += i;
        }
        catch (EOFException e){
            endOfFile = true;
        }
        catch (IOException e){
            JOptionPane.showMessageDialog(null,
                "Error, an IOExceptionoccurred reading the file");
            endOfFile = true;
        }
    }
    return total;
```

Read next integer

*If end-of-file occurs this catch-block executes. Setting **endOfFile** true will force while loop to end*

If an IOException occurs this catch-block executes

Handle close-file - `closeFile`

ReadFromBinaryUntilEndOfFile.java

```
public static void closeFile (DataInputStream is){
        // close may throw an IOException
        try{
            is.close();
            JOptionPane.showMessageDialog(null,
              "program terminated normally");
        }
        catch (IOException e){
            JOptionPane.showMessageDialog(null,
              "Error, an IOException occurred closing the file");
        }
    }
```

Close the file

Problems could arise when closing the file

Handling exceptions

We can catch and handle exceptions

We do not have to abnormally terminate a program.

Custom exceptions

You can define your own custom exception, however

- in general, not recommended
- before developing a separate exception class the extensive set of subclasses of Throwable should be examined for one that fits.

Custom exceptions

A custom exception is one that extends either `Exception` or `RunTimeException`

- subclass `Exception` if you require a checked exception
- subclass `Runtime` if you require an unchecked exception

Example:

- prevent an employee from being instantiated if the SIN is invalid

Example - custom exception

InvalidSINException.java

```
public class InvalidSINException extends Exception{
    private String sin;
    /**
     * Constructor that accepts a SIN
     */
    public InvalidSINException(String s){
        sin = s;
    }
    /**
     * getter for SIN that caused the exception
     */
    public String getSin(){
        return sin;
    }
}
```

This is a checked exception

Fields

constructor

Getter for SIN field

Example - custom exception

Employee.java

```
public class Employee {
    private String sin;
    private String name;
    private String position;
    public Employee (String sin, String name, String position)
        throws InvalidSINException {
         if (! validate(sin)) throw new InvalidSINException(sin);
         this.sin = sin;
         this.name = name;
         this.position = position;
    }
    …
    public boolean validate(String sin){
```

The employee class

The constructor tests the SIN

If the sin is invalid then throw the exception

Details in code

TestNewException.java

Testing the custom exception

Create two employees

This one causes an exception

Catch a custom exception

```
public class TestNewException{
    public static void main(String[] args){
    try {
        // a valid SIN.
        System.out.println( "Attempting to create employee 046454286" );
        Employee e1 = new Employee( "046454286", "Joe Who", "instructor");
        System.out.println("Create OK: "+e1);
        // not valid SIN.
        // An exception will be thrown
        System.out.println( "\nAttempting to create employee 123456789" );
        Employee e2 = new Employee("123456789", "Steve Stephens", "instructor");
        System.out.println( "Create OK: " +e2);
    }
    catch (InvalidSINException e){
        System.out.println("Error creating employee. The provided SIN, "  +
                e.getSin()+" is invalid ");
        e.printStackTrace();
    }
```

Summary

- Two important categories of exception
  - Unchecked
    - Let these be thrown
    - Do not catch
    - These indicate logic errors
  - Checked exceptions
    - Programs can recover from these
- Multiple catch-blocks can be used to catch different problems
- Specific examples
  - File not found
  - Reading until end-of-file
- Custom exceptions can be defined and used
  - Extend Exception – for checked exceptions
  - Extend Runtime – for unchecked exceptions