

Applied Exception Handling

Exception handling techniques

- Understanding what exceptions are and how they are generated is very important in Java programming
- However, applying Java exceptions correctly is equally important

Exception handling techniques

- Applying exceptions in Java encompasses the following:
 - What are the options available when handling exceptions
 - What are the best practices for handling exceptions
 - What should be avoided!!!

To “handle” or “declare”?...that is the question

- When there is no possible foreseeable action that you can take as a programmer to deal with an exception, only then should you declare it (use the throws declaration)
- In other words, you should deal with exceptions by default unless there's a very clearly defined reason for declaring
- Given this, it is important to examine the options available to a programmer when handling exceptions

Options for handling

- As mentioned previously there are a variety of ways to handle exceptions, suppose that a piece of code threw an exception while connecting to a server, the options might be:
 - Wait and retry the connection in a while
 - Try to connect at a different port
 - Connect to an alternate server
 - Give the user the choice
 - Simply display an error message

Options for declaring

- Perhaps the biggest advantage to declaring exceptions is that it makes it possible to create extremely flexible code
- Handling exceptions usually involves some specific responses, these specific responses may reduce the flexibility of the code
- API's tend to supply methods that declare rather than handle exceptions in order to maximize the flexibility and reusability of the code

Common Exception Handling Options

- The following slides examine some the options available when handling exceptions, including:
 1. Logging
 2. Require User input/response
 3. Default and/or alternate values
 4. Ignore
 5. Retry the action
 6. Prepare for shutdown

1. Logging

- Logging can be used to record exceptions that occurred during the course of a program
- Logging can be simple output statements (System.out or System.err) or can take the form of external log files

Logging using default output

- Within a catch statement the programmer places *System.out.println("Describe response")* or *System.err.println("Describe error")*
- This approach can be adapted for file output by editing the default output device to an output file, however, some server side applications do not suit this approach (JVM is running on the server and access to the default output may not be available)

Logging using Custom Logger class

- The programmer can create a custom logger class
- This class provides static methods to log events and exceptions that occur during the course of the program

Logging using the Java Logging API

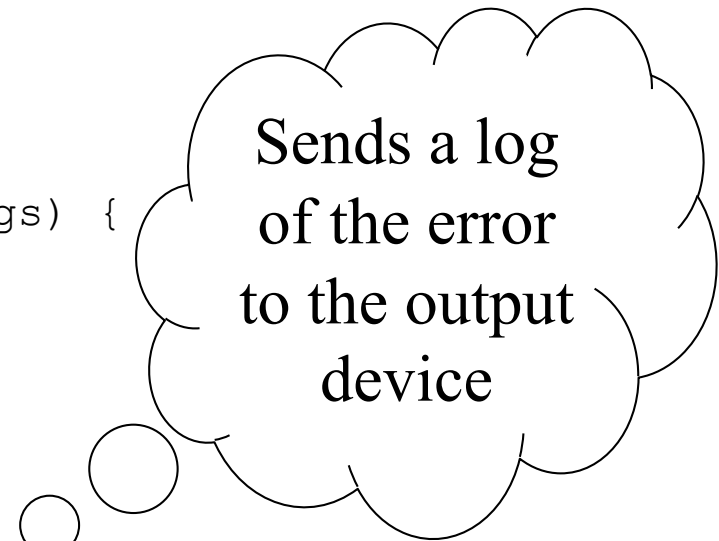
- Java (since Java 1.4) provides an API for logging in programs
- The log classes are found in the *java.util.logging* package
- Logs can be sent to the standard output or simply sent to external files

Java API logging to output

```
import java.util.logging.*;

public class LoggerOutputTest {

    public static void main(String[] args) {
        try {
            Object s = null;
            s.toString();
        }
        catch (NullPointerException ex) {
            Logger logException = Logger.getLogger(
                "basic.exception.example");
            logException.log(Level.SEVERE, "This is added to the
                log");
        }
    }
}
```



Sends a log
of the error
to the output
device


Java API for log file output

- The following slide shows an example of the logging of errors to an output file using using the Java logger API
- Running this program will cause a *NullPointerException* that will be logged in an external file
- Note the use of the nested *try--catch*

```
import java.io.IOException;
import java.util.logging.*;

public class LoggingFileOutputTest {

    public static void main(String[] args) {
        Logger logException = Logger.getLogger("exception.file.example");
        try {
            Object s = null;
            s.toString();
        }
        catch (NullPointerException ex) {
            try {
                Handler fileOut = new FileHandler("errorLog.txt", true);
                fileOut.setFormatter(new SimpleFormatter());
                logException.addHandler(fileOut);
            }
            catch (IOException ioEx) {
                //ignored
            }
        }
        logException.log(Level.SEVERE, "An exception occurred in here");
    }
}
```



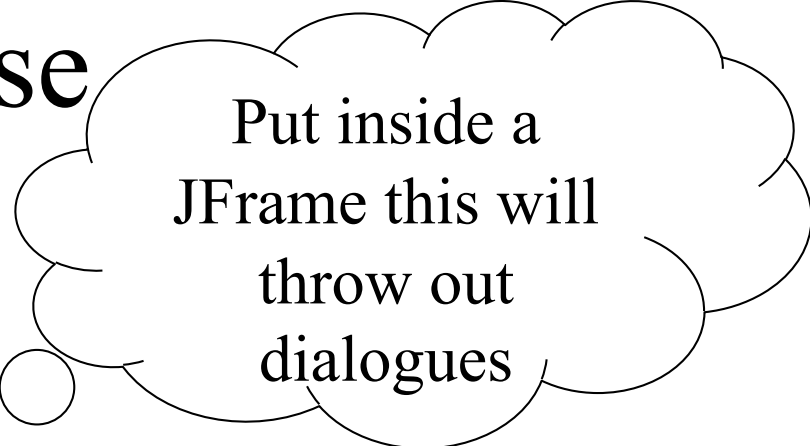
Log sent to
the file called
errorLog.txt

2. Require User Input/Response

- Some errors may be “close” to the user
- The user may be able to decide the course of action they require
- Often happens using a user interface (GUI)
- Can use pop-ups to warn or ask user for a response (JDialog or JOptionPane)

User Input/Response

```
try {
    Object s = null;
    s.toString();
}
catch (NullPointerException e) {
    String message = "There appears to be something set to null in this
        program, do you want to QUIT!!!";
    String title = "System Error";
    int result = JOptionPane.showConfirmDialog(this, message, title,
        JOptionPane.ERROR_MESSAGE);
    if (result == JOptionPane.YES_OPTION) {
        System.exit(0);
    }
    else {
        String warnMessage = "YOU ARE CHOOSING TO CONTINUE WITH AN ERROR
            CONDITION";
        String warnTitle = "System Warning";
        JOptionPane.showMessageDialog(this, warnMessage, warnTitle,
            JOptionPane.WARNING_MESSAGE);
    }
}
```



Put inside a
JFrame this will
throw out
dialogues

3. Default and/or alternate value

- In order for this approach to exception handling to work there needs to be some possible default or alternate value
- For example, suppose a program is attempting to connect to a server at IP address 127.123.244.101 on Port “1234” and it fails
- It may be possible to try a different Port number in the catch statement of the exception!!

Default/Alternate values

- Alternate values can be set as static final variables in the program, e.g. :

```
public static final String ALT_PORT = "1000";
```

- Now the catch statement of the initial attempt to connect can contain a second attempt to connect this time to port 1000

4. Ignore

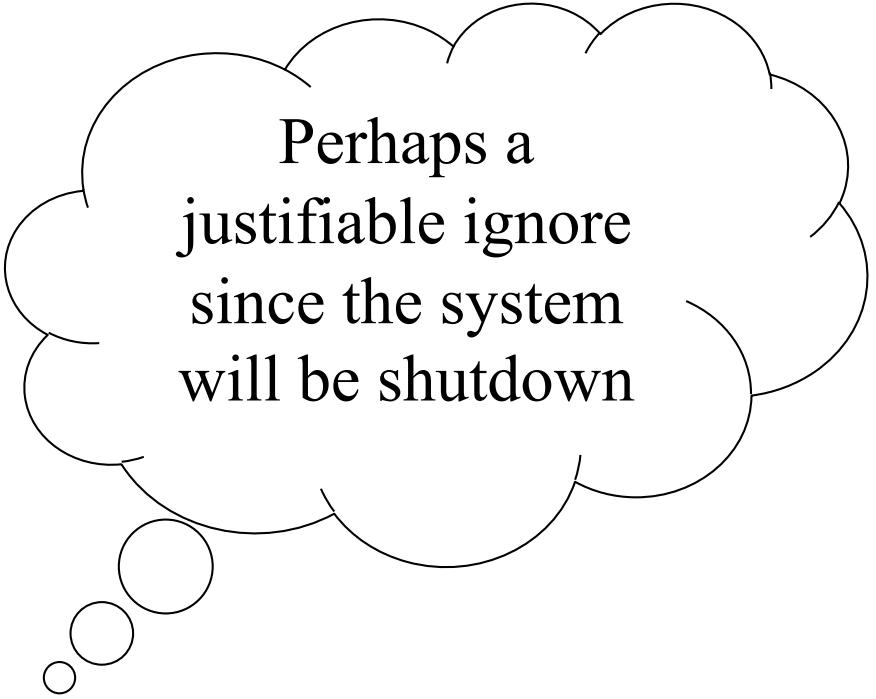
- Ignoring an exception is effectively ignoring a problem when there is one
- For less serious errors this will not cause problems, however, ignoring exceptions is dangerous as your systems robustness may be compromised
- Therefore, do not ignore exceptions unless there is a very good reason for it

Ignore

- To ignore an exception, create a try catch block and catch the Exception you wish to ignore
- Leaving the catch block blank will ignore the exception COMPLETELY!!!!

Ignore

```
try {  
    database.close();  
}  
catch (IOException e) {  
    //The next thing that happens is that the  
    //system shuts down so ignoring the  
    //exception  
}
```



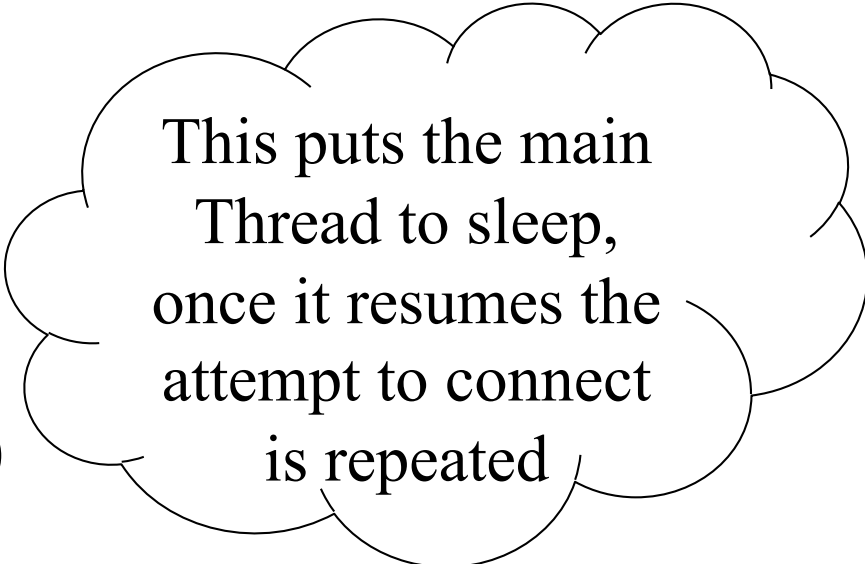
Perhaps a
justifiable ignore
since the system
will be shutdown

5. Retry the action

- It's also possible for the catch statement to attempt to retry the action after some set period of time
- For instance if the application was attempting to load a Webpage or connect to a server it could be a good idea to wait and then try again

Retry action

```
try {  
    database.connect();  
}  
catch (IOException e) {  
    try {  
        Thread.sleep(10000); //waits 10000 milliseconds  
    }  
    catch (InterruptedException ex) { //No action taken }  
    try { //Second attempt  
        database.connect();  
    }  
    catch (IOException ex) {  
        abortConnection();  
    }  
}
```



This puts the main
Thread to sleep,
once it resumes the
attempt to connect
is repeated

6. Prepare for shutdown

- In the case where there is a serious fault a handler may decide to minimize the negative impact of the failure by tidying up resources or closing open connections before shutting down
- In this case the catch statement performs the necessary functions

Custom Exceptions

- In Java it is possible to create your very own exceptions
- It is more common to use the existing Java Exception classes, however, there may be situations where the existing classes do not adequately describe the condition
- The programmer can then create their own “meaningful” exceptions for their program

Custom Exceptions

- There are three basic steps to created and using your own exceptions:
 1. Firstly you create your own Exception class and this class extends the generic Exception class
 2. Second, you need to decide where this exception will be thrown
 3. Thirdly you need to handle or declare the exception within your program (or someone else using your program)

Custom Exception Example

- Suppose we were creating a SIMS type game and needed to create families, these families must satisfy the following requirements:
 - Within each Family it is NOT permitted to have the same first name
 - Within each Family ALL of the family members MUST have the same surname

FirstNameExistsException

```
package com.raeside.family.exceptions;

public class FirstNameExistsException extends Exception {

    private static final long serialVersionUID=123456;

    public FirstNameExistsException(String message) {
        super(message);
    }

}
```

FirstNameExistsException

- This exception is designed to deal with the scenario where a new member is added to the family but has the same first name
- Note: It extends the *Exception* superclass
- Note: It calls the *super()* constructor and passes the message to the super constructor

FirstNameExistsException

- Next we need to decide where in our system this exception is throw
- In the Family example this exception will be throw in the following method of the Family class:

```
public void addFamilyMember(Person member) throws FirstNameExistsException {
```

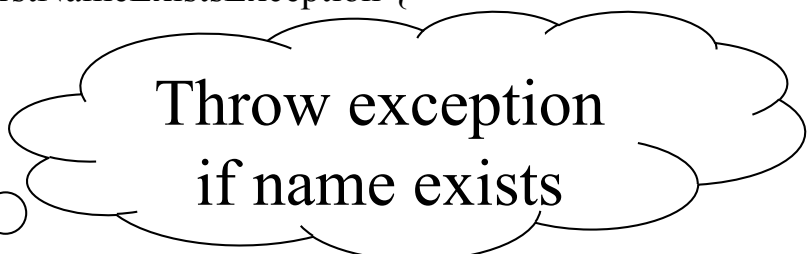
```
    if(newFirstName(member.getFirstName())) {  
        familyMembers.addElement(member);  
    }
```

```
    else {
```

```
        throw new FirstNameExistsException("This family already has a " +  
            member.getFirstName());
```

```
    }
```

```
}
```

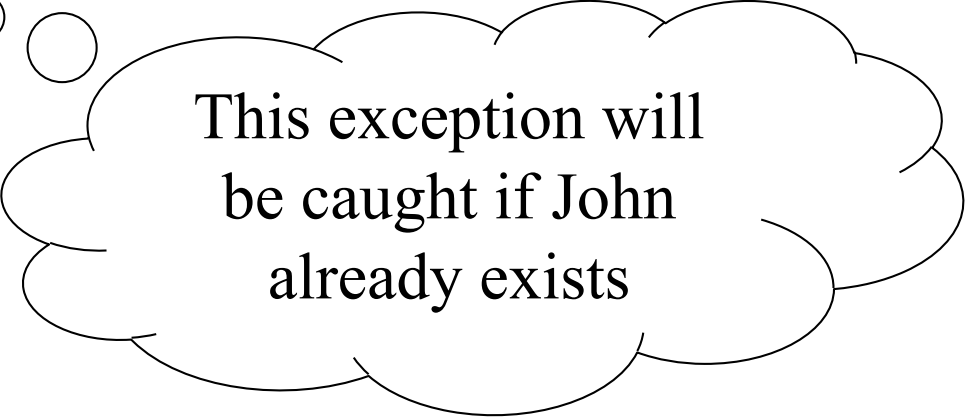


Throw exception
if name exists

FirstNameExistsException

- Finally we need to handle the exception in some client class e.g.:

```
try {  
    robinsons.addFamilyMember(new Person("John","Robinson");  
}  
catch(FirstNameExistsException ex) {  
    ex.printStackTrace();  
}
```



This exception will
be caught if John
already exists

Labwork – Exercise 1

- Create a new project in Eclipse
- Test the Java API file output example that was shown in this lecture (class called `LoggingFileOutputTest`)
- Find the file that was output and examine the contents
- Run the program a few more times and examine the error file!!!

Labwork – Exercise 2

- Create a new project in Eclipse
- Create a JFrame application called MyCalculatorGUI, this program must contain TWO JButtons labelled “Double” and “Triple”. Add TWO JTextFields to the GUI, one text field to input an integer and one textfield to OUTPUT the “Double” or “Triple” of the number entered by the user (you may re-use code from previous weeks’ exercises)
- Place a try–catch block around the calculation code in the actionPerformed method. Specifically catch the NumberFormatException. Output a JOptionPane error to the user and state the following “Error only a number may be entered into the JTextField”.
- Test this program by entering a letter into the JTextField instead of a number.

Labwork – Exercise 3

- Create a new project in Eclipse
- Import the *com.raeside.family* and *com.raeside.exceptions* packages into this project (available on the student share). Run an test this program (run the *MakeFamilyRobinson* class). **(Turn up at the lab and I'll help you with these steps!!!)**
- Add a new exception class to the existing system called *SurnameMismatchException*. Add some meaningful message to the exception e.g. “You must have surname Robinson to join this family”
- Declare the *addFamilyMember()* method to throw *SurnameMismatchException* aswell as the *FirstNameExistsException*. You will need to use the *correctFamilyName()* method to check that the *Person* being added to the *Family* has the correct surname, throw a new *SurnameMismatchException*.
- Finally HANDLE this exception in the *MakeFamilyRobinson* class in addition to handling the *FirstNameExistsException*. Test the new exception class is working by attempting to add a *Person* called “Jessy James” to the Robinson family.