Lecture overview

- What are exceptions?
- Why have exceptions?
- ‘Handling’ versus ‘Declaring’ exceptions
- Exception Classes and the hierarchy
- Following stack traces

What are exceptions?

- Programming is difficult!
- Why?, because trying to cover all of the possibilities of what could go wrong in your programs is complex, if not impossible
- Some of the things that go wrong are predictable and should be handled by a good programmer
- However, even the best written code is subject to unforeseen errors

Why are exceptions used?

- Exceptions in Java provide a programmer with the ability to prepare for possible errors in programs

- Exceptions provide the programmer with the opportunity to take pre-emptive action should some exceptional circumstances arise

Exception example

- Let’s examine the following code:

This will not compile!!!…why not??

```
import java.io.*;

public class ExceptionExample1 {

    public static void main(String[] args) {

            System.out.println("Enter text and I'll echo it");
            InputStreamReader istream = new InputStreamReader(System.in);
            BufferedReader input = new BufferedReader(istream);
            String inputLine = input.readLine();
            System.out.println("Read: " + inputLine);
    }

}
```

Exception example cntd.

- The following error is shown by the compiler, so what does it mean???:

```
ExceptionExample1.java:10: unreported exception
java.io.IOException; must be caught or declared to be thrown
        String inputLine = input.readLine();
```

- Simply put, the above error is informing you that this IO method can generate errors and you (the programmer) must take responsibility for that possibility!!!

Exception example cntd.

```
public static void main(String[] args) {

        try {
                System.out.println("Enter text and I'll echo it");
                InputStreamReader istream = new InputStreamReader(System.in);
                BufferedReader input = new BufferedReader(istream);
                String inputLine = input.readLine();
                System.out.println("Read: " + inputLine);
        }
        catch(IOException e) {
                System.out.println("The input went wrong");
        }

}
```

No compile error here!!!

Exception example cntd.

- Once the programmer has included the *try..catch* block around the *readLine()* code the programmer has fulfilled their obligation to acknowledge this probable error!!!

- But there is another way to get this code to compile!!!

Exception example cntd.

```
import java.io.*;

public class ExceptionExample1 {

    public static void main(String[] args) throws IOException {

                System.out.println("Enter text and I'll echo it");
                InputStreamReader istream = new InputStreamReader(System.in);
                BufferedReader input = new BufferedReader(istream);
                String inputLine = input.readLine();
                System.out.println("Read: " + inputLine);
    }
}
```

This compiles also…without try—catch!!!

Exception example cntd.

- In the previous slide the programmer has also acknowledged that the code can cause IO exceptions

- The difference here the is that this programmer has “passed the buck”, i.e., the programmer has decided not to deal with this exception and passes the error on to whomever calls this code!!!

Handling versus Declaring Exceptions

- In the first example the programmer “handled” the exception, handling an exception uses a *try—catch* block
- In the second example the programmer “declared” the exception, declaring an exception uses a *throws* statement

Handling Exceptions

- As stated previously handling exceptions uses the *try—catch* block

```
try {


}
catch (Exception exName) {


}
```

try

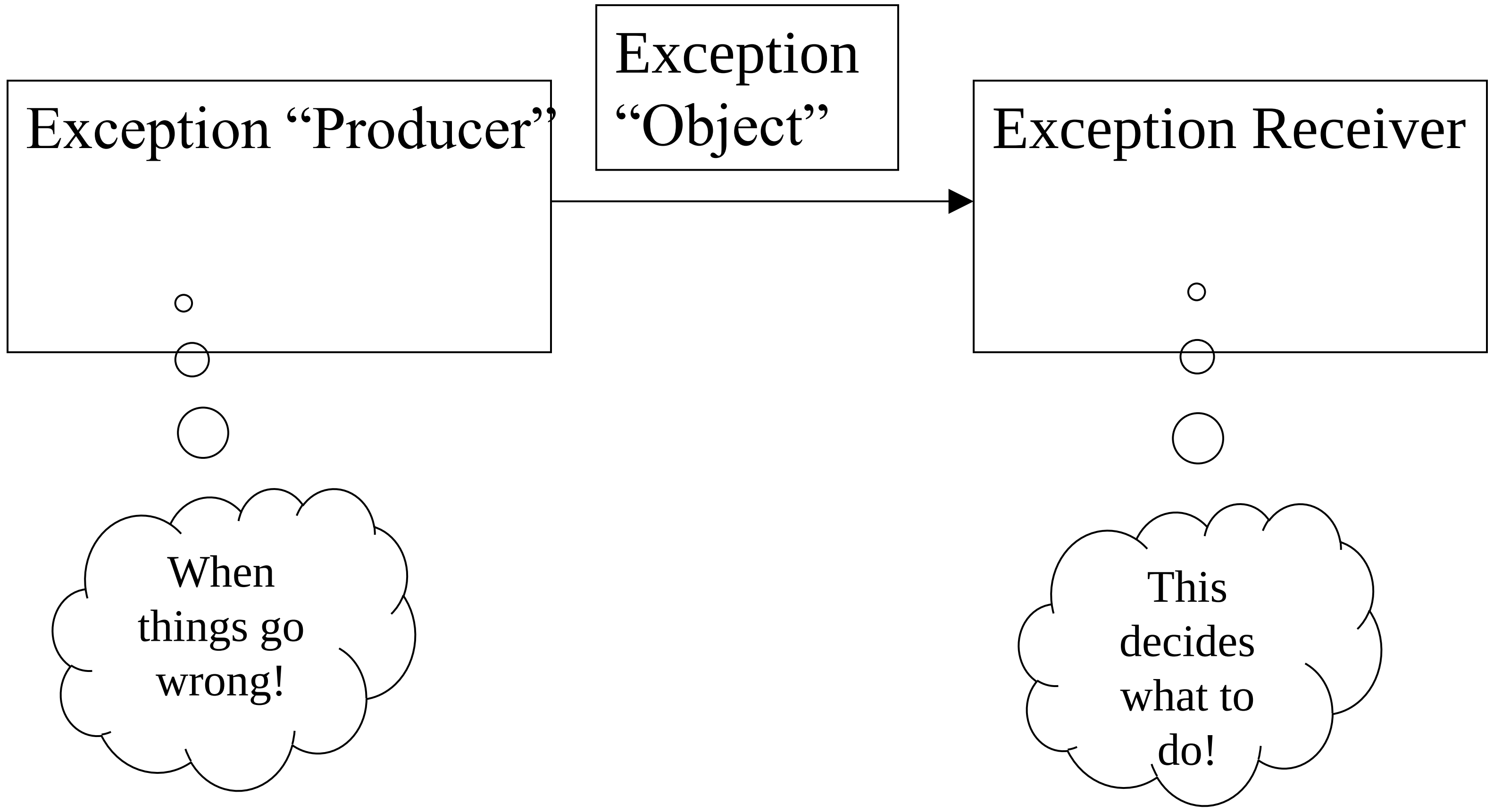
- Placing a *try* block around a code statements indicates that there are possibilities for exceptional circumstances within the enclosed block

- Assuming that an exception does occur then it is the attached catch statement(s) that deals with the error!!!

catch

- This is the “handler” code
- If something went wrong in the try block this is what the programmer wishes to do in order to handle this exception gracefully (the alternative might be to let the system crash!!!)
- The catch must have a particular type of exception associated with it!!!

Exception Mechanism



finally

- *try—catch* blocks may also have another block attached

- This block will execute no matter whether an exception has occurred or not

- This block is called a *finally* block

try - catch - finally

```
try {
    //Something can go wrong in here!!!
}
catch (IOException exp) {
    //If something does go wrong and it’s an IOException, come in here
}
finally {
    //I don’t care whether something goes wrong or not, just do this anyway!!!
}
```

Handling Exceptions

1. A try block must always be followed by a catch or a finally, may have both but must have <u>at least ONE</u>!!!

2. The order of the blocks is enforced, i.e., if your code uses *catch* and *finally*, catch block must follow the try

Handling Exceptions

3. No statements allowed between *try* and its corresponding *catch* and finally blocks

4. A *catch* block is associated with a particular type

5. A *try* can have multiple *catch* blocks. The first matching block will be called (Rule of thumb: Catch the most specific exceptions first)

Handling Exceptions

6. A finally block will always be executed to completion unless:

    i. The JVM crashes (System.exit(int))
    ii. Unhandled exception is thrown in the finally block
    iii. A catastrophe, e.g., unplug the computer

7. You can ‘nest’ *try-catch-finally*

8. You can re-throw an exception inside the *try-catch-finally* structure

Checked versus Unchecked Exceptions

- There’s a big distinction between *checked* and *unchecked* exceptions in Java

- Checked exceptions are verified by the compiler, i.e., the compiler insists on the exception either being handled or declared (like the *IOException* example shown earlier, the compile error indicated that we needed to catch or throw…the compiler <u>checked</u> to make sure!!)

- Unchecked exceptions or NOT subject to the ‘handle or declare’ rule

Unchecked Exceptions

- Descendents of *RuntimeException* are examples of unchecked exceptions, the compiler will not check that you have caught or thrown the exception
- *RuntimeException*'s can happen at any moment during the normal execution of the JVM
- The reason some exceptions are not checked is because the programmer should be checking for the potential errors, e.g. Null Pointers or dividing by zero
- If these exceptions were checked you would never be able to write a line of code without a *try* and *catch*…so things would get messy!!!

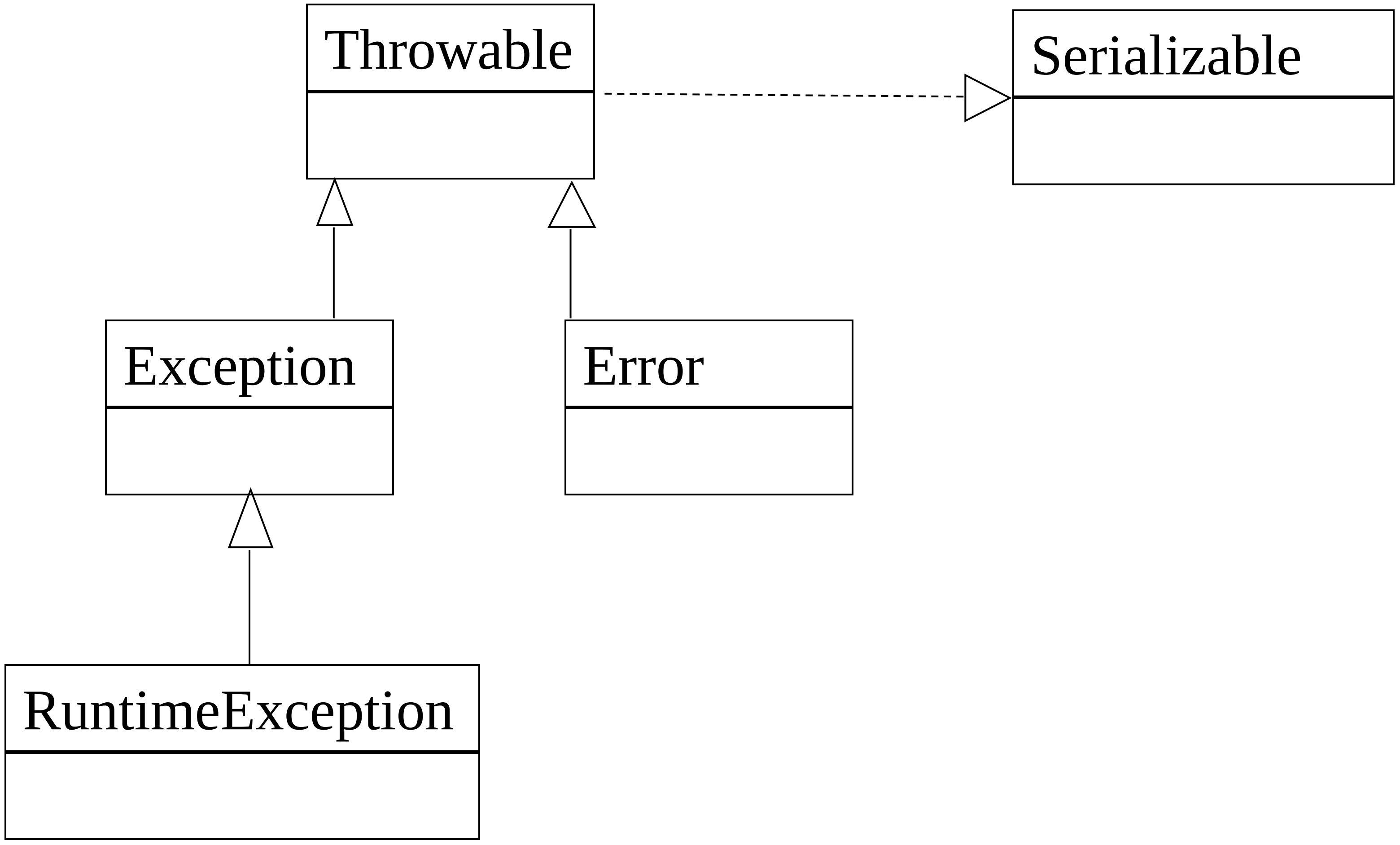
Declaring Exceptions

1. You must declare any *checked exceptions* that your method can throw

2. You may declare *unchecked exceptions*, but you are not required to

3. Calling methods must follow the *handle* or *declare* rule for any *checked* exceptions

Exception class and hierarchy



Exception class hierarchy

- Throwable class:
  - This is the common ancestor to all exceptions
  - Represents some abnormal condition in the code that can be transmitted through Java using the exception propagation mechanism

Exception class hierarchy

- Exception class:
  - This is a descendent of Throwable
  - Represents a *potentially recoverable* abnormal condition in a program
  - Usually represents a moderate abnormal condition; the cause can be pinpointed

Exception class hierarchy

- Error class:
  - This is a descendent of Throwable
  - Represents a potentially unrecoverable abnormal condition in a program not necessarily under the control of the programmer
  - Usually represents a serious problem, e.g. *OutOfMemoryError* or *StackOverflowError*

Features of *Throwable* class

- The Throwable class holds all of the generic attributes and behaviours of Exceptions

- Three of the more interesting features are:
  - Message
  - Stack Trace
  - Cause

Message

- This is of class *String*
- This provides a more detailed description of the exception being thrown, i.e., its for the client user to read
- It can be set in the constructors of an exception
- It can be retrieved using the *getMessage* method

Stack trace

- This records the call stack for the program
- It supplies clues to the programmer that can be followed, usually include line numbers and method names where the error occurred
- Can be forced to output using the method: *printStackTrace()*

Print a Stack trace

```
public class StackTraces {

    public static void main(String[] args) {
            traceMe();
    }

    public static void traceMe() {
            try{
                        Object s = null;
                        s.toString();;
            }
            catch(Exception e) {
              e.printStackTrace();
            }
    }
}
```

WARNING!: This program does not work, it generates an exception and prints the trace

Cause

- This property can designate another Throwable object as the “root” cause of the problem
- Can be set using the constructor for the Throwable object

Labwork – Exercise 1

- Create a project in Eclipse
- Create a class called TryCatchDemo with a main method
- Test the *try—catch—finally* block behaviours using *System.out.println* statements (Hint: You might wish to force a *NullPointerException* and catch for that….turn up for the lab and I’ll help!!!)
- Print a stack trace for the catch block using the *printStackTrace()* method

Labwork – Exercise 2

- Create a new project in Eclipse
- Add the *CheckedVUnchecked.java* class to the project (you'll find it in under code listing for lecture 3)
- Call the *unchecked()* method from within the main
- Attempt to call the *checked()* method in a similar way….comment an explanation of the error that was generated into the java
- Get the class to compile with both the *checked* method and *unchecked* methods called