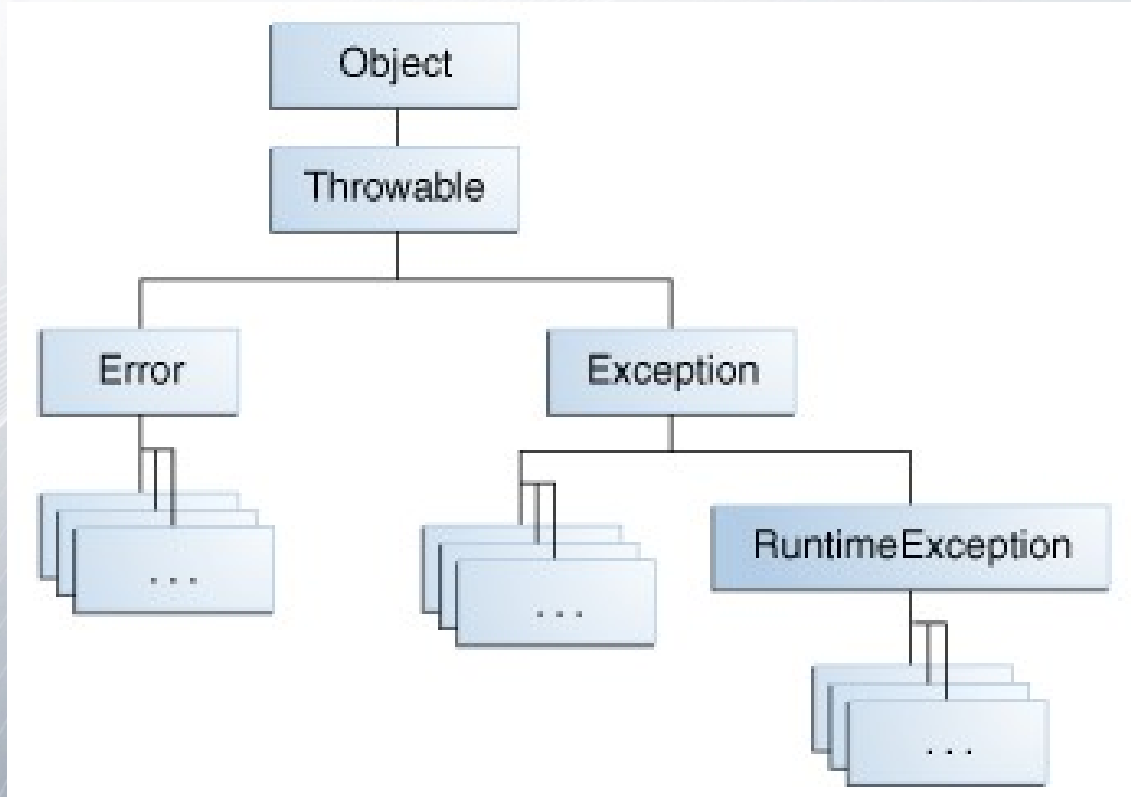
Lecture 13

Exceptions (part 2)

Using debugger in Eclipse IDE

Throwable class and its subclasses

- Throwable – parent of all exceptions
- There are 3 types of exceptions:
  - Errors
  - Unchecked exceptions
  - Checked exception



More about Throwable class

An instance of Throwable class contains

- Message
- Stacktrace
- Cause (instace of Throwable)

More about Throwable class

Constructors:

```
public Throwable()
public Throwable(String message)
public Throwable(Throwable cause)
public Throwable(String message, Throwable cause)
```

Important methods:

```
public String getMessage()
public Throwable getCause()
public void printStackTrace()
public StackTraceElement[] getStackTrace()
```

Chained Exceptions

An application often responds to an exception by throwing another exception.

In effect, the first exception causes the second exception.

It can be very helpful to know when one exception causes another.

Chained Exceptions help the programmer do this.

Wrapped exceptions (cause)

Why are necessary?

Chained Exceptions example

In this example, when an IOException is caught, a new SampleException exception is created with the original cause attached and the chain of exceptions is thrown up to the next higher level exception handler.

```
try {
    //...
} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```

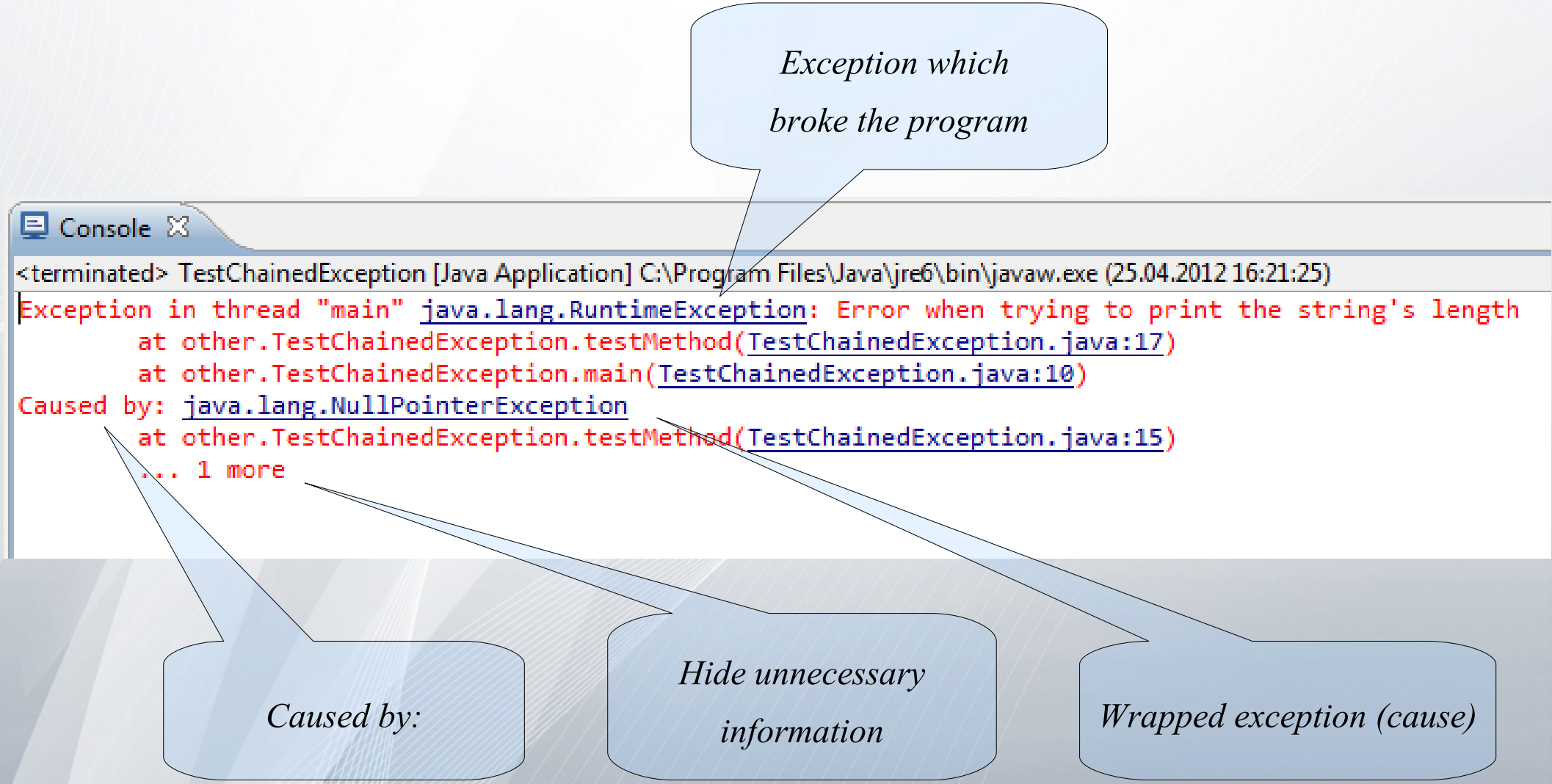
How to read stacktrace with chained exceptions

```
package other;

public class TestChainedException {
    public static void main(String[] args) {
        String s = null;
        testMethod(s);
    }

    public static void testMethod(String s) {
        try {
            System.out.println(s.length());
        } catch (NullPointerException npe) {
            throw new RuntimeException("Error when trying to print the
string's length", npe);
        }
    }
}
```

How to read stacktrace with chained exceptions



How exceptions should be shown to the end user

- The end user is not programmer
- So, it's not a good practice to show technical details (stacktrace) to the end user
- Instead, nice message should be shown
- If we want, we can add technical information but it should be shown only if the user want to see it

Global handler

- It's good practice to write a global handler in some upper layer which handle all uncatched exceptions (unchecked exceptions)
- This way we'll prevent user from unhandled exceptions and showing unpleasant messages

Question

What happens with this code?

```
try {
    //..
} catch (Exception e) {
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}
```

Answer

Compilaton error!

Unreachable catch block for IOException. It is already handled by the catch block for Exception

Because IOException extends class Exception,

so the second catch block will never execute.

We can handle multiple exceptions using catch block with parent class.

This is useful when we want to handle more than one exceptions in the same way

(we use a (too) general exception handler)

Re-throwing exception

- Sometimes we want to handle the exception just for a moment, use it for something (write in the log) and then re-throw it, because we can't handle it at all.
- Keyword *throw* is used(we saw it in the previous slides)

```
try {
    //...
} catch (IOException e) {
    logger.log(Level.WARNING, "Error in testMethod: " + e.getMessage());
    throw e;
}
```

Defining own exceptions

- Just extends the class Exception
- Do not create a subclass of RuntimeException or throw a RuntimeException
- If we need, we can add some fields to these which is inherited by Exception
- It's good practice each module to throw only his own exceptions
- For readable code, it's good practice to append the string Exception to the names of all classes that inherit from the Exception class.

Defining own exceptions

```
public class SoftAcadException extends Exception{

    private static final long serialVersionUID = 905082714139195048 3L;

    public SoftAcadException() {
        super();
    }

    public SoftAcadException(String message, Throwable cause) {
        super(message, cause);
    }

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

    public SoftAcadException(Throwable cause) {
        super(cause);
    }

}
```

This field is not mandatory.
We'll explain it later in the course,
when we talk about serialization.

Finally block

- The finally block always executes when the try block exits.
- This ensures that the finally block is executed even if an unexpected exception occurs
- it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.
- Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.

Finally block

The finally block is a key tool for preventing resource leaks. When closing a file or otherwise recovering resources, place the code in a finally block to ensure that resource is always recovered.

```
try {
    // some code which open PrintWriter out
} catch (Exception e) {
    //.. handle exception
} finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```

Catch block is not mandatory.

finally can exist without catch,

but not without try.

Let's discuss some good practices when using exceptions

Using eclipse debugger

- What is bug?
- A common term used to describe an error, mistake, failure, or fault in a computer program that produces an incorrect or unexpected result.
- What is debugger?
- A debugger or debugging tool is a computer program that is used to test and debug other programs (the "target" program)

Now, let's try to debug some code in eclipse