# ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of February 23, 2013

## Abstract


- Handling errors
  - Declaring, creating and handling exceptions
  - Checked and unchecked exceptions
  - Creating new exception types


*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

# Error processing

Errors can occur at compile-time or runtime.

Syntax errors are detected at compile time.

Since Java is a strongly typed language, the compiler will also ensure that each expression is valid, i.e. only valid operations for this data type are performed, avoiding errors that would otherwise occur during the execution of the program.

Although type checking is an efficient way of detecting errors as soon as possible, **there are certain verifications that can only be made at execution time**; e.g. checking that a stack contains an element before removing it.

In Java, runtime errors are handled by a mechanism called Exceptions.

# Sources of errors

- Logic of the program;
- External events: running out of memory, write error, etc.

Detecting and handling errors will make our programs more robust.

Ideally, those mechanisms should help locating precisely the source of the errors.

For instance, the method **pop** of the **ArrayStack** implementation, could throw an **IndexOutOfBoundsException**. Is the source of the error necessarily in the method **pop**?

No. In fact, it is likely that the error situation occurred because the caller did not check if the stack was empty before calling the method **pop**.

# Pre-conditions

**Pre-conditions** are the ensemble of conditions that the **parameters** and the **state** of the object, must verify so that the computation can be successful.

A familiar example is the calculation of the square root of a number. The pre-condition is that the parameter be a non-negative number. Depending on the programming language used or the library, the validity of the calculation is not guaranteed: some programs may loop, others may crash.

In the case of **ArrayStack**, an attempt to remove an element when the stack is empty is an example of an error condition; here, the **state** of the object was inconsistent with the actual goal of the method **pop**.

It's a good programming habit to always think about the preconditions of a method.

**From now on, try to identify the preconditions when writing a method.**

# Of the value of top. . .

# Handling errors

What should we do?

Crash the program (e.g. $1/0$), exit, . . . certainly not.

Write an error message? Good idea but it is not enough.

**Traces** are useful during the development of a program but should be removed from its final version.

# Handling errors

It would be possible to return a special value, **sentinel**, whenever an error is detected.

Analogy. Search by content methods return the index of the element that is sought and -1 if not found.

However, the solution is not general enough, nor effective.

- Some methods don't have a return value or don't have special values that can be used; think of a method that returns a **boolean** value, **member(x)** of a list;

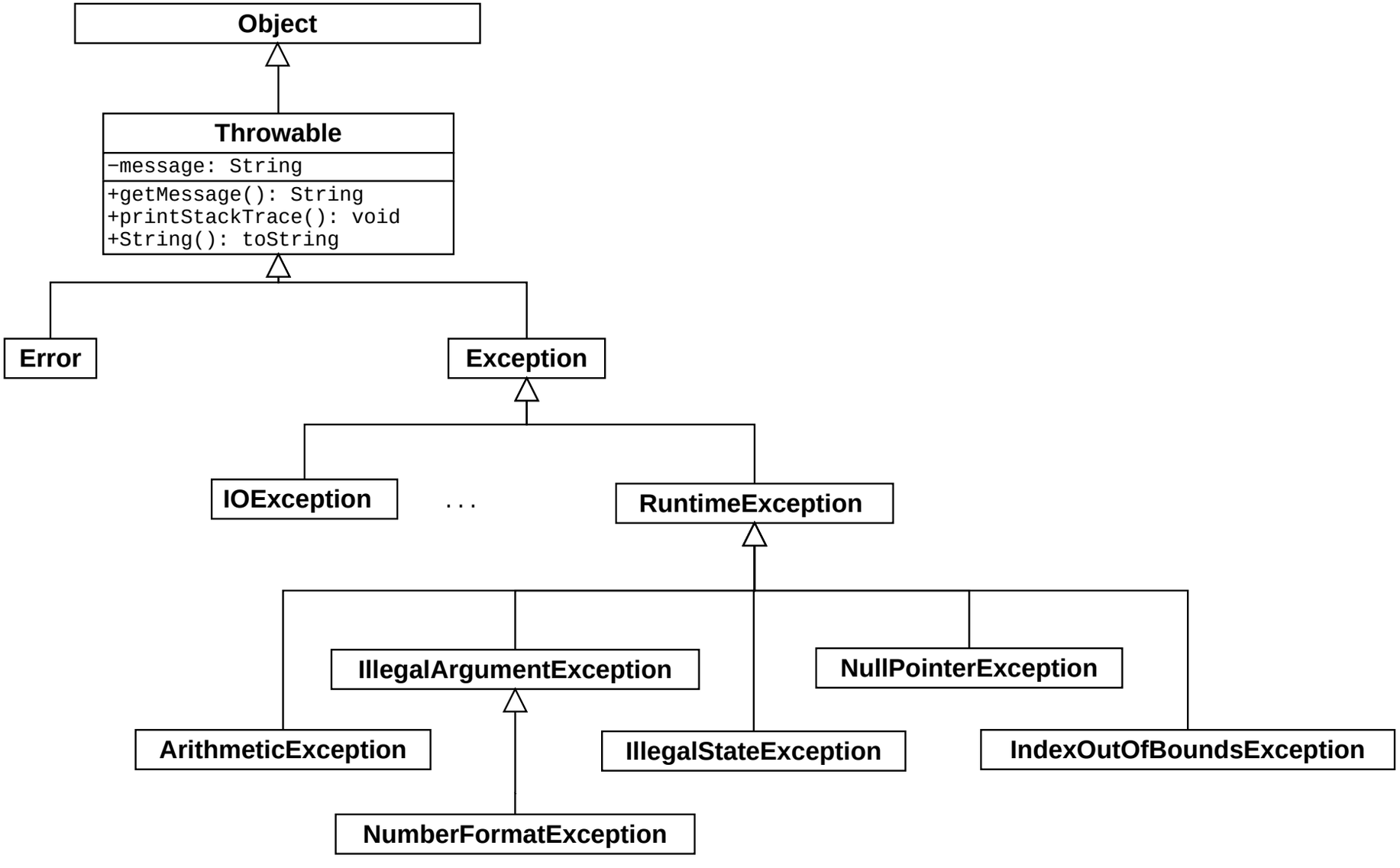
- The caller is not forced to handle the error situation.

# Handling errors

Error handling is a general problem and modern languages offer solutions. In Java the mechanism is called Exceptions.

**Exceptions are objects**.

# Exception is a class



# Exception is a class

**The Exceptions are objects!**

An error situation is modelled using an object of the class **Throwable**, or one of its sub-classes. The object encapsulates the error message.

The class **Throwable** declares the methods **String getMessage()** and **void printStackTrace()**.

Declaring a reference of type **Exception** (it's simply a class),

```
Exception e;
```

Creating an object of the class Exception (it's simply a class),

```
e = new Exception( "Houston, we have a problem!" );
```

# Signaling an error

Consider the class **ArrayStack** and its method **pop()**.

The empty stack is an illegal case (illegal state).

Since we didn't have better tools we used to insert a comment indicating the necessary pre-condition:

```
// if ( top == -1 ) BOOM!
```

In Java, the way to handle illegal cases is to "**throw**" an exception.

```
if ( top == -1 ) {
   throw new IllegalStateException( "Empty stack" );
}
```

Throw statement:

**throw expression**

where **expression** is a reference to an object of the class **Throwable**, or one of its subclasses.

```
IllegalStateException e;

e = new IllegalStateException( "Empty stack" );

if ( top == -1 ) {
   throw e;
}
```

It's **throw** that changes the flow of control! Not the declaration, not the creation of the object.

# Transfer of control

When an **Exception** is thrown:

- The statement or expression *terminates abruptly*;
- If uncaught, the **Exception** will cause the whole stack of method calls to unwind, i.e. each method call on the stack will terminate abruptly. If the exception is not caught the program (thread) will terminate and print a stack trace;
- None of the statements or parts of the expression that follow the point where the exception was thrown are executed;
- Following an exception the next statements to be ran will be those of a **catch** or **finally** block.

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

        System.out.println( "-1-" );

        throw new RuntimeException( "an Exception" );

        System.out.println( "-2-" );

    }
}
```

**Does not compile. Why? Ways around the problem?**

```
javac Test.java
Test.java:8: unreachable statement
        System.out.println( "-2-" );
        ^
1 error
```

Once an **Exception** has occurred the method terminates abruptly.

```
class Test {
    public static void main ( String[] args ) {
        System.out.println( "-1-" );
        if ( true ) {
            throw new RuntimeException( "Help!" );
        }
        System.out.println( "-2-" );
    }
}
```

```
> java Test
-1-
Exception in thread "main" java.lang.RuntimeException: Help!
an Exception at Test.main(Test.java:5)
```

**Notice that -2- has not been printed!**

## Not even the remaining part of an expression is executed.

```
public class Test extends Object {
    public static boolean error() {
        if ( true ) {
            throw new RuntimeException( "an Exception" );
        }
        return true;
    }
    public static boolean display() {
        System.out.println( "-2-" );
        return true;
    }
    public static void main( String[] args ) {
        System.out.println( "-1-" );
        if ( error() || display() ) {
            System.out.println( "-3-" );
        }
        System.out.println( "-4-" );
    }
}
```

```
> java Test
-1-
Exception in thread "main" java.lang.RuntimeException: an Exception
        at Test.error(Test.java:5)
        at Test.main(Test.java:16)
```

## Unwinding the stack of method calls:

```
public class Test {
    public static void c() {
        System.out.println( "c: -1-" );
        if ( true ) {
            throw new RuntimeException( "top of the world!" );
        }
        System.out.println( "c: -2-" );
    }
    public static void b() {
        System.out.println( "b: -1-" );
        c();
        System.out.println( "b: -2-" );
    }
    public static void a() {
        System.out.println( "a: -1-" );
        b();
        System.out.println( "a: -2-" );
    }
    public static void main( String[] args ) {
        System.out.println( "m: -1-" );
        a();
        System.out.println( "m: -2-" );
    }
}
```

```
> java Test
m: -1-
a: -1-
b: -1-
c: -1-
Exception in thread "main" java.lang.Exception: top of the world!
top of the stack
        at Test.c(Test.java:6)
        at Test.b(Test.java:11)
        at Test.a(Test.java:16)
        at Test.main(Test.java:21)
```

# Handling Exceptions

Statements that can throw an exception are enclosed in a `try/catch` block:

```
try {
    statements;
} catch ( exception_type1 id1 ) {
    statements;
} catch ( exception_type2 id2 ) {
    statements;
...
} finally {
    statements;
}
```

If no exception occurs only the statements of the try (and finally) block(s) are executed.

```
public class Grill {
    private Burner burner = new Burner();
    public void cooking() {
        try {
            burner.on();
            addSteak();
            addSaltAndPepper();
            boolean done = false;
            while (! done) {
                done = checkSteak();
            }
        } catch ( OutOfGazException e1 ) {
            callRetailer();
        } catch ( FireException e2 ) {
            extinguishFire();
        } finally {
            burner.off();
        }
    }
}
```

```
int DEFAULT_VALUE = 1;
int value;

try {

    value = Integer.parseInt( "100" );

} catch ( NumberFormatException e ) {

    value = DEFAULT_VALUE;

}

System.out.println( "value = " + value );
```

$\Rightarrow$ prints “value = 100”.

public static int parseInt(String s) throws NumberFormatException

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the parseInt(java.lang.String, int) method.

Parameters:
s - a String containing the int representation to be parsed

Returns:
the integer value represented by the argument in decimal.

Throws:
NumberFormatException - if the string does not contain a parsable integer.

```
int DEFAULT_VALUE = 1;
int value;

try {

    value = Integer.parseInt( "Team A" );

} catch ( NumberFormatException e ) {

    value = DEFAULT_VALUE;

}

System.out.println( "value = " + value );
```

$\Rightarrow$ prints “value = 1”.

If an exception is thrown, the statements of the first **catch** block that matches the exception are executed.

Match means that the exception is of the specified type or a subtype of it.

No other block will be executed.

If no catch matches the exception it percolates outside the try block.

Statements in the **finally** block are always executed.

Blocks **finally** are used to close files, for example.

"**Catch**" expressions must be as specific as possible.

```
int DEFAULT_VALUE = 1;
int value;

try {

    value = Integer.parseInt( "a" );

} catch ( Exception e ) {

    value = DEFAULT_VALUE;

}

System.out.println( "value = " + value );
```

Why? The block above would catch any **Exception** type.

However, the catch statement is not designed to handle any **Exception**, what should it do if a **ClassNotFoundException** occurred?

It should let it percolate so that an enclosing block can catch it.

```
public class Test {
    public static void c() {
        System.out.println( "c() :: about to throw exception" );
        throw new RuntimeException( "from c()" );
    }
    public static void b() {
        System.out.println( "b() :: pre-" );
        c();
        System.out.println( "b() :: post-" );
    }
    public static void a() {
        System.out.println( "a() :: pre-" );
        try {
            b();
        } catch ( RuntimeException e ) {
            System.out.println( "a() :: caught exception" );
        }
        System.out.println( "a() :: calling b, no try block" );
        b();
        System.out.println( "a() :: post-" );
    }
    public static void main( String[] args ) {
        System.out.println( "main( ... ) :: pre-" );
        a();
        System.out.println( "main( ... ) :: post-" );
    }
}
```

```
main( ... ) :: pre-
a() :: pre-
b() :: pre-
c() :: about to throw exception
a() :: caught exception
a() :: calling b, no try block
b() :: pre-
c() :: about to throw exception
Exception in thread "main" java.lang.RuntimeException: from c
        at Test.c(Test.java:4)
        at Test.b(Test.java:8)
        at Test.a(Test.java:19)
        at Test.main(Test.java:24)
```

# Exercise

```
public class Test {
    public static void c() {
        System.out.println( "c() :: about to throw exception" );
        throw new RuntimeException( "from c()" );
    }
    public static void a() {
        System.out.println( "a() :: pre-" );
        try {
            c();
        } catch ( NumberFormatException e ) {
            System.out.println( "a() :: caught exception" );
        } finally {
            System.out.println( "finally" );
        }
        System.out.println( "a() :: calling b, no try block" );
        c();
        System.out.println( "a() :: post-" );
    }
    public static void main( String[] args ) {
        System.out.println( "main( ... ) :: pre-" );
        a();
        System.out.println( "main( ... ) :: post-" );
    }
}
```

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

        boolean valid = false; int value = -1;

        for ( int i=0; i<args.length && ! valid; i++ ) {

            try {
                value = Integer.parseInt( args[i] );
                valid = true;
            } catch ( NumberFormatException e ) {
                System.out.println( "not a valid number: " + args[i] );
            }

        }
        if ( valid ) {
            System.out.println( "value = " + value );
        } else {
            System.out.println( "no valid number was found" );
        }
    }
}
```

```
> java Test a 1.1 "" 2 3
not a valid number: a
not a valid number: 1.1
not a valid number:
value = 2
```

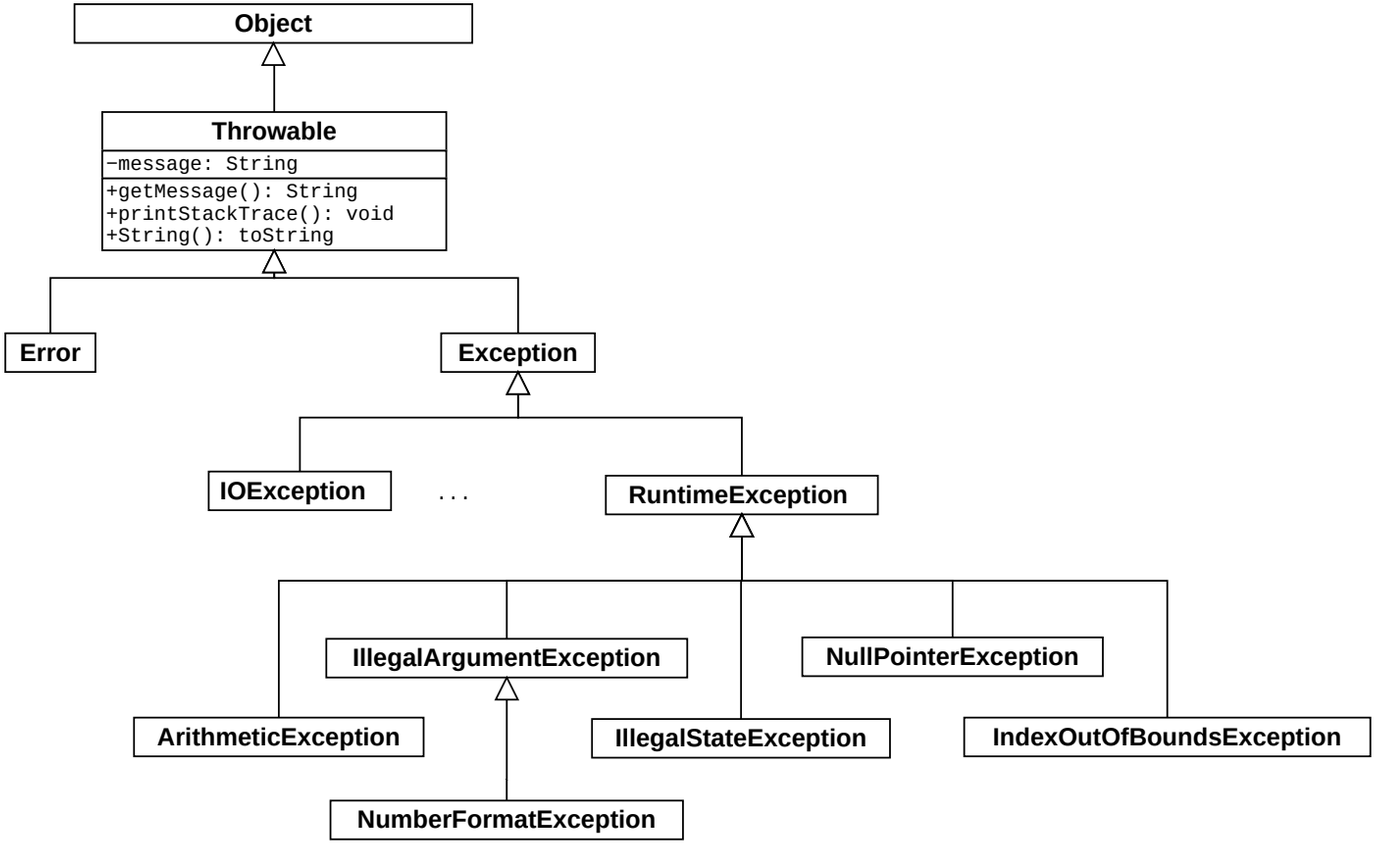
```
int DEFAULT_VALUE = 1;
int value;

try {
    value = Integer.parseInt( "a" );
} catch ( NumberFormatException e ) {

    System.out.println( "warning: " + e.getMessage() );

    value = DEFAULT_VALUE;
}
```

$\Rightarrow$ **e** is a reference to the object that was thrown, like any reference variable you can select the name you want, here we're calling the method **getMessage()** of the exception designated by **e**.




There are "**checked**" and "**unchecked**" exceptions. All the exceptions that are subclasses of **Throwable** are "**unchecked**" (default). Except those that are subclasses of **Exception**, those are "**checked**". Except those that are from a subclass of "**RuntimeException**", which are "**unchecked**".

# Checked and unchecked exceptions

A method that throws a "**checked exception**" must **declare** or **handle** the exception.

```
class Test {
    public static void main( String[] args ) {
        System.out.println( "-1-" );
        if ( true ) {
            throw new Exception( "an Exception" );
        }
        System.out.println( "-2-" );
    }
}
```

```
> javac Test.java
Test.java:5: unreported exception java.lang.Exception;
must be caught or declared to be thrown
            throw new Exception("an Exception");
```

^

# Checked and unchecked exceptions

A method that throws a "**checked exception**" must **declare** or **handle** the exception.

```
class Test {
    public static void main( String[] args ) throws Exception {
        System.out.println( "-1-" );
        if ( true ) {
            throw new Exception( "an Exception" );
        }
        System.out.println( "-2-" );
    }
}
```

# Checked and unchecked exceptions

**Unchecked** exceptions include **NullPointerException** and **IndexOutOfBounds-Exception**.

Any method that declares a reference variable can potentially throw a **NullPointerException**!

Similarly, any method that uses an array can potentially throw a **IndexOutOfBoundsException**!

Under normal circumstances (i.e. unless there is a bug) these methods should not be throwing exceptions.

**Java does not require you to declare (or catch) unchecked exceptions**.

# Checked and unchecked exceptions

**Checked** exceptions are used to **force** the caller to define a strategy for handling these exceptions (declare or catch).

An example of a checked exception is **IOException**.

Often used to deal with situations caused by external events: read/write error.

Used in contexts where the caller can take actions to handle the error.

# Handling exceptions

Methods that could potentially throw checked exceptions must declare the exceptions.

- Catching an exception;
- **Declaring an exception (throws)**.

```
public static void main( String[] args )
    throws IOException, FileNotFoundException, IllegalAccessException {

    ...
}
```

# Handling exceptions

A method can declare checked and unchecked exceptions; the declaration of unchecked exceptions is optional.

```
public static void main( String[] args )
    throws IOException, FileNotFoundException, NullPointerException {

    ...
}
```

Here, **NullPointerException**, an unchecked exception, is optional.

# Transitive effects (1/5)

```
import java.io.*;

public class Keyboard {
    public static int getInt() {
        byte[] buffer = new byte[ 256 ];

        System.in.read( buffer ); // throws IOException

        String s = new String( buffer );
        int num = Integer.parseInt( s.trim() );
        return num;
    }
    public static void main( String[] args ) {
        System.out.print( "Please enter a number: " );
        int n = Keyboard.getInt();
        System.out.println( "You've entered the number: " + n );
    }
}
```

# Transitive effects (1/5)

```
> javac Keybord.java
Keyboard.java:9: unreported exception java.io.IOException;
must be caught or declared to be thrown
     System.in.read(buffer);
                    ^
1 error
```

# Transitive effects (2/5)

```
import java.io.*;

public class Keyboard {
    public static int getInt() throws IOException {
        byte[] buffer = new byte[ 256 ];

        System.in.read( buffer ); // throws IOException

        String s = new String( buffer );
        int num = Integer.parseInt( s.trim() );
        return num;
    }
    public static void main( String[] args ) {
        System.out.print( "Please enter a number: " );

        int n = Keyboard.getInt(); // throws IOException

        System.out.println( "You've entered the number: " + n );
    }
}
```

# Transitive effects (2/5)

```
> javac Keyboard.java
Keyboard.java:22: unreported java.io.IOException;
must be caught or declared to be thrown
     int n = Keyboard.getInt();
                              ^
1 error
```

# Transitive effects (3/5)

```
import java.io.*;

public class Keyboard {
    public static int getInt() throws IOException {
        byte[] buffer = new byte[ 256 ];
        System.in.read( buffer ); // throws IOException
        String s = new String( buffer );
        int num = Integer.parseInt( s.trim() );
        return num;
    }
    public static void main( String[] args )
    throws IOException {
        System.out.print( "Please enter a number: " );
        int n = Keyboard.getInt(); // throws IOException
        System.out.println( "You've entered the number: " + n );
    }
}
```

# Transitive effects (3/5)

```
> java Keyboard
Please enter a number: oups
Exception in thread "main" java.lang.NumberFormatException
For input string: "oups"
     at java.lang.NumberFormatException.
        forInputString(NumberFormatException.java:48)
     at java.lang.Integer.parseInt(Integer.java:468)
     at java.lang.Integer.parseInt(Integer.java:518)
     at Keyboard.getInt(Keyboard.java:13)
     at Keyboard.main(Keyboard.java:23)
```

# throws

A method that can possibly throw checked[1] Exceptions, i.e.:

- throws an Exception itself; or
- calls a method that throws an exception and
- the exceptions are not caught by a block `finally` or `catch`,

must declare all those exceptions:

```
void method()
throws EOFException, FileNotFoundException {
     ...
}
```

[1] not a subclass of Error or RuntimeException

# Creating New Exception Types

Extends **Exception** or one of its subclasses.

Extending **Exception** (or one of its subclasses, except the subclasses of **RuntimeException**) is creating a **checked** exception.

Extending **RuntimeException** (or one of its subclasses) is creating an **unchecked** exception.

```
public class MyException extends Exception {
}
```

```
public class MyException extends Exception {
    public MyException() {
        super();
    }
    public MyException( String message ) {
        super( message );
    }
}
```

Why creating new types?

To specialize an exception, or to add new implementation details.

Because exceptions are caught selectively using their type (filter).

```
try {
 ...
} catch ( Exception e ) {
    // can you really handle any type of error?
}
```

```
try {
 ...
} catch ( NumberFormatException e ) {
    var = DEFAULT_VALUE;
}
```

To document the program and its execution.

# Class Time

In the following example, the method **parseTime** catches **NumberFormatException** and **NoSuchElementException**, and throws a more informative type, **TimeFormatException**.

```
public class TimeFormatException extends IllegalArgumentException {

    public TimeFormatException() {
        super();
    }

    public TimeFormatException( String msg ) {
        super( msg );
    }
}
```

```
public class Time {

    // ...

    public static Time parseTime( String timeString ) {

        StringTokenizer st = new StringTokenizer( timeString, ":", true );

        int h, m, s;

        try {
            h = Integer.parseInt( st.nextToken() );
        } catch( NumberFormatException e1 ) {
            throw new TimeFormatException( "first field is not a number");
        } catch( NoSuchElementException e2 ) {
            throw new TimeFormatException( "first separator not found" );
        }

        try {
            st.nextToken();
        } catch( NoSuchElementException e2 ) {
            throw new TimeFormatException( "first separator not found" );
```

```
}

try {
    m = Integer.parseInt( st.nextToken () );
} catch( NumberFormatException e1 ) {
    throw new TimeFormatException( "second field is not a number" );
} catch( NoSuchElementException e2 ) {
    throw new TimeFormatException( "second separator not found" );
}

try {
    st.nextToken();
} catch( NoSuchElementException e2 ) {
    throw new TimeFormatException( "second separator not found" );
}

try {
    s = Integer.parseInt( st.nextToken() );
} catch( NumberFormatException e1 ) {
    throw new TimeFormatException( "third field is not a number" );
} catch( NoSuchElementException e2 ) {
    throw new TimeFormatException( "third field not found" );
```

```
        }

        if ( st.hasMoreTokens() )
            throw new TimeFormatException( "invalid suffix );

        if ( (h<0) || (h>23) || (m<0) || (m>59) || (s<0) || (s>59) )
            throw new TimeFormatException( "values out of range:" + timeString );

        return new Time( h,m,s );
    }
}
```

# Error message

```
throw new Exception( "meaningful message" );
```

# Remarks

Use existing exception types as much as possible instead of defining new ones.

Have clear messages.

Since programs can recuperate from exceptions make sure that the state of objects remain consistent.