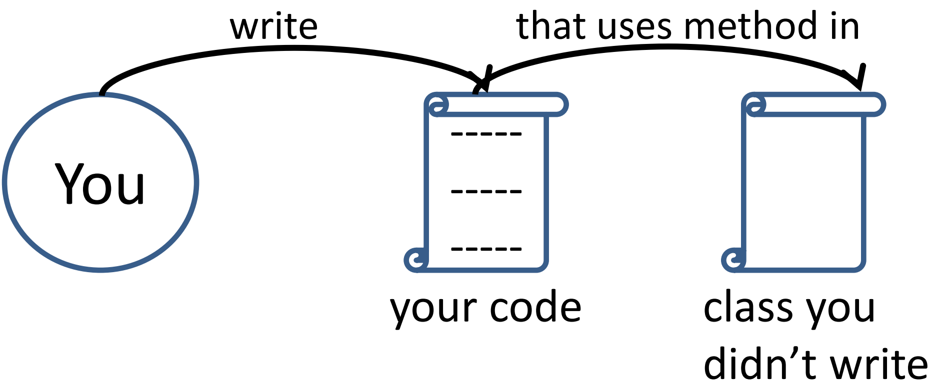
Risky behavior. Java exceptions

Lecture 20

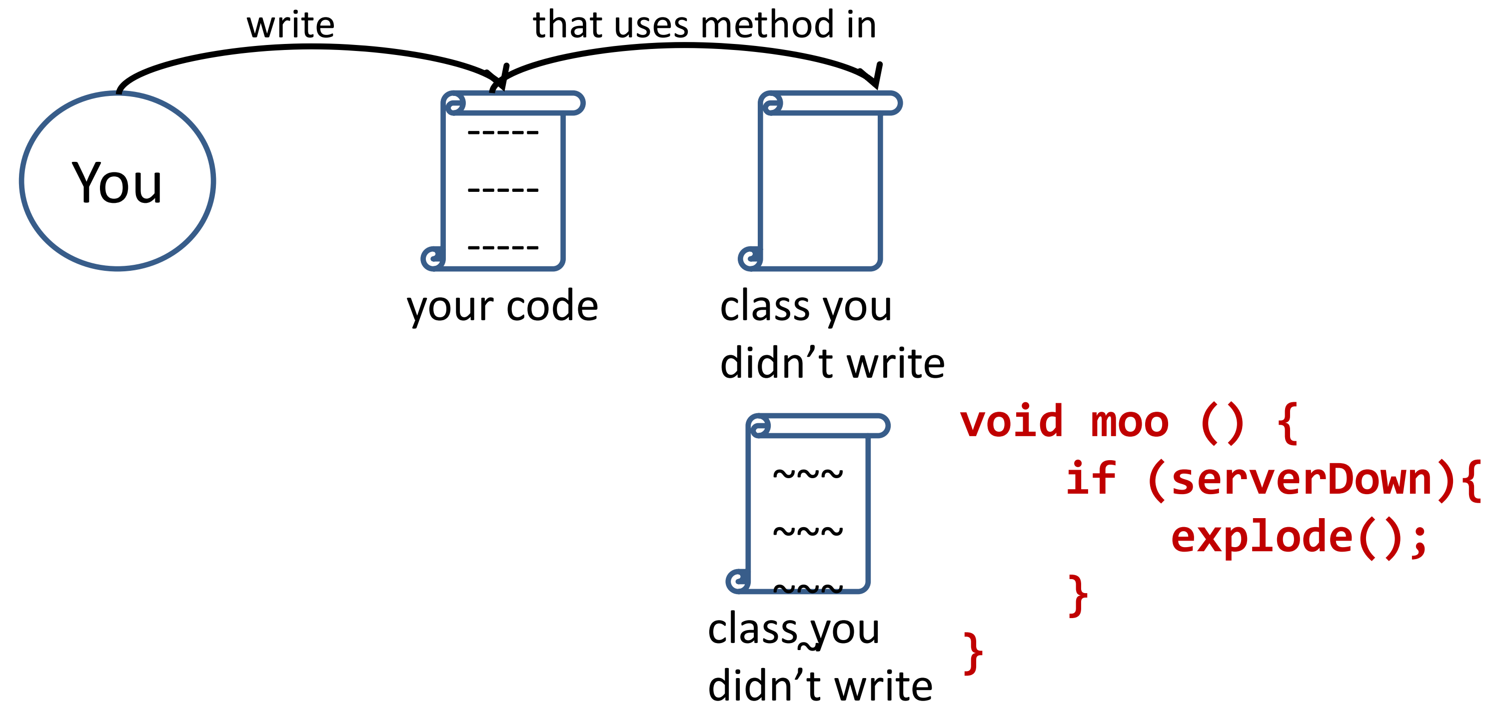
- Even the most carefully designed system may fail:
  - The file isn't there
  - The server is down
  - …
- When you write a risky method, you need code to handle the bad things that might happen

The general idea



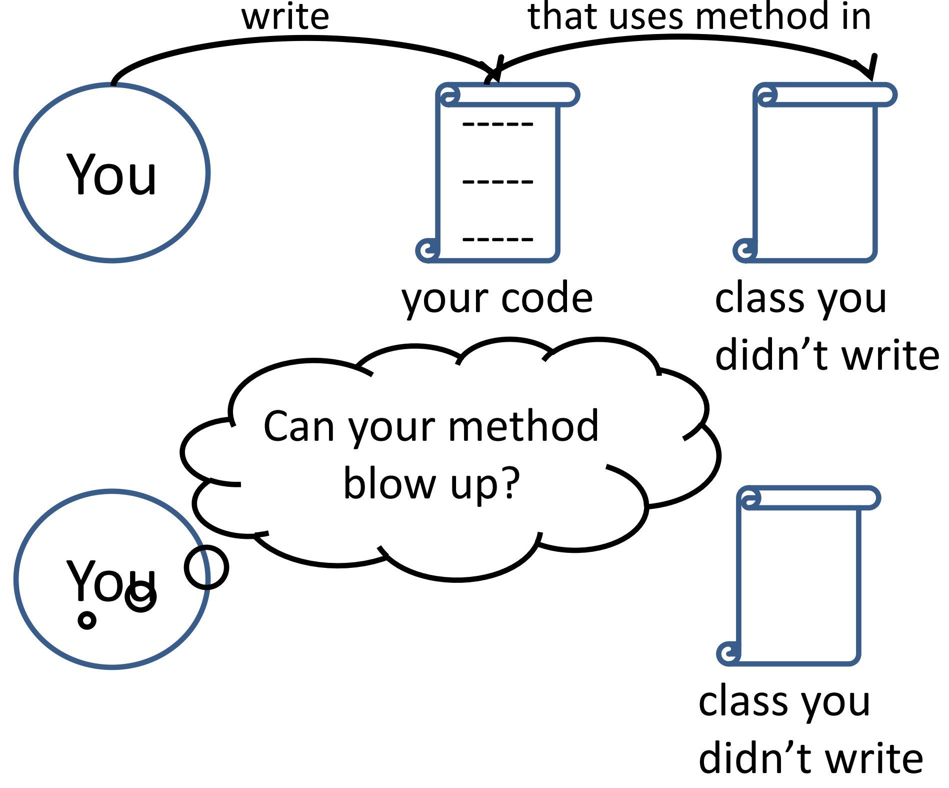
You want to call a method in a class that you did not write

The general idea



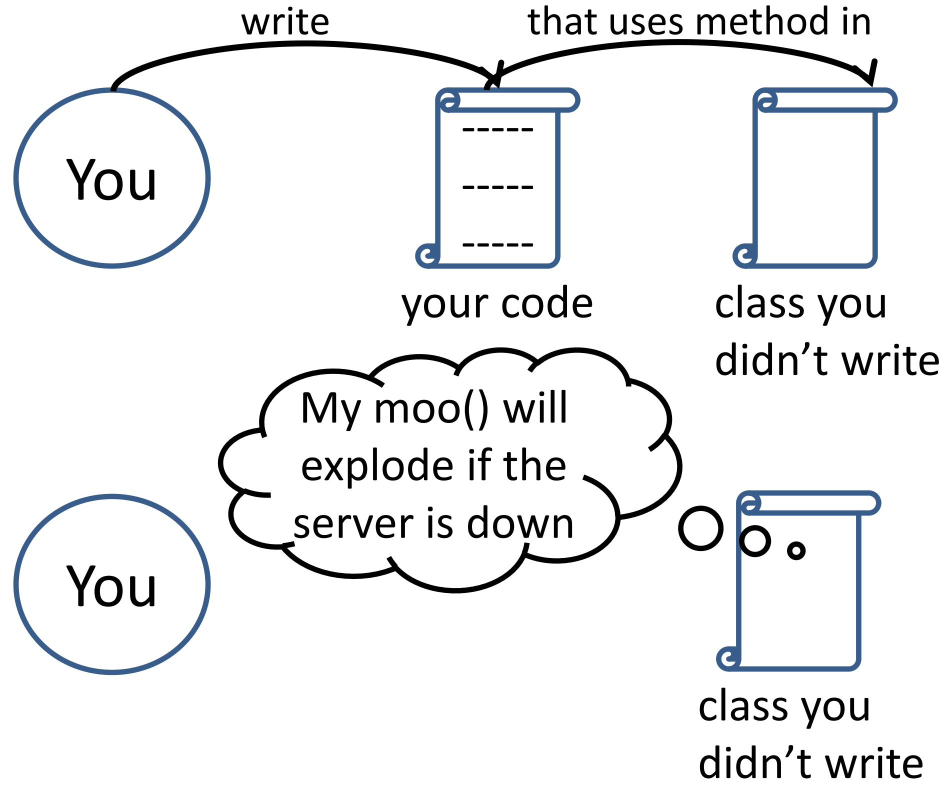
That method does something risky – might not work at runtime

The general idea



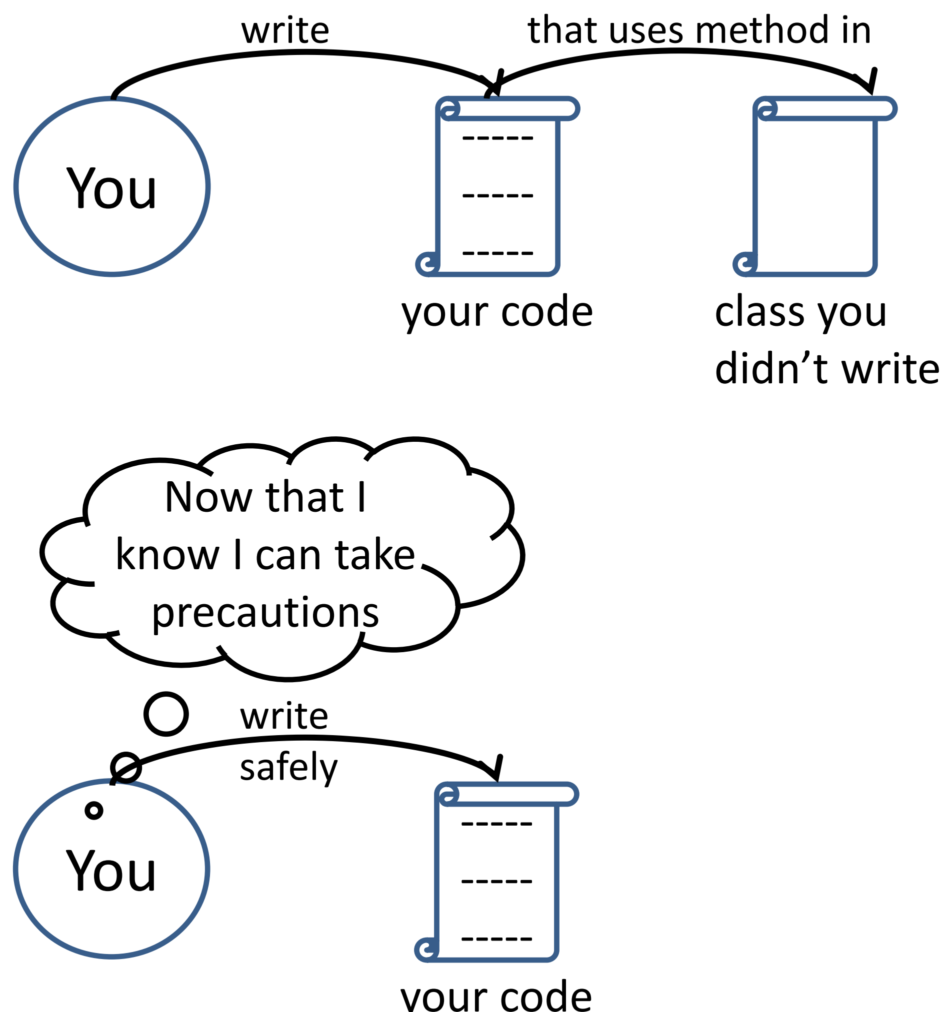
You need to know whether the method is risky

The general idea



You need to know whether the method is risky

The general idea



Then you can write code that can handle the failure if it happens

How to know that method throws exceptions

You find a **throws** clause in the risky method’s declaration

Example: BufferedReader readLine()

java.io
Class BufferedReader

public String readLine()
    throws IOException

Read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

Returns:

A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

Throws:

IOException - If an I/O error occurs

Try/catch block

```
try {
    //do risky thing
} catch (Exception e) {
    //try to recover
}
```

It is like declaring a method argument

This code only runs if an Exception is thrown

What to write in a catch block depends on the exception. If a server is down, you may try another server. If the file is not there, you may try to ask a user for a new location

try/catch block example

If you wrap the method which throws exception with **try**/**catch** block, then this tells compiler that you know: an exceptional thing may happen and you are prepared to handle it

```
import javax.sound.midi.*;
public class MusicTest {
      public void play() {
            try {
                  Sequencer sequencer =
                              MidiSystem. getSequencer();
                  System.out.println("Successfully got a
                                                  sequencer");
            } catch(MidiUnavailableException ex) {
                  System.out.println("Bummer");
            }
      }
```

try/catch block example

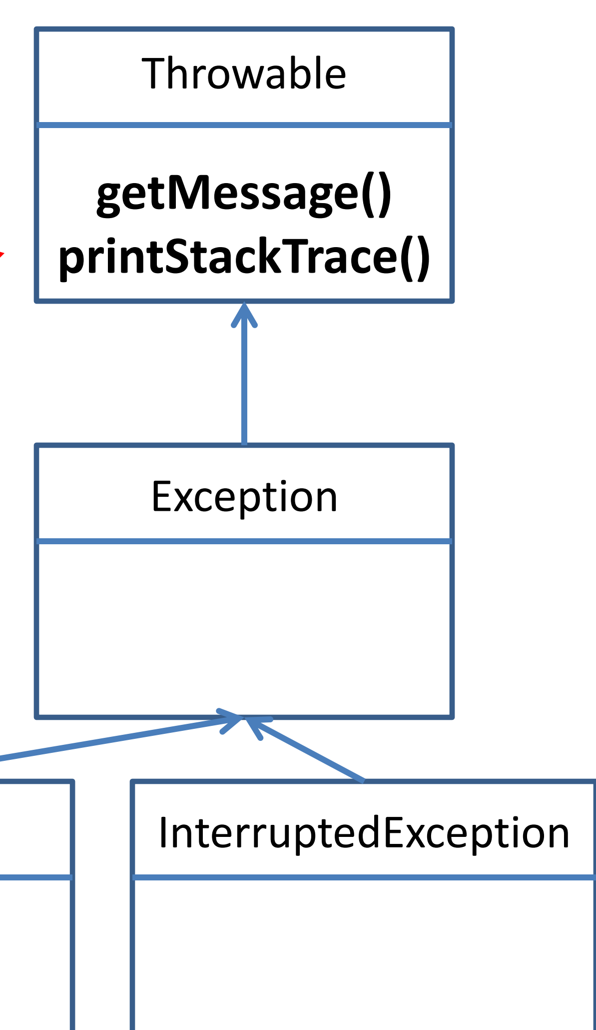
If you wrap the method which throws exception with **try**/**catch** block, then this tells compiler that you know: an exceptional thing may happen and you are prepared to handle it

Catches an exception which may be thrown from getSequencer() method

```
import javax.sound.midi.*;
public class MusicTest {
      public void play() {
            try {
                  Sequencer sequencer =
                              MidiSystem. getSequencer();
                  System.out.println("Successfully got a
                                          sequencer");
            } catch(MidiUnavailableException ex) {
                  System.out.println("Bummer");
            }
      }
```

An Exception is an object of type Exception

- It can be an instance of any subclass of Exception
- All exceptions inherit two key methods



Declaring an exception

This method MUST to declare that it throws a BadException

Risky, exception-throwing code:

```
public void takeRisk() throws BadException{
      if (abandonAllHope) {
            throw new BadException();
      }
}
```

Create a new Exception object and throw it

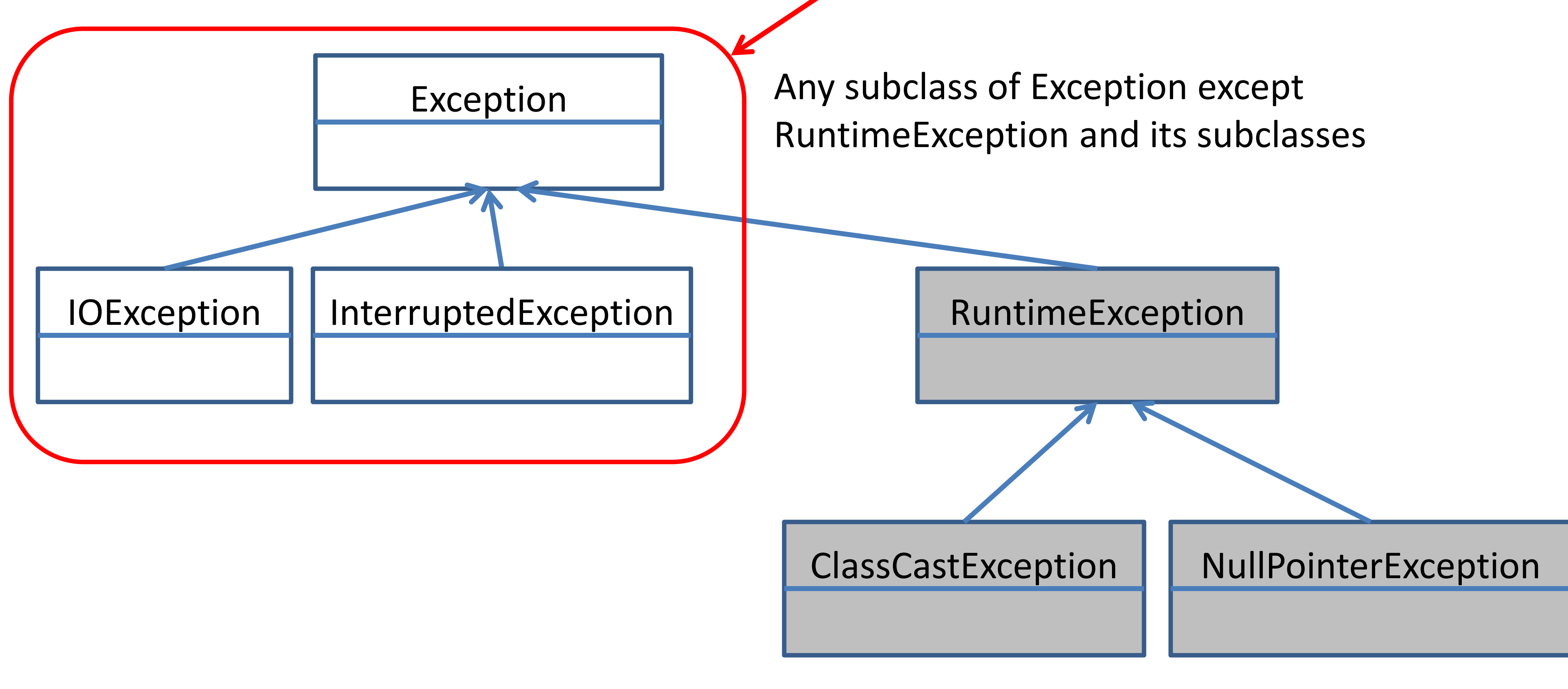
The code that calls the risky method:

```
public void crossFingers(){
    try {
        anObject.takeRisk();
    } catch (BadException e) {
        System.out.println("Aaargh!");
        e.printStackTrace();
    }
}
```

The compiler guarantees:

- If you throw Exception in your code, then you must declare it using the throws keyword in your method declaration
- If you call a method that throws an Exception, you must acknowledge that you are aware of the Exception possibility:
  - Try/catch block
  - Re-throw Exception

The compiler checks only "checked exceptions"



RuntimeExceptions are NOT checked by the compiler. You can throw, catch and declare, but you don't have to

Examples: RuntimeException

ArithmeticException: an exceptional arithmetic situation has arisen, such as integer division with zero divisor.

ClassCastException: attempt made to cast reference to an inappropriate type.

IllegalArgumentException: method invoked with invalid or inappropriate argument, or inappropriate object.

NullPointerException: attempt to use a null reference in case where an object reference was required.

SecurityException: a security violation was detected.

You WANT RuntimeExceptions to happen

- Most RuntimeExceptions come from a problem in your code logic
- You should make sure
  - that array index is not out of bound
  - that you do not try to call methods on *null*
  - that you do not divide by zero
- You don't want to catch and recover from something which should not happen in the first place

Bullet points

- A method can **throw an exception** if something fails at runtime
- An exception is always **an object of type Exception**
- The compiler does not pay attention to exceptions which are of type RuntimeException
- A method throws an exception using keyword **throw**
- A method that may throw a checked exception <u>must</u> announce this with a **throws Exception** declaration
- If your code calls the method that throws an Exception, it must reassure the compiler that the precautions have been taken:
  - Handle with **try/catch**
  - **Re-throw**

Flow control in try/catch block

If the try succeeds

```
try
{
    Foo f = x.doRiskyThing();     (1)
    int b = f.getNum();
} catch (Exception ex)
{
    System.out.println("failed");
}
System.out.println("We made it!");   (2)
```

If the try fails

```
try
{
    Foo f = x.doRiskyThing();     (1)
    int b = f.getNum();
} catch (Exception ex)
{
    System.out.println("failed");   (2)
}
System.out.println("We made it!");   (2)
```

Flow control excercise

```
1.  try {
2.       i=i/i;
3.       j=0;
4.       name=s.name();
5.       j=1;
6.  } catch (ArithmeticException e) {
7.       j=3; }
8.  } catch (NullPointerException e) {
9.       j=4; }
10. } catch (Exception e) {
11.      j=5;}
12. System.out.println (j) ;
```

Case 1: i!=0, s != null

Case 2: i=0

Case 3: i!=0, s=null

Finally: for the things you want to do no matter what

```
try {
        turnOvenOn();
        x.bake();
} catch (BakingException ex) {
        ex.printStackTrace();
} finally {
        turnOvenOff();
}
```

Here you put the code that must run **regardless** of an exception

Why do we need to clean resources when Java has garbage collector

The **finally** clause is necessary when we need to set something *other* than memory back to its original state: like an open file or network connection

The method can throw more than 1 exception

```
public class Laundry {
    public void doLaundry() throws PantsException, LingerieException {
            // code that could throw either exception
    }
}

public class Foo {
    public void go() {
        Laundry laundry = new Laundry();
        try {
            laundry.doLaundry();
        } catch(PantsException pex) {
                //pants recovery code
        } catch(LingerieException lex) {
                // lingerie recovery code
        }
    }
}
```

Exceptions are polymorphic

You can declare exceptions using a supertype of the exceptions you throw

```
public void doLaundry ()
      throws ClothingException
```

You can catch exceptions using a supertype of the exception thrown

```
try {
      laundry.doLaundry();
}catch (ClothingException e)
```

Good practice: Write a separate catch block for each unique exception

```
try {
   laundry.doLaundry();
} catch(TeeShirtException tex) {
  // recovery from TeeShirtException
} catch(PantsException lex) {
  // recovery from LingerieException
} catch(ClothingException cex) {
  // recovery from all others
}
```

TeeShirtException and PantsException need different recovery code, so we use different catch block

All other ClothingExceptions are caught here

Multiple catch blocks must be ordered from smallest to biggest

`catch(TeeShirtException tex)`

`catch(ShirtException sex)`

`catch(ClothingException cex)`

- The higher up in the inheritance tree, the bigger the catch "basket"
- The biggest of all catch arguments is type ***Exception***: it will catch any exception, including RuntimeException, so we do not generally use *Exception* type outside the program testing
- You can't put bigger baskets above smaller baskets (it won't compile)
- Siblings can be in any order, because they can't catch one another's exceptions

Re-throwing an exception

- When you don’t want to handle an exception, just re-throw it
- For this, declare that your method throws the same type of Exception
- Let the method that calls YOU catch the exception

```
public void foo() throws ReallyBadException(){
   //can call risky method without try/catch
   laundry.doLaundry();
}
```

What is happening on the stack

- If you do not catch an exception, then what happens if the risky method does throw an exception?
- When the method throws an exception, this method is popped off the stack immediately, and the exception is thrown to the next method on the top of the stack – the caller.
- But if the caller re-throws the exception, so the caller pops off the stack, and the exception is thrown to the next method and so on … where does it end?

Re-throwing an exception only delays inevitable

```
public class Washer {
   Laundry laundry=new Laundry();

   public void foo() throws ClothingException {
      laundry.doLaundry();
   }

   public static void main (String [] args)
            throws ClothingException {
      Washer a=new Washer();
      a.foo();
   }
}
```

Risky method that throws ClothingException

Stack

1

doLaundry() throws a ClothingException

doLaundry
foo
main

main() calls foo()

foo() calls doLaundry()

doLaundry() is running and throws a ClothingException

2

foo() re-throws the exception

foo
main

doLaundry() pops off the stack and the exception is thrown back to foo()

But foo() does not have try/catch, so…

3

main() re-throws the exception

main

foo() pops off the stack and the exception is thrown back to main()

But main() does not have try/catch, and nobody left but JVM

4

The JVM shuts down

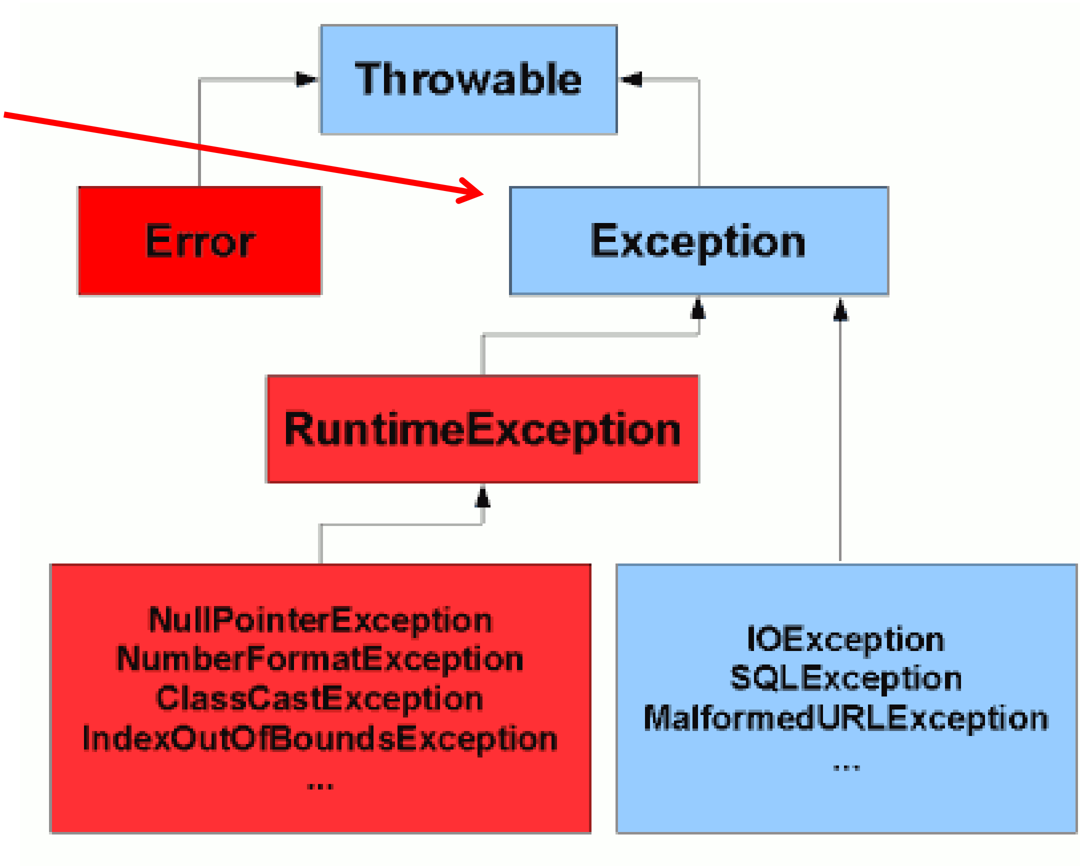
THE END

Handle or declare. It’s the law

- Handle: wrap the risky method in a try/catch block
- Declare: declare that your method throws the same exception as the risky method you are calling

Java Error class

Exception subclasses represent errors that a program can reasonably recover from -errors that are **expected to occur in the normal course of duty**



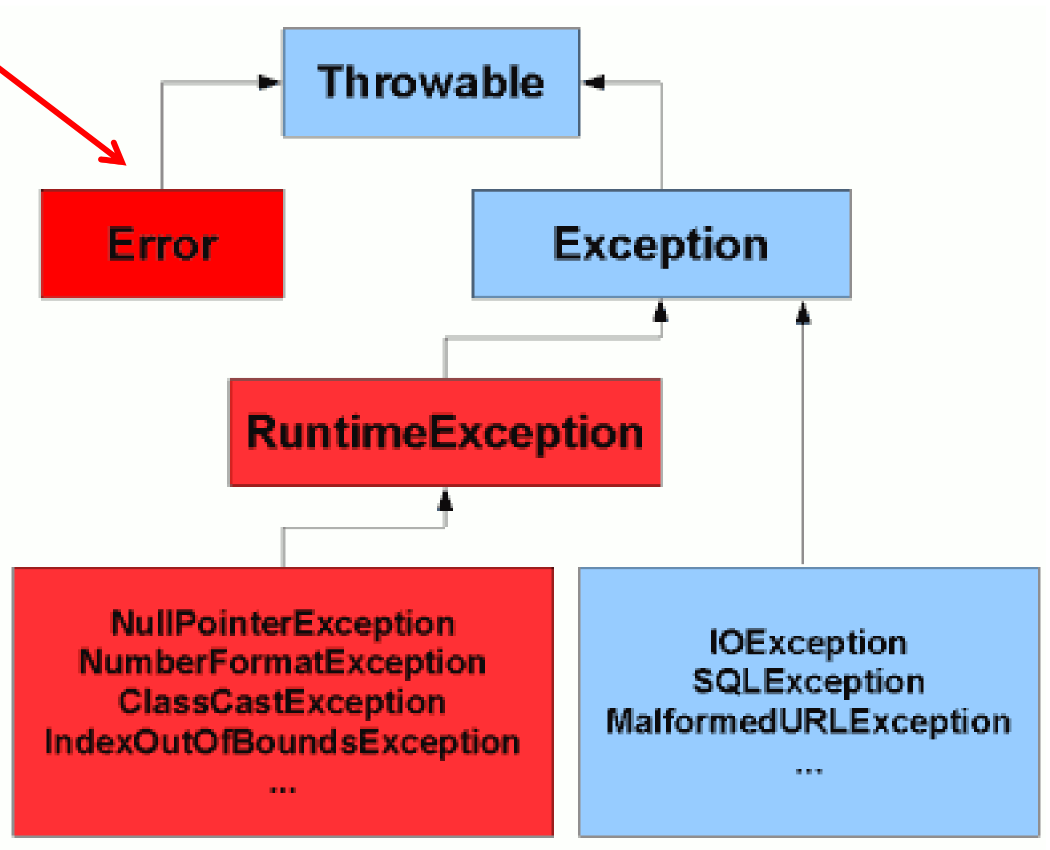
Java Error class

RuntimeExceptions are exceptions that a program **shouldn't generally expect to occur, but could potentially recover from.** They are likely to be **programming errors**

Throwable
Error
Exception
RuntimeException
NullPointerException
NumberFormatException
ClassCastException
IndexOutOfBoundsException
...
IOException
SQLException
MalformedURLException
...

Java Error class

Error subclasses represent "serious" errors that a progra generally **shouldn't expect to catch and recover from**. (an expected class file being missing, a StackOverflow, an OutOfMemoryError).



When (not) to use Exceptions

- Connect to a remote server
- Access an array beyond its length
- Display a window on the screen
- Get an input from the user
- Retrieve data from the database
- See if a text file is where you think it is
- Create a new file

Example: using exceptions

```
public static void placeShip(
      Board board, Ship ship)
      throws Exception
{
      ….
   tries--;
   if (tries < 1) {
   Exception e = new Exception("Could not
                               place ship");
   throw e;
}
```

Example: using exceptions

```
try {
    AI.placeShip(board,
                 board.ships[i]);
}catch (Exception e) {
    System.out.println(e.getMessage());
    System.out.println("Returning
                 to main menu.");
    return;
}
```