LUXOFT TRAINING

# Module 8

# Exceptions

# Unsafe code

Something may go wrong. We must be ready for that. How to control it?

**Option #1:** use error code and if blocks:

```
FileManager f = new FileManager();
boolean opened = f.openFile();
if (opened) {
    int success = f.readFile(strPtr);
    if (success) {
        int result = f.closeFile();
        if (result!=OK) {
            log("Cannot close the file");
        }
    } else {
        log("Cannot read from the file");
    }
} else {
    log("Cannot open the file");
}
```

Method must return 2 results at once:

1) Result of the execution
2) Success status

Application logic is mixed with the exception handling => we get a messy code.

# Unsafe code: use exceptions

Let FileManager methods may throw exceptions:

```
interface FileManager {
        public void openFile() throws FileNotFoundException;
        public String readFile() throws IOException;
        public void closeFile() throws FileCloseException;
}

FileManager f = new FileManagerImpl();
try {
        f.openFile();
        String str = f.readFile();
        f.closeFile();
} catch(FileNotFoundException e) {
        log("Cannot open the file");
} catch(IOException e) {
        log("Cannot read from the file");
} catch(FileCloseException e) {
        log("Cannot close the file");
}
```

*Safe code*

*Exception handlers*

**Advantages:**

- We can concentrate on code and do not think about exceptions
- Handling of all unsafe situations is placed to the single block

# Exception is a class

For example let us use ***if*** before:

```
if (person != null) {
    person.sendMessage(message);
} else {
    log("Person not found!");
}
```

Let create **PersonNotFoundException**:

```
class PersonNotFoundException extends Exception {}
```

And we can use it this way:

```
public Person findPerson(String name) {
    Person person = personDirectory.find(name);
    if (person == null) {
        throw new PersonNotFoundException();
    }
}
```

Now code to work with person will be like this:

```
try {
    Person person = findPerson("John Smith");
    person.sendMessage("Hello, John!");
} catch(PersonNotFoundException e) {
    log("Person not found!");
}
```

# Adding parameter to the exception

Let us save a person name:

```
class PersonNotFoundException extends Exception {
    String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Now the code to handle PersonNotFoundException will be like this:

```
try {
    Person person = findPerson("John Smith");
    person.sendMessage("Hello, John!");
} catch(PersonNotFoundException e) {
    log("Person "+e.getName()+" not found!");
}
```

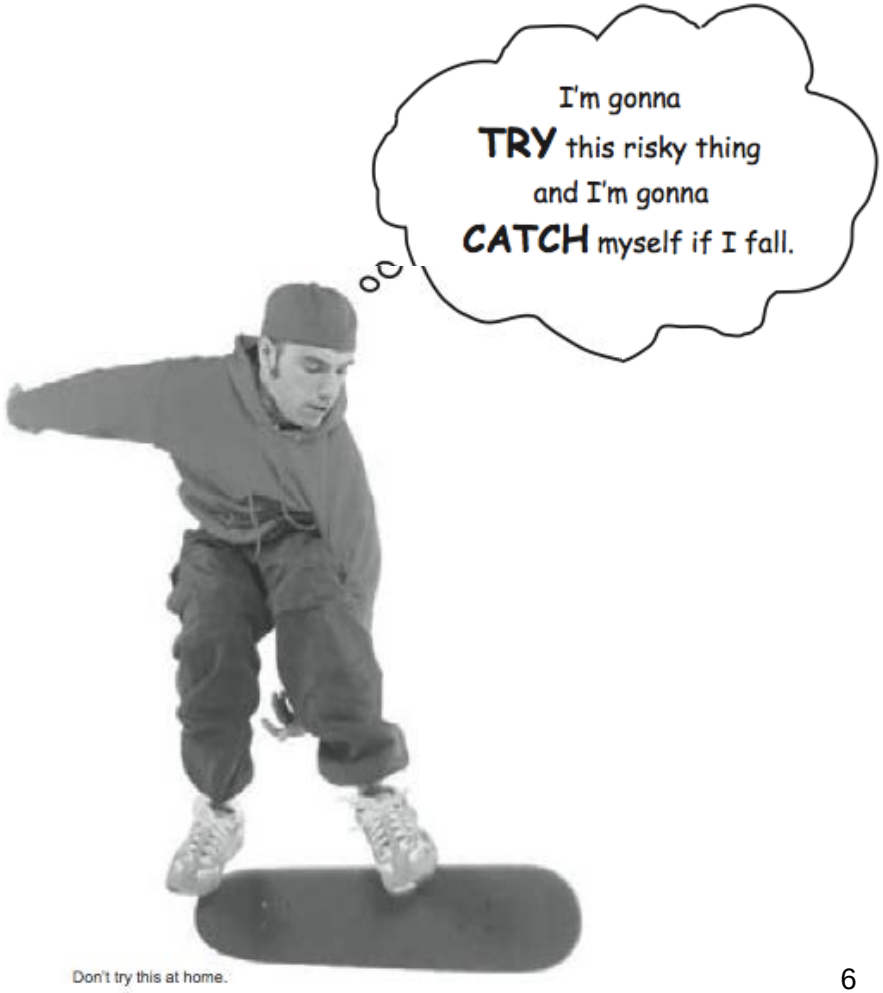
# Exception – is an object… of type Exception

```
try {
    // do risky thing

} catch(Exception ex) {
    // try to recover
}
```

it's just like declaring a method argument.

This code only runs if an Exception is thrown.




**How to recover?**

- If the server does not respond, you can use catch block to try again or connect to another server
- If file is not found, you can ask user to help to find it
- If you cannot fix it, you should inform user/admin/developer about it

# If it's your code that catches the exception, then whose code throws it?

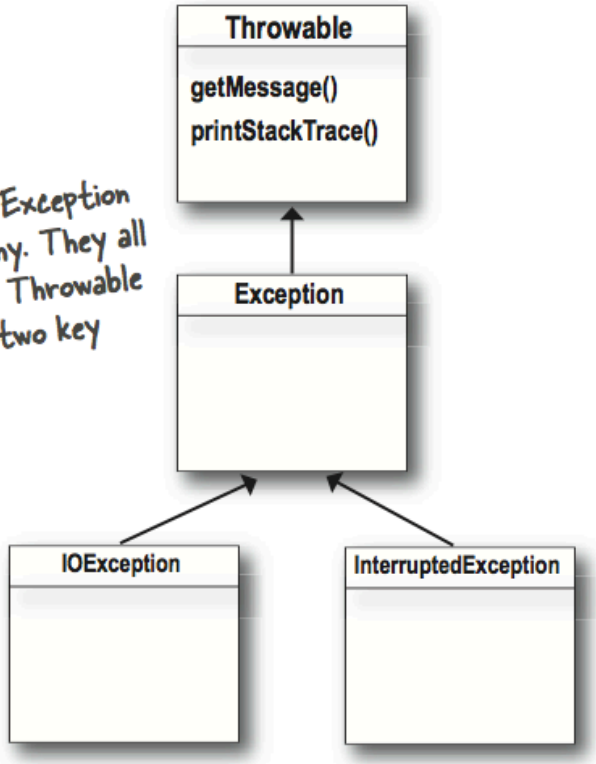
**① Risky, exception-throwing code:**

this method MUST tell the world (by declaring) that it throws a BadException

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

create a new Exception object and throw it.

Part of the Exception class hierarchy. They all extend class Throwable and inherit two key methods.

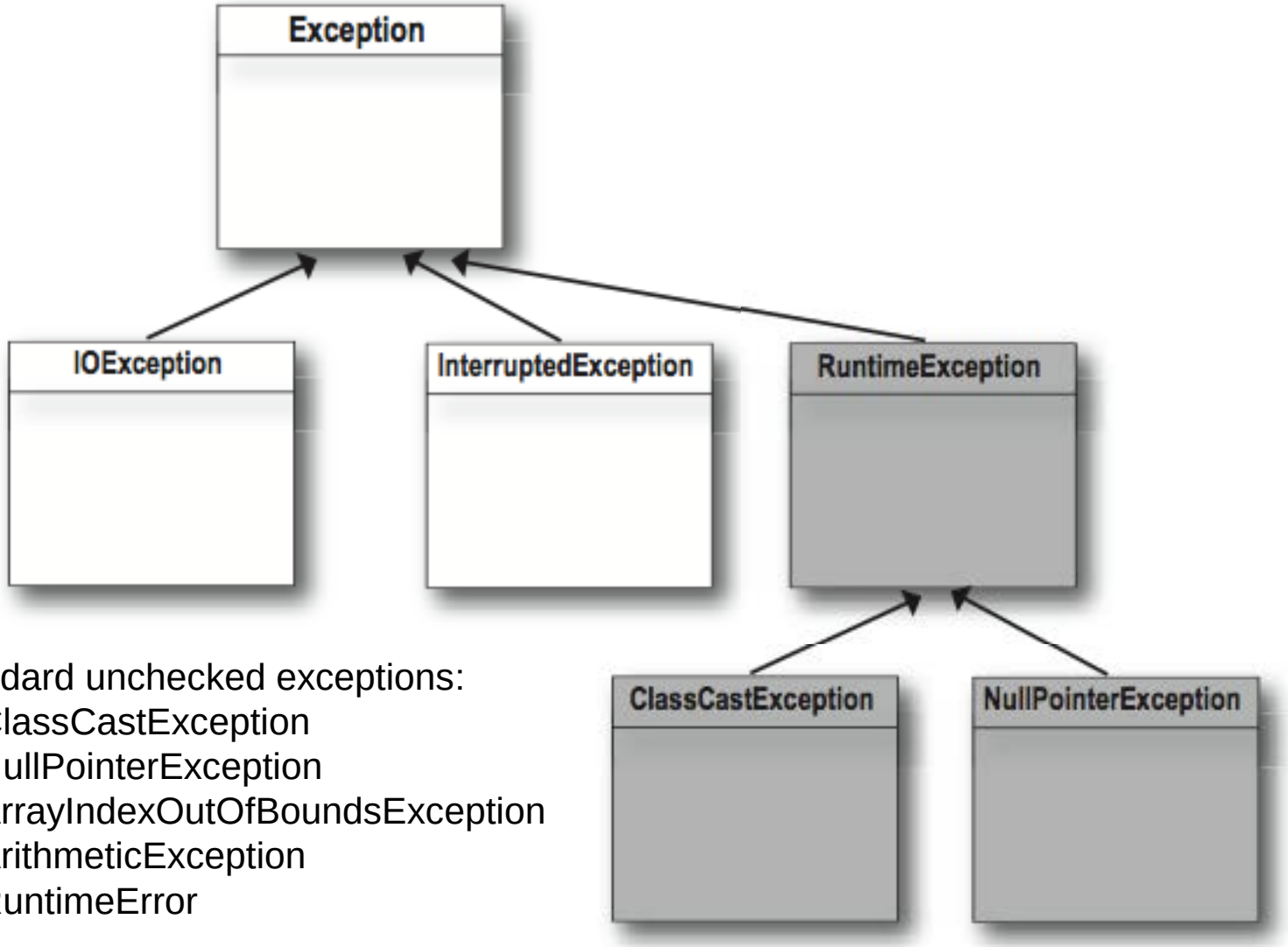



**② Your code that *calls* the risky method:**

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

If you can't recover from the exception, at LEAST get a stack trace using the printStackTrace() method that all exceptions inherit.

# The compiler checks for everything except RuntimeExceptions




Standard unchecked exceptions:

- ClassCastException
- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException
- RuntimeError

# Standard unchecked exceptions

ArrayIndexOutOfBoundsException

Get out of array bounds.

```
int[] array = new int[10];
array[20] = 0; // ArrayIndexOutOfBoundsException will be thrown
```

**How to avoid?** Check index before use.

ArithmeticException

Arithmetic error, for example division by zero.

```
int a = 10/0; // ArithmeticException will be thrown
```

**How to avoid?** Check if the divider is 0.

ClassCastException

Error of type casting:

```
int x = toInt("I am a String");

int toInt(Object o) {
    Integer i = (Integer) o; // ClassCastException will be thrown
    return i.intValue();
}
```

**How to avoid?** Check the type on casting by using **instanceof**:

```
if (o instanceof Integer) {
    Integer i = (Integer) o;
}
```

# Standard unchecked exceptions

NullPointerException

Error when try to access field with null value:

```
Bar bar = null;
bar.foo(); // NullPointerException will be thrown
```

**How to avoid?** Check variable on null before use.

IllegalArgumentException

Standard exception used when wrong arguments were used

```
public class MyBadCode {
  public static void main(String[] args) {
    Percentage percentage = new Percentage(121);
    System.out.println(percentage.getValue());
  }
}
```

```
Exception in thread "main"
java.lang.IllegalArgumentException:
121
  at Percentage.(Percentage.java:12)
```

Code which throws IllegalArgumentException:

```
public Percentage(int value) {
  if (value < 0 || value > 100) {
    throw new IllegalArgumentException(Integer.toString(value));
  }
  this.value = value;
}
```

# Call stack and the exceptions

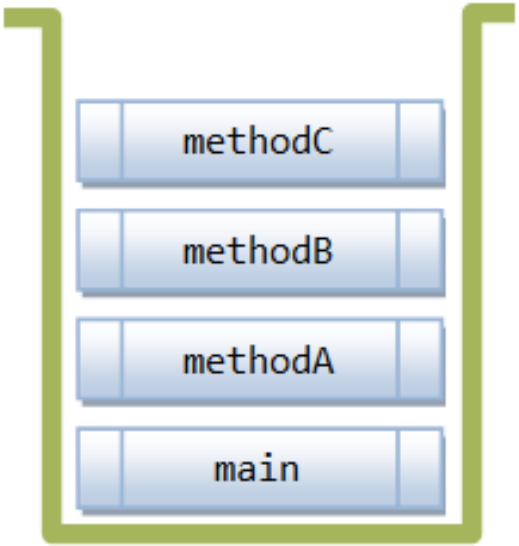
```java
public class MethodCallStackDemo {
    public static void main(String[] args) {
        System.out.println("Enter main()");
        methodA();
        System.out.println("Exit main()");
    }

    public static void methodA() {
        System.out.println("Enter methodA()");
        methodB();
        System.out.println("Exit methodA()");
    }

    public static void methodB() {
        System.out.println("Enter methodB()");
        methodC();
        System.out.println("Exit methodB()");
    }

    public static void methodC() {
        System.out.println("Enter methodC()");
        System.out.println("Exit methodC()");
    }
}
```

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exit methodC()
Exit methodB()
Exit methodA()
Exit main()
```




**Method Call Stack**
**(Last-in-First-out Queue)**

# Call stack and the exceptions

methodC() throws ArithmeticException:

```
public static void methodC() {
   System.out.println("Enter methodC()");
   System.out.println(1 / 0);  // divide-by-0 triggers an ArithmeticException
   System.out.println("Exit methodC()");
}
```

Result of the program execution will be as follows:

```
Enter main()
Enter methodA()
Enter methodB()
Enter methodC()
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MethodCallStackDemo.methodC(MethodCallStackDemo.java:22)
        at MethodCallStackDemo.methodB(MethodCallStackDemo.java:16)
        at MethodCallStackDemo.methodA(MethodCallStackDemo.java:10)
        at MethodCallStackDemo.main(MethodCallStackDemo.java:4)
```

This is a execution stack or call stack.
This is a default behavior of exception.**printStackTrace()**

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

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

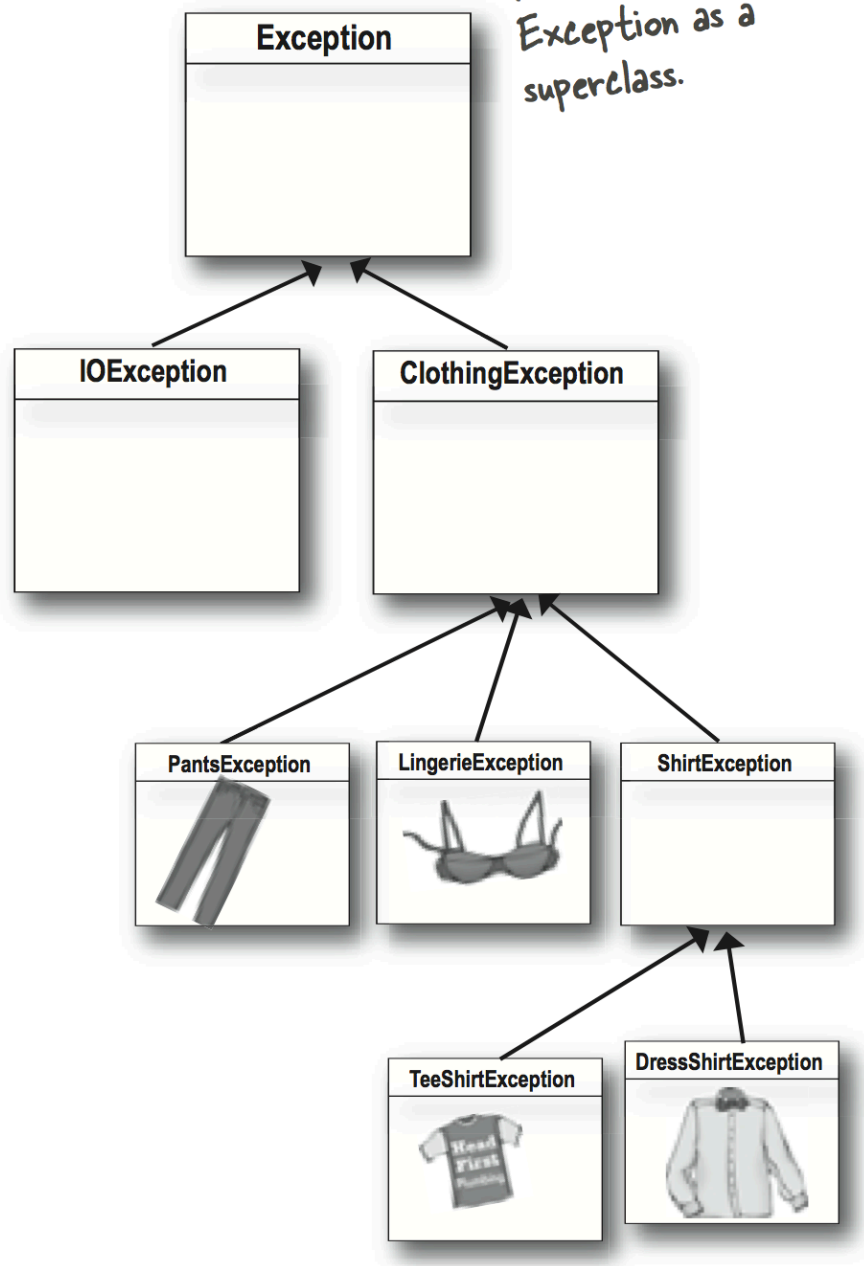

**If try block fails**
- control immediately moves to catch {}
- finally {} block runs

**try block succeeds (*no* exception)?**
- finally {} block runs

**try or catch has return?**
- finally {} block runs anyway!

# Exceptions are polymorphic

All exceptions have Exception as a superclass.




① **You can DECLARE exceptions using a supertype of the exceptions you throw.**

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

Declaring a ClothingException lets you throw any subclass of ClothingException. That means doLaundry() can throw a PantsException, LingerieException, TeeShirtException, and DressShirtException without explicitly declaring them individually.

```
{
```

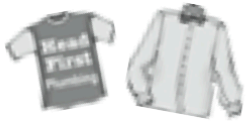
## ② You can CATCH exceptions using a supertype of the exception thrown.

```
try {
    laundry.doLaundry();
```

can catch any ClothingException subclass

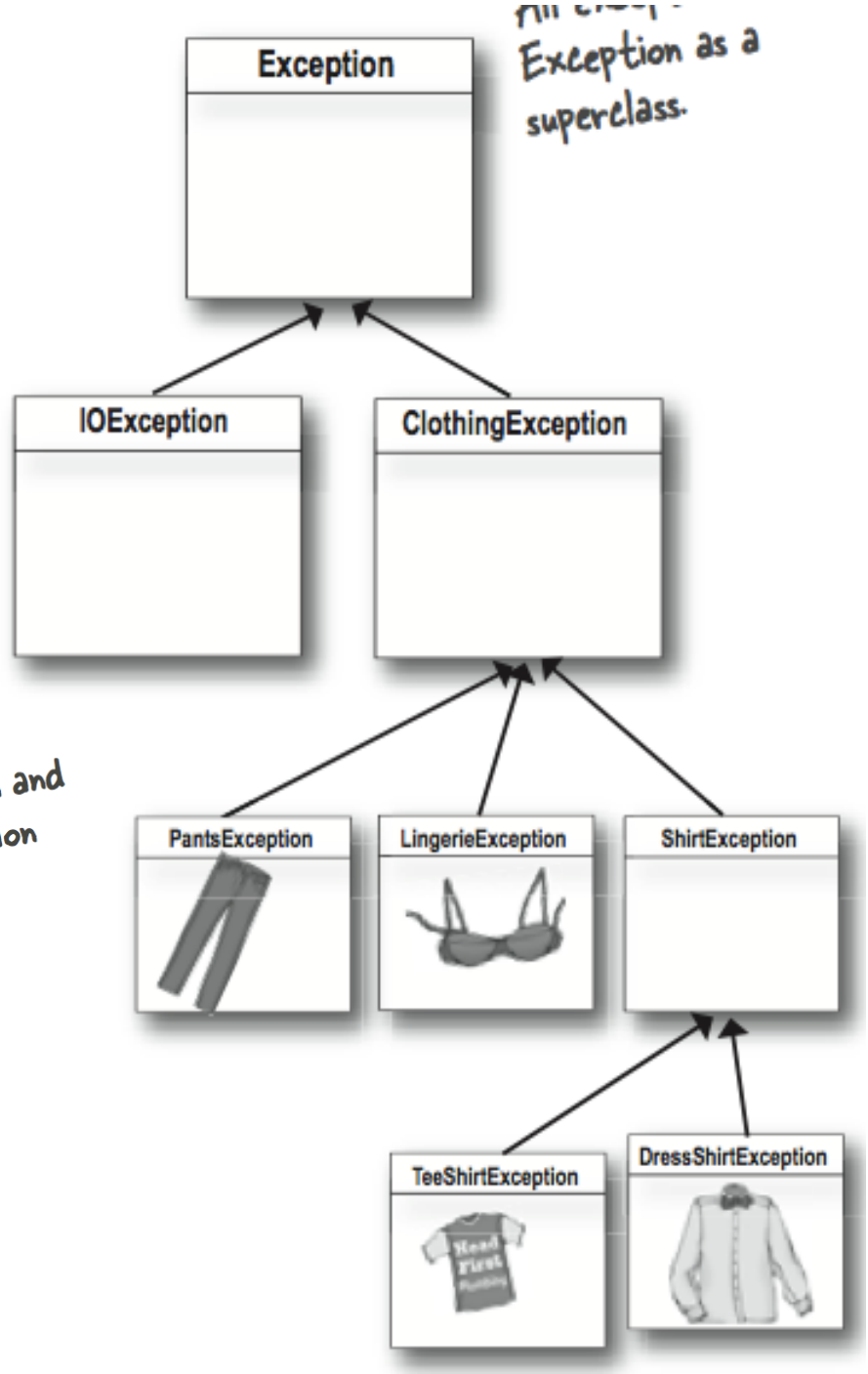
```
} catch(ClothingException cex) {
    // recovery code
}
```

```
try {
    laundry.doLaundry();
```



can catch only TeeShirtException and DressShirtException

```
} catch(ShirtException sex) {
    // recovery code
}
```



# Exceptions are polymorphic

**Write a different catch block for each exception that you need to handle uniquely.**

```
try {
    laundry.doLaundry();
```

```
} catch(TeeShirtException tex) {
    // recovery from TeeShirtException
```

```
} catch(LingerieException lex) {
    // recovery from LingerieException
```

```
} catch(ClothingException cex) {
    // recovery from all others
}
```

TeeShirtExceptions and LingerieExceptions need different recovery code, so you should use different catch blocks.

All other ClothingExceptions are caught here.

# Multiple catch blocks must be ordered from smallest to biggest


TeeShirtExceptions are caught here, but no other exceptions will fit.

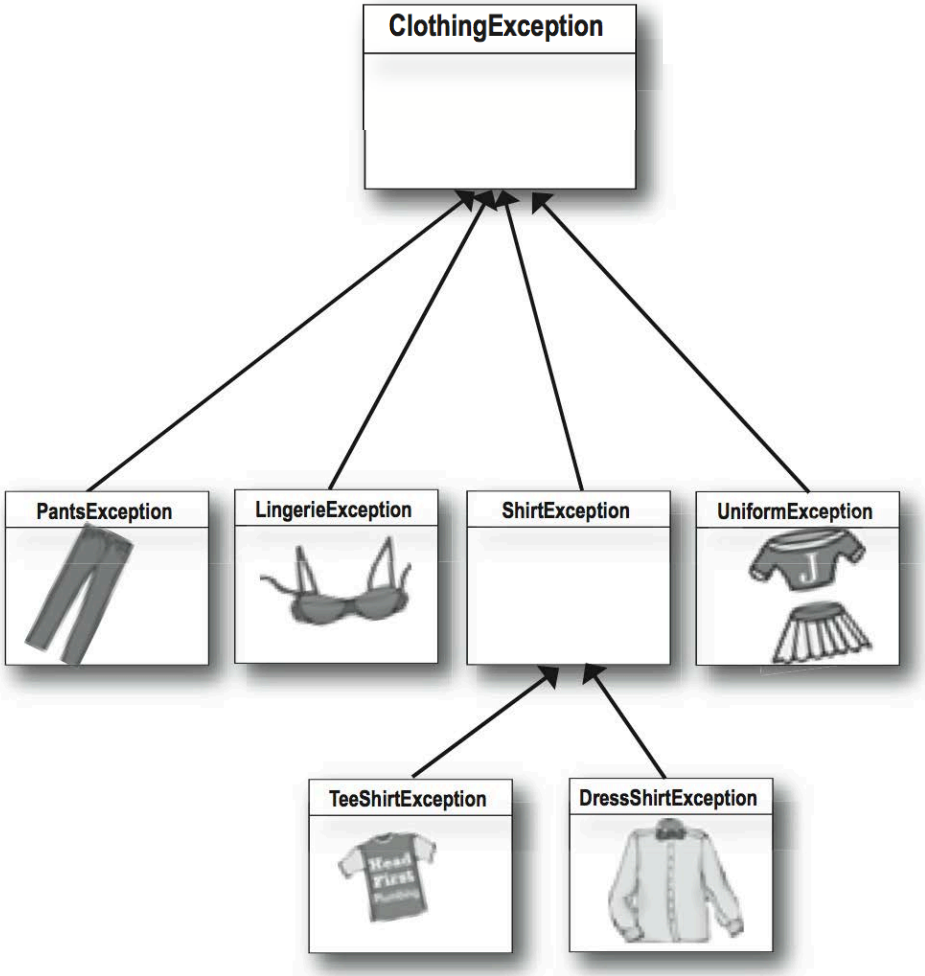
```
catch(TeeShirtException tex)
```

TeeShirtExceptions will never get here, but all other ShirtException subclasses are caught here.

```
catch(ShirtException sex)
```

All ClothingExceptions are caught here, although TeeShirtException and ShirtException will never get this far.

```
catch(ClothingException cex)
```



## Sooner or later, *somebody* has to deal with it. But what if *main()* ducks the exception?

```
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();
    }
}
```

Both methods duck the exception (by declaring it) so there's nobody to handle it! This compiles just fine.

**1** doLaundry() throws a ClothingException


main() calls foo()

foo() calls doLaundry()

doLaundry() is running and throws a ClothingException

**2** foo() ducks the exception


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

But foo() doesn't have a try/catch, so...

**3** main() ducks the exception


foo() pops off the stack immediately and the exception is thrown back to... who? What? There's nobody left but the JVM, and it's thinking, "Don't expect ME to get you out of this."

**4** The JVM shuts down

# **Checked exceptions**: Handle or Declare.

## ① HANDLE

Wrap the risky call in a try/catch

```
try {
    laundry.doLaundry();
} catch(ClothingException cex) {
      // recovery code
}
```

This had better be a big enough catch to handle all exceptions that doLaundry() might throw. Or else the compiler will still complain that you're not catching all of the exceptions.

## ② DECLARE (duck it)

Declare that YOUR method throws the same exceptions as the risky method you're calling.

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

The doLaundry() method throws a ClothingException, but by declaring the exception, the foo() method gets to duck the exception. No try/catch.

# Work with resources: finally

```
InputStream input = null;

try{

  input = new FileInputStream(”file.txt");

  // do something with the stream

} catch(IOException e){ // first catch block
  throw new WrapperException(e);
} finally {
  try {
     if(input != null) input.close();
  } catch(IOException e) {  // second catch block
     throw new WrapperException(e);
  }
}
```

# Work with resources: try-with-resources

```
private static void printFileJava7() throws IOException {

   try(FileInputStream input = new FileInputStream("file.txt")){

      int data = input.read();
      while(data != -1){
         System.out.print((char) data);
         data = input.read();
      }
   }
}
```

# Work with resources: try-with-resources multiple

```
private static void printFileJava7() throws IOException {

    try(  FileInputStream input = new FileInputStream("file.txt");
          BufferedInputStream bufferedInput = new BufferedInputStream(input)
    ) {

        int data = bufferedInput.read();
        while(data != -1){
            System.out.print((char) data);
            data = bufferedInput.read();
        }
    }
}
```

# Work with resources: AutoClosable

```
public interface AutoClosable {
    public void close() throws Exception;
}

public class MyAutoClosable implements AutoCloseable {
    public void doIt() {
        System.out.println("MyAutoClosable doing it!");
    }

    @Override
    public void close() throws Exception {
        System.out.println("MyAutoClosable closed!");
    }
}

private static void myAutoClosable() throws Exception {
    try(MyAutoClosable myAutoClosable = new MyAutoClosable()){
        myAutoClosable.doIt();
    }
}
```

```
MyAutoClosable doing it!
MyAutoClosable closed!
```

# Catching Multiple Exceptions in Java 7

Before Java 7

```
try {
   // execute code that may throw 1
     // of the 3 exceptions below.

} catch(SQLException e) {
   logger.log(e);

} catch(IOException e) {
   logger.log(e);

} catch(Exception e) {
   logger.severe(e);
}
```

Java 7+

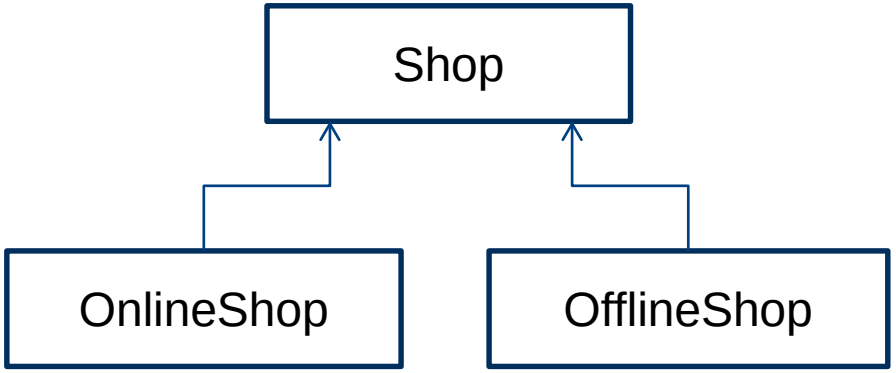
```
try {
  // execute code that may throw
  //1 of the 3 exceptions below.

} catch(SQLException |
          IOException e) {
   logger.log(e);

} catch(Exception e) {
   logger.severe(e);
}
```

# Exceptions and inheritance

The inherited class cannot extend the list of thrown exceptions by a method:




```
class Shop {
    public float getBalance();
    public static Shop createShop()
    {
        return new OnlineShop();
    }
}

class OnlineShop {
    public float getBalance()
        throws ServerException;
        // WRONG!
}
```

```
Shop shop = Shop.createShop(); // creating OnlineShop or OfflineShop
shop.getBalance(); // we rely on the safe code
```

But if shop is an instance of **OnlineShop**, then **getBalance()** is already dangerous.

# Exceptions and try-catch area

*Assume that client has no money*
*Exception* **NotEnoughFundsException** will be thrown.
*What is wrong here?*

```
try {
    Client client = createClient();
    Account account = client.createAccount();
    client.withdraw(10000);
    client.deposit(1000);
    client.withdraw(500);
} catch (NotEnoughFundsException e) {
    e.printStackTrace();
}
```

# Exceptions and try-catch area

*What is wrong here?*

```
try {
    Client client = createClient();
    Account account = client.createAccount();
    client.withdraw(10000);
    client.deposit(1000);
    client.withdraw(500);
} catch (NotEnoughFundsException e) {
    e.printStackTrace();
}
```

*Remaining operations are not executed*

```
try {
    client.withdraw(10000);
} catch (NotEnoughFundsException e) {
    e.printStackTrace();
}
try {
    client.deposit(1000);
} catch (NotEnoughFundsException e) {
    e.printStackTrace();
}
```

*Logically related set of operations should be in its own try… catch block*

# Exceptions and try-catch area

*What is wrong with this code?*
Assume that John Smith exists.

```
try {

    Client client1 = createClient("John Smith");
    Account account = client.createAccount();
    client.deposit(1000);
    client.withdraw(500);

    Client client2 = createClient("Jane Brown");
    Account account = client.createAccount();
    client.deposit(1000);

} catch (ClientExistsException e) {
    e.printStackTrace();
}
```

# Exercise

Lab guide:

- Exercise 16