Java SE 7

Module 4
Inner Classes & Exception Handling

LUXOFT TRAINING

Contents

1. Inner class

2. Static inner class

3. Anonymous and local inner classes

4. Exception handling

Inner class

You can declare a class in any block including blocks that are part of a method.

Inner class

```
public class Dog
{
    private boolean isAngry;

    public void bark()
    {
        if (isAngry)
        {
            class SecretPartOfTheBrain
            {
                private String theThoughts = "No, barking is not enough this time...";

                public void action()
                {
                    // ...
                }
            }
            new SecretPartOfTheBrain().action();
        }
    }
}
```

Inner class

After compilation, a separate file created with the name generated according to the next template:

OuterClassName$InnerClassName.class

The full name of the inner class will look like this:

packageName.OuterClassName.InnerClassName

Inner class

Inner classes can only be instantiated through an outer class instance.

```
public class Outer
{
    public class Inner {}

    public static void main(String[] args)
    {
        Inner instance = new Outer().new Inner();
    }
}
```

Can be **public**, **private**, **protected** or with default access.

Inner class

Inner classes instance has access to all the data of enclosing type including **private**.

```
public class Outer
{
    private int data;

    class Inner
    {
        public int calculate()
        {
            return (data + data) * 2;
        }
    }
}
```

Inner class

Why do we need a class like that?

private inner class

Sometimes we need **data structures** that are very important **inside** the class but **meaningless outside** it.

public inner class

Sometimes we need a **data structure** that provides controllable access to the instance private data.

static inner class

Sometimes we need independent **internal data structure** that can be accessed from the outer world.

static inner class

- Inner class can be declared as **static**.
- A static nested class cannot use the **this** keyword to access outer object attributes.
- Yet, it can request static variables and static outer class methods.

Local inner class

Also, there is an option, that maybe nobody will find the place where to implement.

Local inner class

- Anything declared within a method is not a class member.

- Local objects cannot have access modifiers and cannot be declared as **static**.

Anonymous inner class

inners.i3anonymous

Sometimes we need an instance of the interface…

Anonymous inner class

- You can declare an inner class within the body of a method without naming it.

- Can be declared as extension to another class or as an interface implementation.

- A constructor cannot be defined for an anonymous class.

- The superclass constructor can be called.

Anonymous inner class

- Practical when you do not want to use trivial names for classes.

- The class code contains several lines.

- When compiling an anonymous class, a separate class named **EnclosingClassName$n** is created, where **n** is an anonymous class order number in the outer class.

Anonymous inner class

inners.i3anonymous.Printer

For **local** and **anonymous** classes you can only access outer variables if they are declared as **final**.

```
public static void print(String data, int times)
{
    final String fData = data;

    for (int i = 0; i < times; i++)
    {
        new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                System.out.println(fData);
            }
        }).start();
    }
}
```

Inner classes

inners.i5animals

```java
List<Pet> pets = new ArrayList<>();

// create anonymous class inherited from Cat
pets.add(new Cat("Tiger")
{
    public String getName() { return ""; }

    public void beFriendly()
    {
        System.out.println("I'm Tiger, not friendly!");
    }
});

// adding Pet interface implementation
pets.add(new Pet()
{
    public String getName() { return "I'm a Pet"; }

    public void beFriendly() { }
});
```

```java
public interface Pet
{
    String getName();
    void beFriendly();
}

public class Cat implements Pet {
    private String name;

    public Cat(String name) {
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

    public void beFriendly() {
        System.out.println(name
            + ": I'm friendly!");
    }
}
```

Inner classes

inners.i5animals

```
pets.add(new RoboDog("Robik"));

// create inner class
pets.add(new SpecialRoboDog());

// create anonymous class inherited from RoboDog
pets.add(new RoboDog()
{
    public void beFriendly()
    {
        System.out.println(getName()
            + ": I'm more friendly then everyone else!");
    }
});

// ask all pets to be friendly
for (Pet pet : pets) { pet.beFriendly(); }

static class SpecialRoboDog extends RoboDog
{
    public void beFriendly()
    {
        System.out.println(getName()
            + ": I'm very special for you!");
    }
}
```

```
public class RoboDog extends Robot
    implements Pet
{
    private String name;

    public RoboDog()
    {
        this("Noname Robodog");
    }

   @Override
    public String getName()
    {
        return name;
    }

    @Override
    public void beFriendly()
    {
        System.out.println(name
            + ": I'm friendly!");
    }
}
```

Inner classes and Java 8

inners.java8.AnonymousPrinter

For local and anonymous classes you can only access outer variables if they are **effectively final** (not final, but never changed).

```
public static void print(String data, int times)
{
    for (int i = 0; i < times; i++)
    {
        new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                System.out.println(data);
            }
        }).start();
    }
}
```

Inner classes and Java 8

inners.java8.LambdaPrinter

An anonymous class can be replaced with a **lambda expression**.

```
public static void print(String data, int times)
{
    for (int i = 0; i < times; i++)
    {
        new Thread(() -> System.out.println(data)).start();
    }
}
```

Note: lambda expression is a new dynamic type in Java.

Inner Classes

- Exercise 1 - Dog
- Exercise 2 - Bank Application

Contents

1. Inner class

2. Static inner class

3. Anonymous and local inner classes

4. **Exception handling**

Unsafe Code

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

Option #1 Use error code and if blocks:

Method must return:

1. Result of the execution
2. Success status

```
FileManager manager = new DefaultFileManager();
boolean opened = manager.openFile();

if (opened)
{
    if (manager.readFile())
    {
        //...
        if (!manager.closeFile()) { System.out.println("Can't close the file."); }
    }
    else { System.out.println("Can't read from file."); }
}
else { System.out.println("Can't open the file."); }
```

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

Unsafe Code: use exceptions

Option #2 Let FileManager methods may throw exceptions:

```
public interface FileManager
{
    boolean openFile() throws FileNotFoundException;

    boolean readFile() throws IOException;

    boolean closeFile() throws FileCloseException;
}
```

Unsafe Code: use exceptions

Option #2 Let FileManager methods may throw exceptions:

```
FileManager manager = new DefaultFileManager();
try
{
    manager.openFile();
    manager.readFile();
    manager.closeFile();
}
catch (FileNotFoundException e) { System.out.println("Can't open the file."); }

catch (IOException e) { System.out.println("Can't read from file."); }

catch (FileCloseException e) { System.out.println("Can't close the file."); }
```

Safe block

Exception handlers

Advantages:

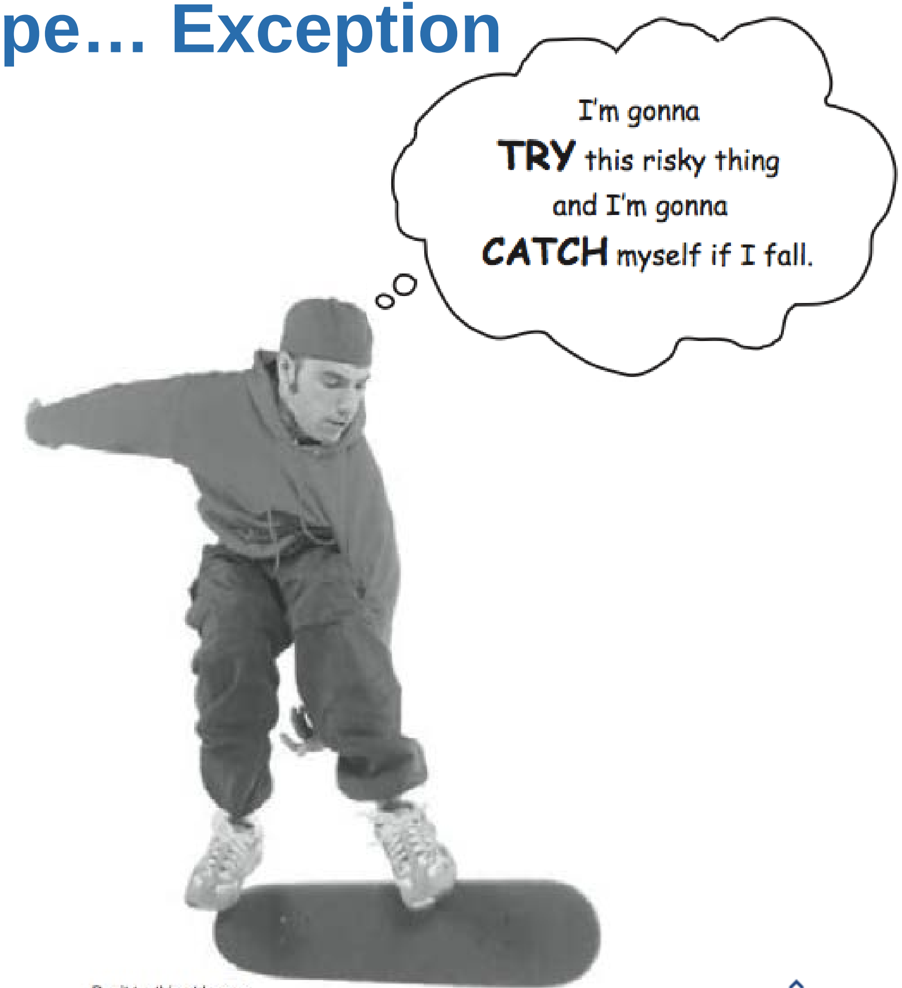
- We can concentrate on code and do not think about exceptions.
- Handling of all unsafe situations is placed to the single block.

Exception - is an object of type… Exception

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

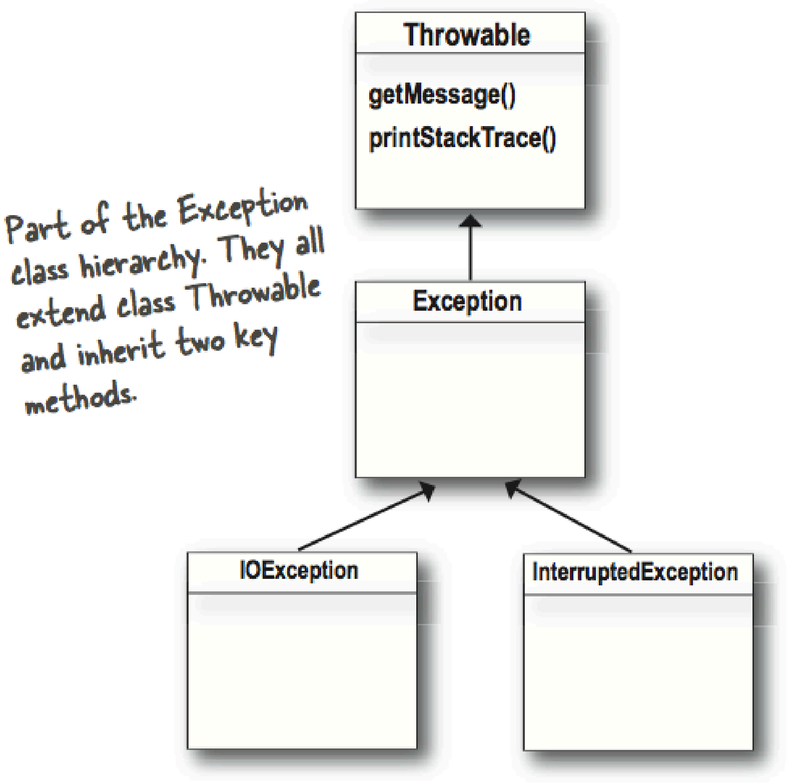
How to recover?

- 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.



Don't try this at home.

Exception hierarchy


Throwable
getMessage()
printStackTrace()
Exception
IOException
InterruptedException
Part of the Exception class hierarchy. They all extend class Throwable and inherit two key methods.

Exception is a class

```
public class PersonNotFoundException extends RuntimeException { }
```

And we can use it this way:

```
Person person = personsHolder.find(name);

if (person == null)
{
    throw new PersonNotFoundException();
}
return person;
```

Now code to work with person will be like this:

```
try
{
    Person person = findPerson("John Smith");
    person.sendMessage("Hello John");
}
catch (PersonNotFoundException e)
{
    System.out.println("Person not found.");
}
```

Exception with parameters

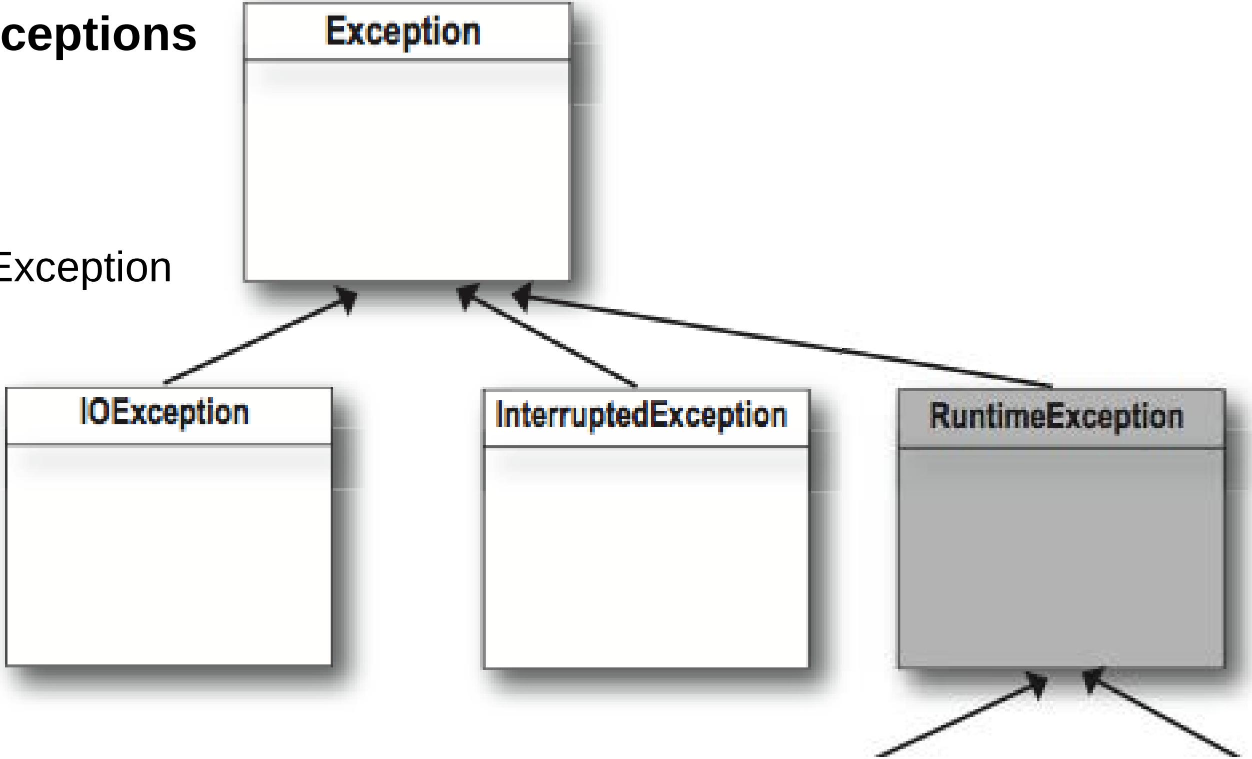
```
public class PersonNotFoundException extends RuntimeException
{
    private String name;

    public String getName() { return name; }

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

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

The compiler checks for everything except RuntimeException

Standard unchecked exceptions

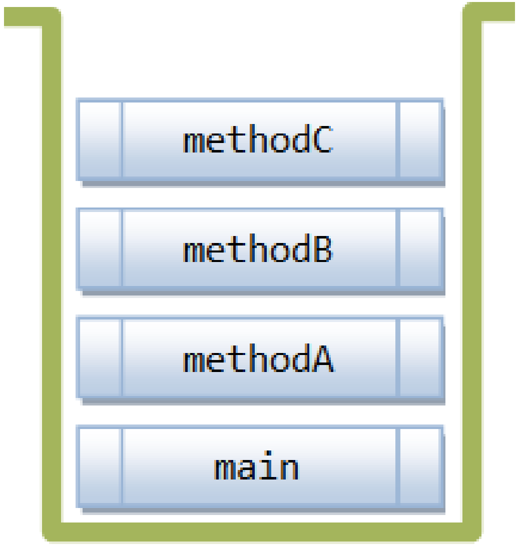
- ClassCastException
- NullPointerException
- ArrayIndexOutOfBoundsException
- ArithmeticException
- RuntimeError



Call stack and the Exceptions

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

```
public class MethodCallStackDemo
{
    public static void main(String[] args)
    {
        methodA();
    }

    public static void methodA()
    {
        methodB();
    }

    public static void methodB()
    {
        methodC();
    }

    public static void methodC()
    {

    }
}
```



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

Call stack and the Exceptions

```
public static void methodC()
{
    System.out.println(1 / 0); // this line triggers an ArithmeticException
}
```

```
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 (Exception e) { e.printStackTrace(); }

finally { turnOvenOff(); }
```

Are you sure you want to try this?
No matter what, do NOT let me forget to turn off the oven! Last time I torched half the neighborhood.

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?

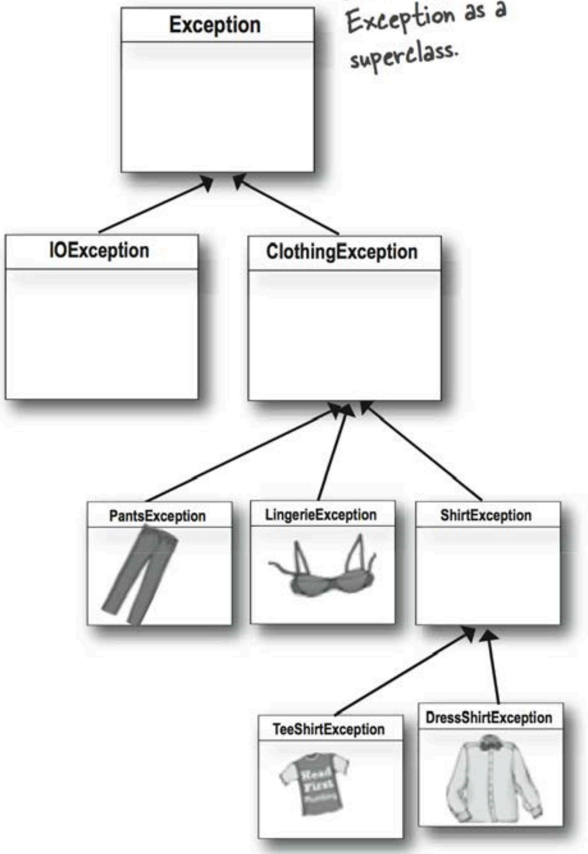
finally {} block runs anyway!

Exceptions are polymorphic

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.



All exceptions have Exception as a superclass.

{

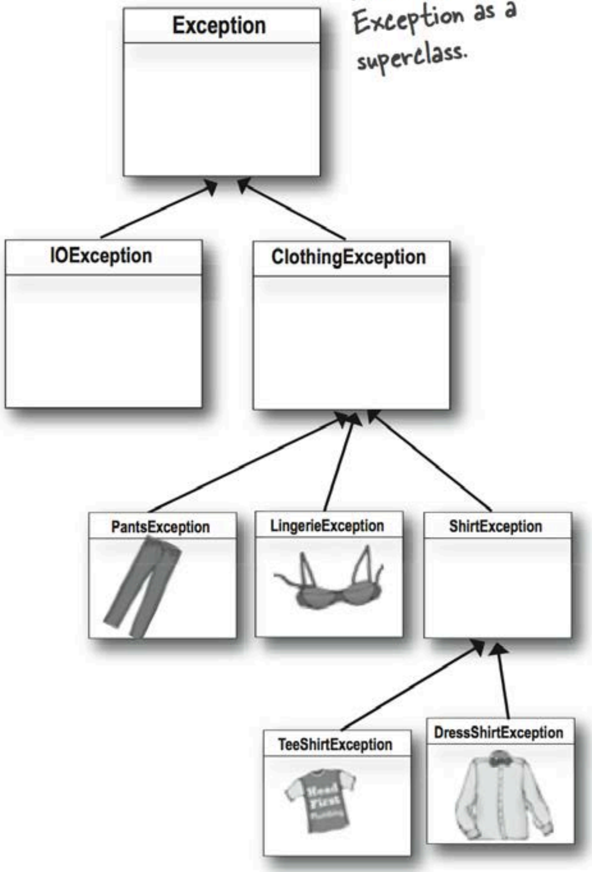
② You can CATCH exceptions using a supertype of the exception thrown.

```
try {

   laundry.doLaundry();

} catch(ClothingException cex) {

     // recovery code

}
```

can catch any ClothingException subclass

```
try {

   laundry.doLaundry();

} catch(ShirtException sex) {

     // recovery code

}
```

can catch only TeeShirtException and DressShirtException



All exceptions have Exception as a superclass.

{

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

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

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

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

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

All other ClothingExceptions are caught here.

Multiple catch blocks must be ordered according to class hierarchy

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

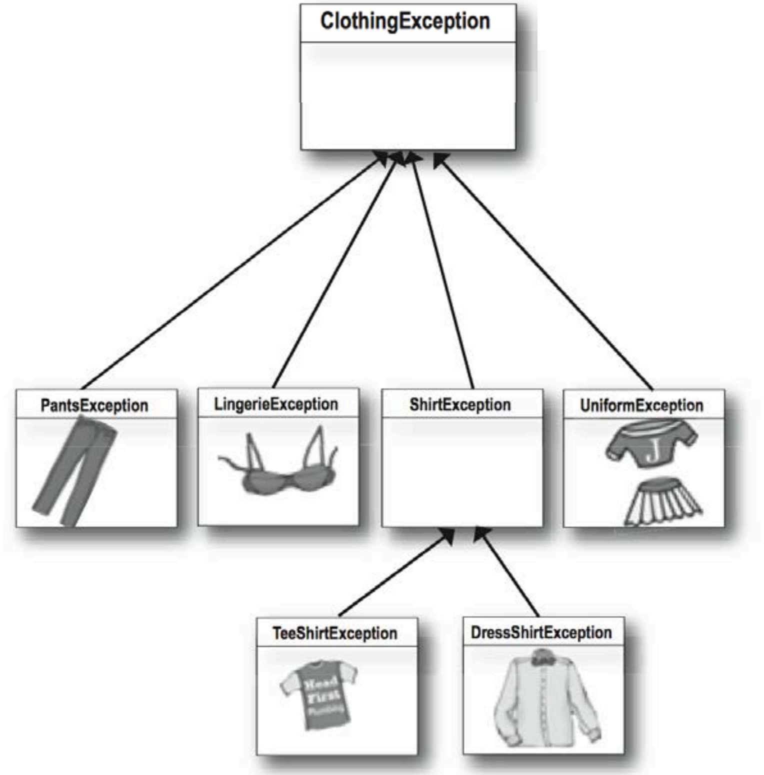
```
catch(TeeShirtException tex)
```

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

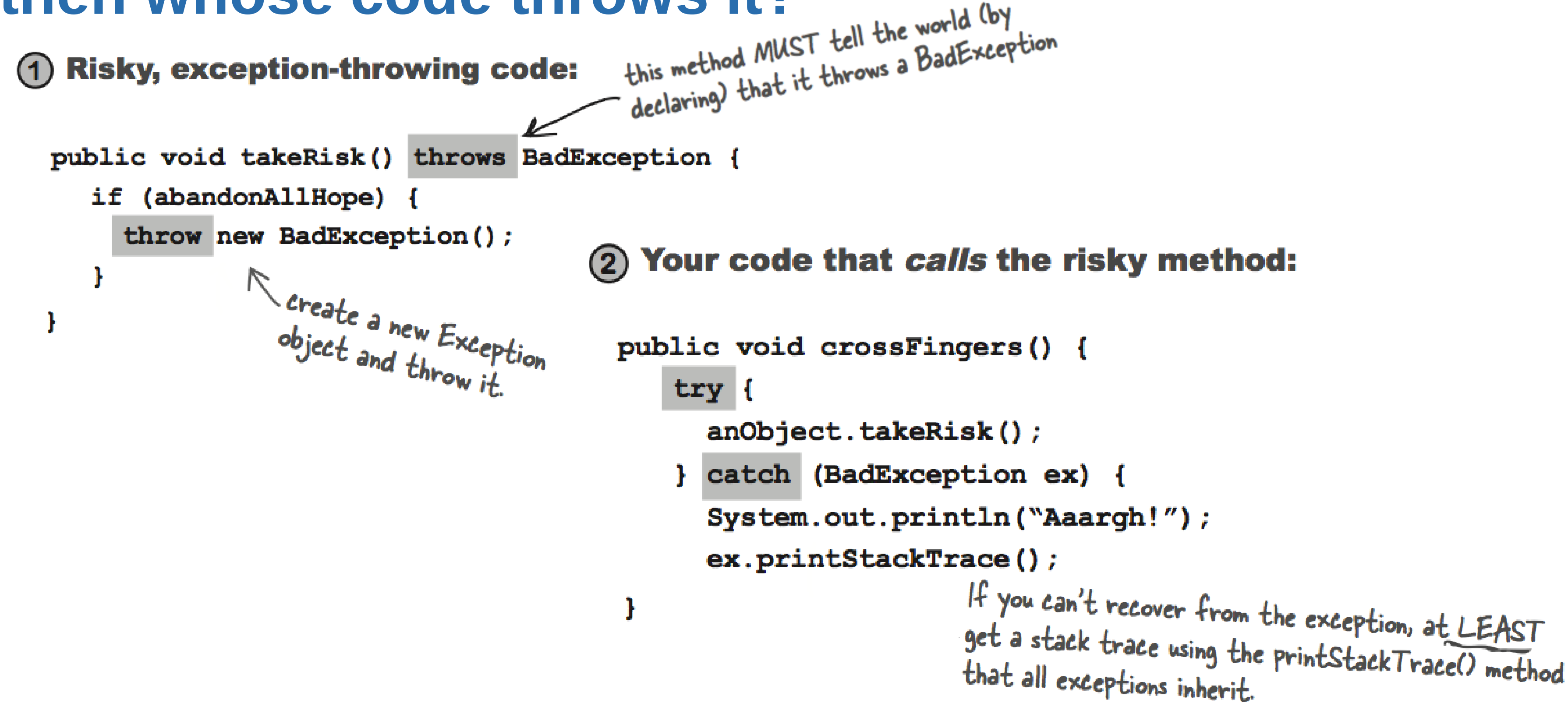
```
catch(ShirtException sex)
```

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

```
catch(ClothingException cex)
```



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();
   }
}
```

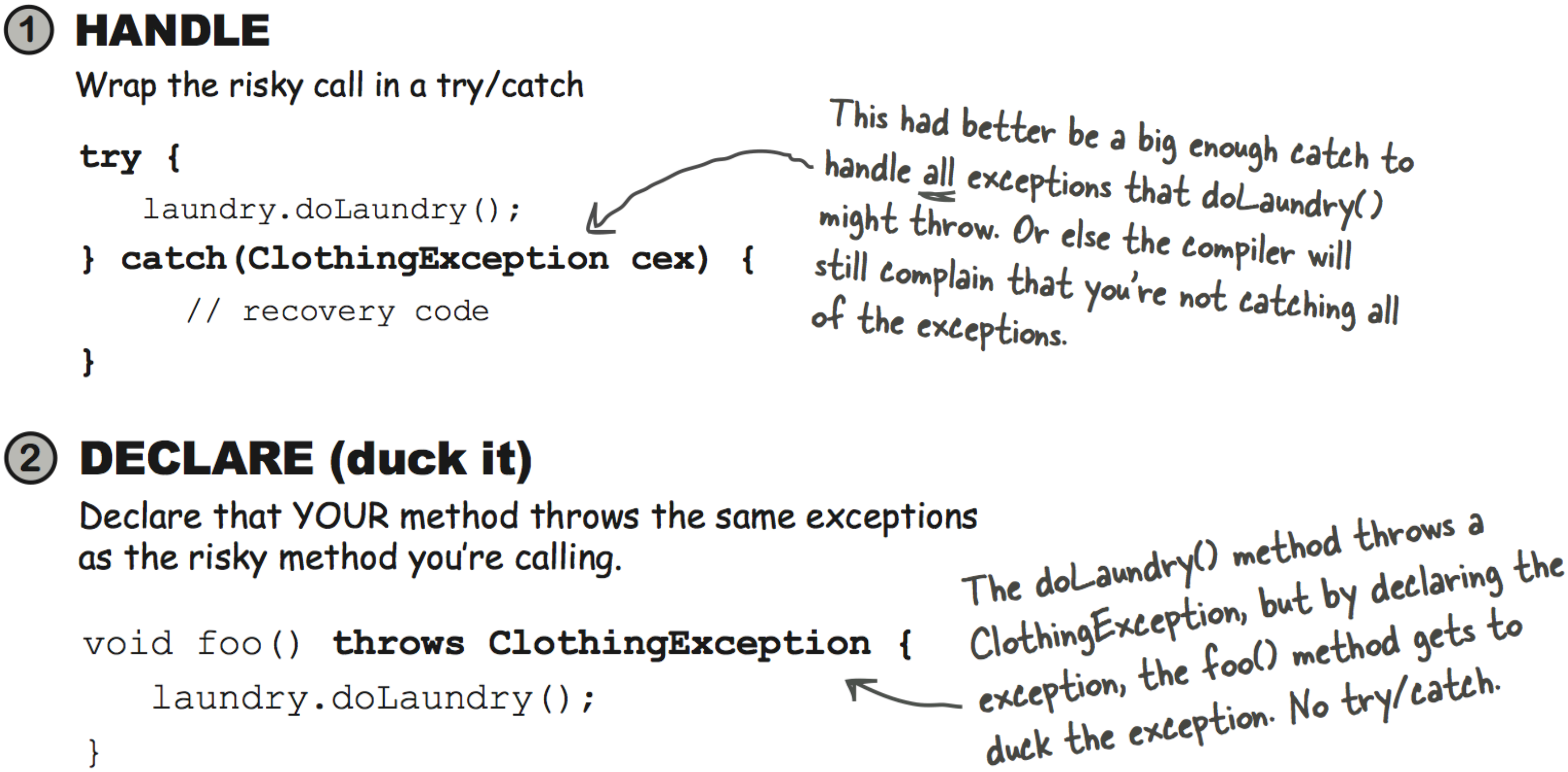
create a new Exception object and throw it.

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.

Checked exceptions: Handle || 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();
}
```

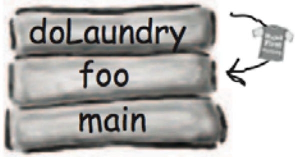
The doLaundry() method throws a ClothingException, but by declaring the exception, the foo() method gets to duck the exception. No try/catch.

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

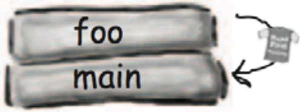


main() calls foo()

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

Work with resources: Java 6

```
InputStream in = null;

try
{
    in = new FileInputStream("file.txt");
}
catch (IOException e) { // try to recover }

finally
{
    try
    {
        if (in != null)
        {
            in.close();
        }
    }
    catch (IOException e) { // try to recover }
}
```

Work with resources: Java 7

```java
try (InputStream in = new FileInputStream("file.txt"))
{
    int data = in.read();
    // ...
}
catch (IOException e)
{
    throw new UncheckedIOException(e);
}
```

Work with multiple resources: Java 7

```
try
(
    InputStream in = new FileInputStream("file.txt");
    BufferedInputStream buffer = new BufferedInputStream(in)
){
    int data = buffer.read();
    // ...
}
catch (IOException e)
{
    // try to recover
}
```

Work with resources: AutoClosable

```
/**
 * An object that may hold resources (such as file or socket handles)
 * until it is closed...
 *
 * @author Josh Bloch
 * @since 1.7
 */
public interface AutoCloseable
{
    void close() throws Exception;
}
```

Catching multiple exceptions: Java 7

```
try
{
    // ...
}
catch (SQLException | IOException e)
{
    log(e);
}
```

Exceptions and inheritance

```
public interface I
{
    void i();
}

public class A
{
    public void a() { };
}

public class B extends A implements I
{
    public void i() throws Exception {}; // will not compile

    @Override
    public void a() throws Exception {}; // will not compile
}
```

Exception Handling

- Exercise 3 - Exceptions