

異常

An example

You make a beatbox loop (a 16-beat drum pattern) by putting checkmarks in the boxes.

Cyber BeatBox

Bass Drum	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Closed Hi-Hat	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Open Hi-Hat	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Acoustic Snare	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Crash Cymbal	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Hand Clap	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
High Tom	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Hi Bongo	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Maracas	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Whistle	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Low Conga	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Cowbell	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Vibraslap	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Low-mid Tom	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
High Agogo	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Open Hi Conga	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

Start
Stop
Tempo Up
Tempo Down
sendit

dance beat

Andy: groove #2
Chris: groove2 revised
Nigel: dance beat

your message, that gets sent to the other players, along with your current beat pattern, when you hit "SendIt"

incoming messages from other players. Click one to load the pattern that goes with it, and then click 'Start' to play it.

the JavaSound API

- JavaSound is a collection of classes and interfaces added to Java starting with version 1.3. These aren't special add-ons , they're part of the standard j2SE class library.
- MIDI stands for Musical instrument Digital Interface, and is a standard protocol for getting different kinds of electronic sound equipment to communicate.

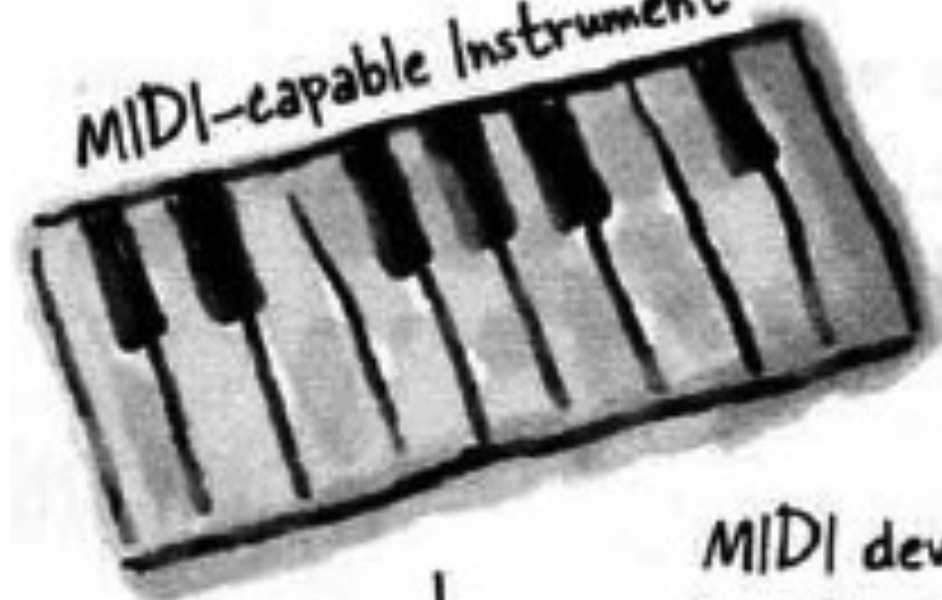
MIDI file

play high C,
hit it hard
and hold it
for 2 beats

MIDI file has information
about how a song should be
played, but it doesn't have any
actual sound data. It's kind of
like sheet music instructions
for a player-piano.



MIDI-capable Instrument



Speaker



MIDI device knows how to
'read' a MIDI file and play back
the sound. The device might
be a synthesizer keyboard or
some other kind of instrument.
Usually, a MIDI instrument
can play a LOT of different
sounds (piano, drums, violin,
etc.), and all at the same time.
So a MIDI file isn't like sheet
music for just one musician in
the band — it can hold the
parts for ALL the musicians
playing a particular song.

First we need a Sequencer

Before we can get any sound to play, we need a Sequencer object. The sequencer is the object that takes all the MIDI data and sends it to the right instruments. It's the thing that *plays* the music. A sequencer can do a lot of different things, but in this book, we're using it strictly as a playback device. Like a CD-player on your stereo, but with a few added features. The Sequencer class is in the javax.sound.midi package (part of the standard Java library as of version 1.3). So let's start by making sure we can make (or get) a Sequencer object.

```
import javax.sound.midi.*;
public class MusicTest1 {
    public void play() {
        Sequencer sequencer = MidiSystem.getSequencer();
        System.out.println("We got a sequencer");
    } // close play

    public static void main(String[] args) {
        MusicTest1 mt = new MusicTest1();
        mt.play();
    } // close main
} // close class
```

← import the javax.sound.midi package

We need a Sequencer object. It's the main part of the MIDI device/instrument we're using. It's the thing that, well, sequences all the MIDI information into a 'song'. But we don't make a brand new one ourselves -- we have to ask the MidiSystem to give us one.

Something's wrong!

This code won't compile! The compiler says there's an 'unreported exception' that must be caught or declared.

File Edit Window Help SayWhat?

```
% javac MusicTest1.java
```

```
MusicTest1.java:13: unreported exception javax.sound.midi.  
MidiUnavailableException; must be caught or declared to be  
thrown
```

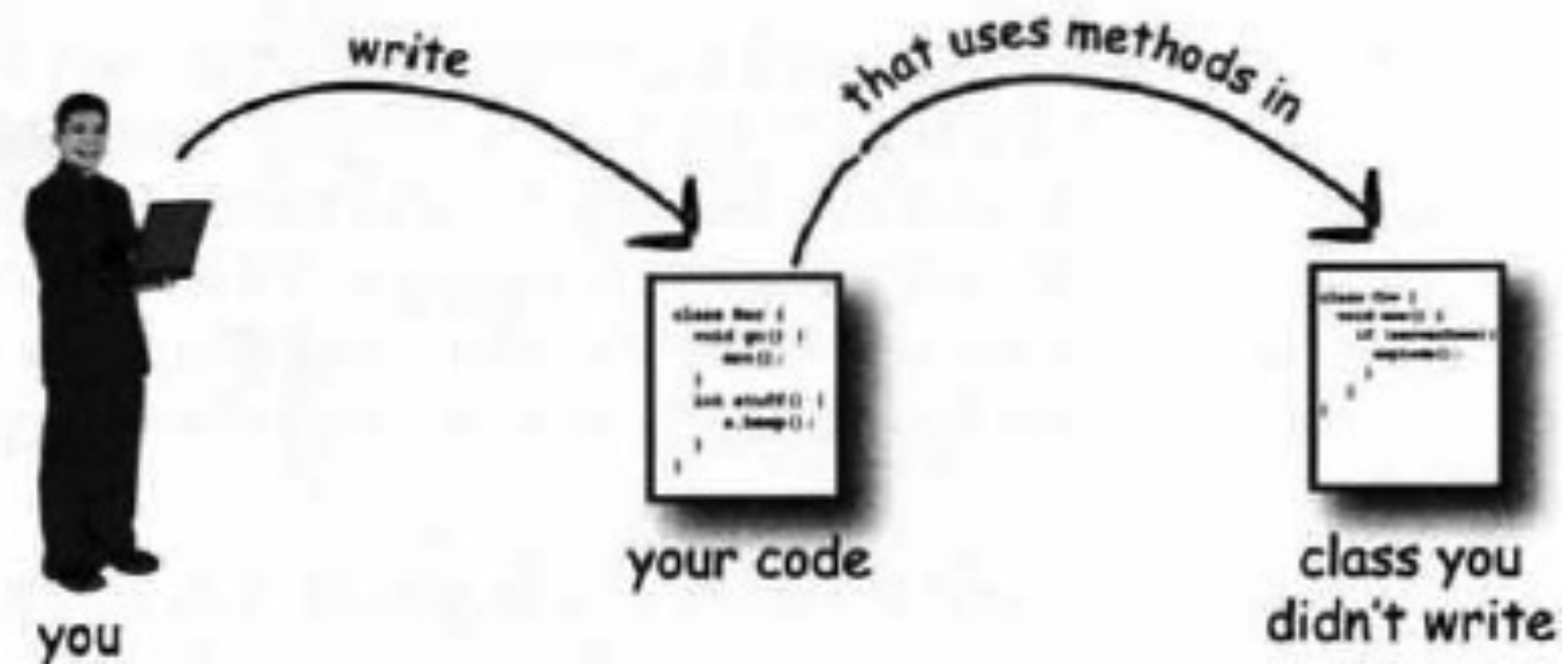
```
    Sequencer sequencer = MidiSystem.getSequencer();
```

^

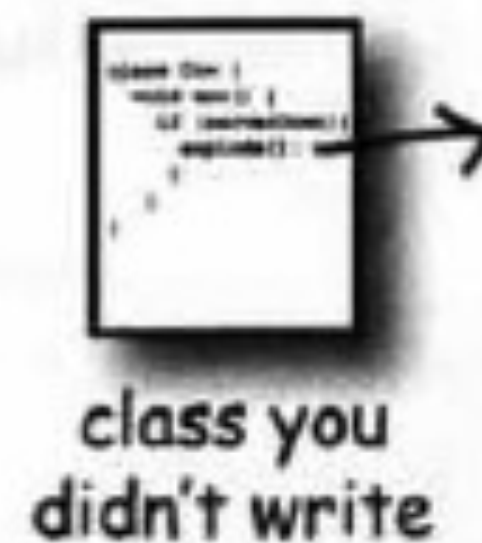
```
1 errors
```

What happens when a method you want to call (probably in a class you didn't write) is risky?

- ① Let's say you want to call a method in a class that you didn't write.



- ② That method does something risky, something that might not work at runtime.



```
void moo() {  
    if (serverDown) {  
        explode();  
    }  
}
```


- ③ You need to *know* that the method you're calling is risky.



you

I wonder if that method could blow up...

My moo() method will explode if the server is down.

```
class Cow {  
  void moo() {  
    if (serverDown())  
      explode();  
  }  
}
```

class you didn't write

- ④ You then write code that can handle the failure if it *does* happen. You need to be prepared, just in case.



you

Now that I know, I can take precautions.

write safely

```
class Her {  
  void go() {  
    try {  
      moo();  
    } catch (DEW de) {  
      wry();  
    }  
  }  
}
```

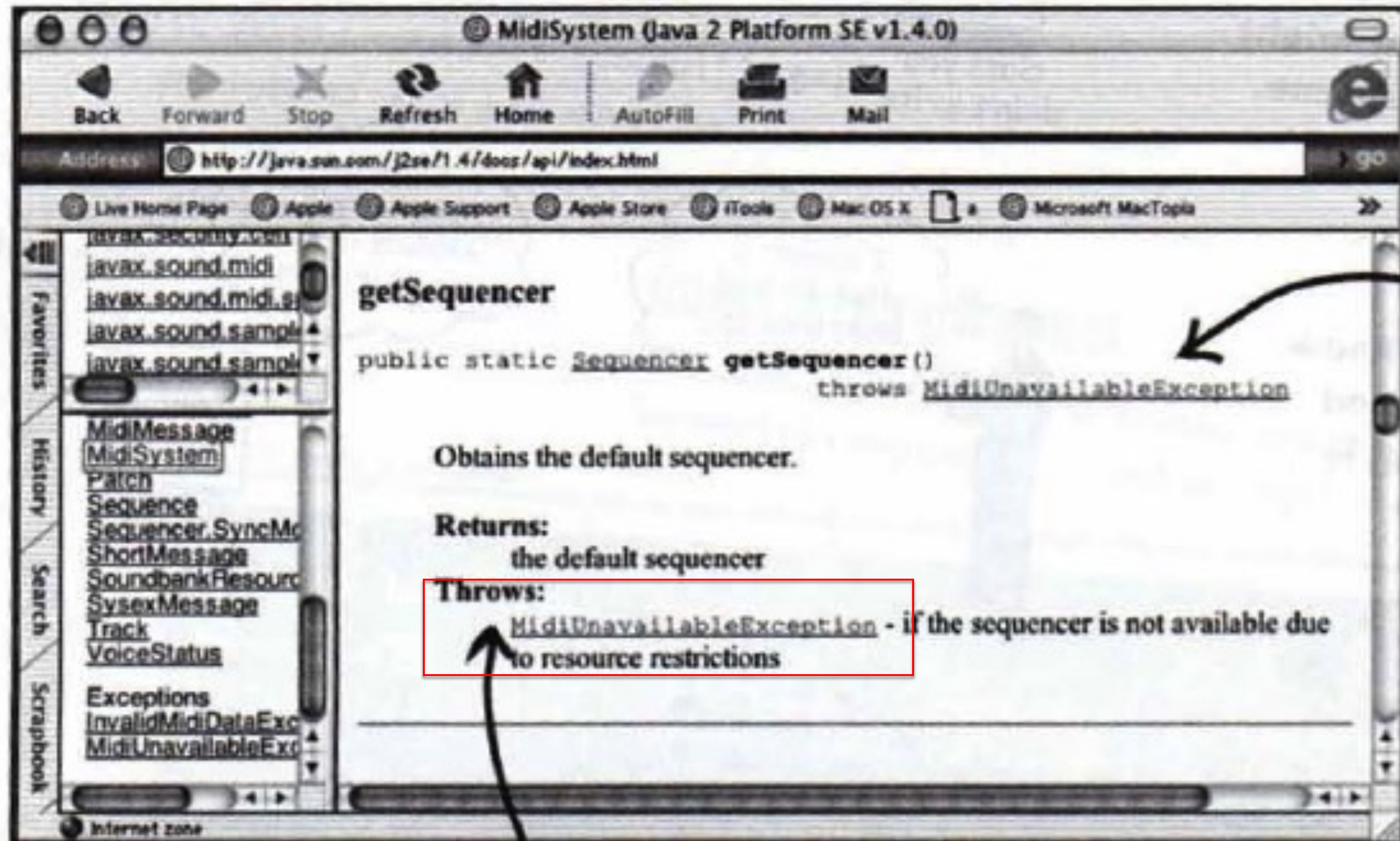
your code

**Methods in Java use *exceptions* to tell the calling code,
"Something Bad Happened. I failed."**

Java's exception-handling mechanism is a clean, well-lighted way to handle "exceptional situations" that pop up at runtime; it lets you put all your error-handling code in one easy-to-read place. It's based on you *knowing* that the method you're calling is risky (i.e. that the method *might* generate an exception), so that you can write code to deal with that possibility. If you *know* you might get an exception when you call a particular method, you can be *prepared* for—possibly even *recover* from—the problem that caused the exception.

So, how *do* you know if a method throws an exception? You find a **throws** clause in the risky method's declaration.

**The `getSequencer()` method takes a risk. It can fail at runtime.
So it must 'declare' the risk you take when you call it.**



The API does tell you that `getSequencer()` can throw an exception: `MidiUnavailableException`. A method has to declare the exceptions it might throw.

This part tells you **WHEN** you might get that exception -- in this case, because of resource restrictions (which could just mean the sequencer is already being used).

The compiler needs to know that YOU know you're calling a risky method.

If you wrap the risky code in something called a **try/catch**, the compiler will relax.

A try/catch block tells the compiler that you *know* an exceptional thing could happen in the method you're calling, and that you're prepared to handle it. That compiler doesn't care *how* you handle it; it cares only that you say you're taking care of it.

```
import javax.sound.midi.*;
```

```
public class MusicTest1 {
```

```
    public void play() {
```

```
        try {
```

```
            Sequencer sequencer = MidiSystem.getSequencer();
```

```
            System.out.println("Successfully got a sequencer");
```

```
        } catch (MidiUnavailableException ex) {
```

```
            System.out.println("Bummer");
```

```
        }
```

```
    } // close play
```

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

```
        MusicTest1 mt = new MusicTest1();
```

```
        mt.play();
```

```
    } // close main
```

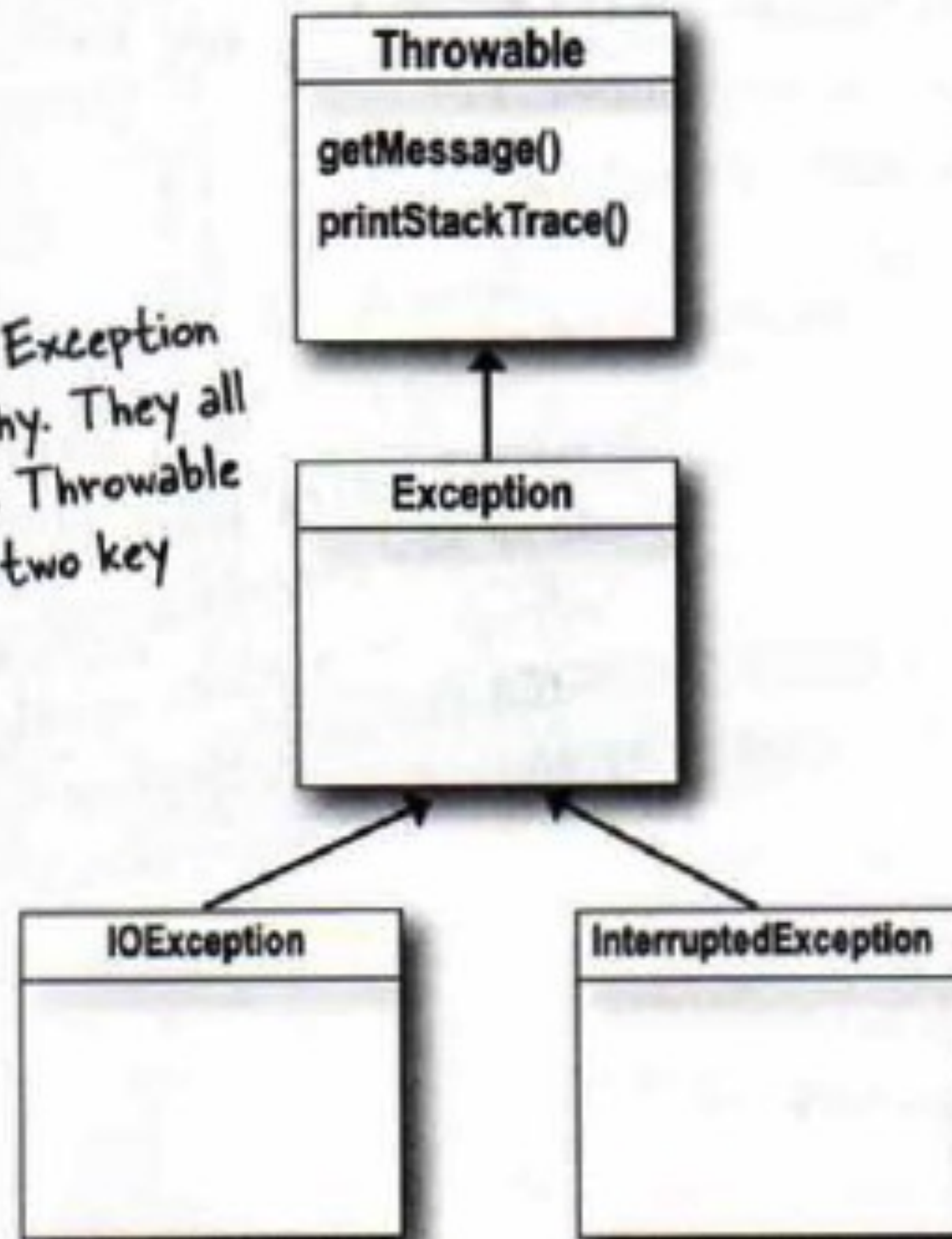
```
} // close class
```

← put the risky thing
in a 'try' block.

← make a 'catch' block for what to
do if the exceptional situation
happens -- in other words, a
MidiUnavailableException is thrown
by the call to `getSequencer()`.

An exception is an object... of type Exception.

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



```
try {
    // do risky thing
```

it's just like declaring
a method argument

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

This code only runs if an
Exception is thrown.

What you write in a catch block depends on the exception that was thrown. For example, if a server is down you might use the catch block to try another server. If the file isn't there, you might ask the user for help finding it.

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

- 处理异常要比创建和抛出异常花费更多的时间。
- 当代码调用有风险的方法(声明异常的方法)时，则该代码处理异常。

① Risky, exception-throwing code:

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

this method **MUST** tell the world (by declaring) that it throws a `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.

One method will catch what another method throws. An exception is always thrown back to the caller.

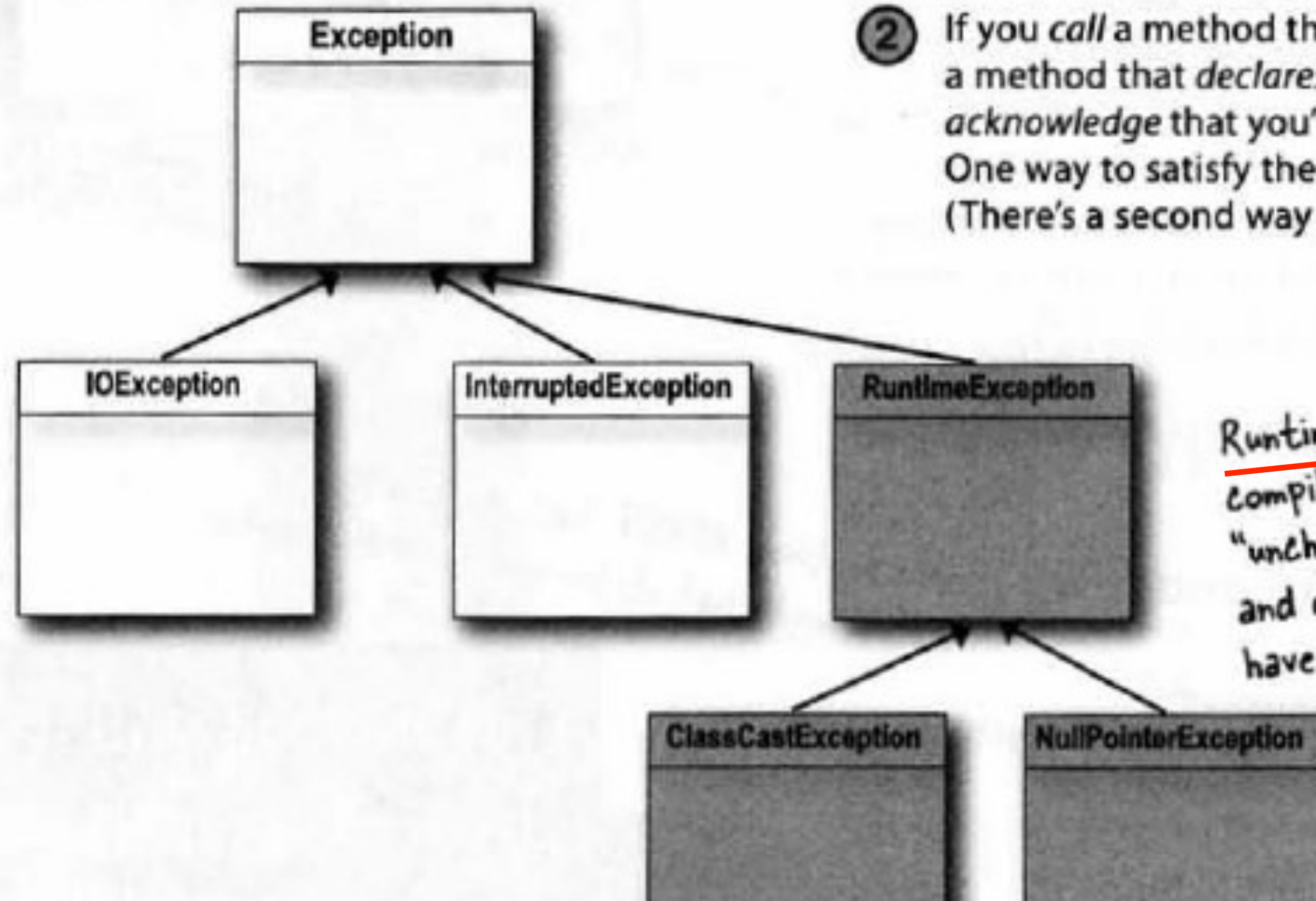
The method that throws has to declare that it might throw the exception.

The compiler checks for everything except RuntimeExceptions.

The compiler guarantees:

- ① If you *throw* an exception in your code you *must* declare it using the *throws* keyword in your method declaration.
- ② If you *call* a method that throws an exception (in other words, a method that *declares* it throws an exception), you must *acknowledge* that you're aware of the exception possibility. One way to satisfy the compiler is to wrap the call in a try/catch. (There's a second way we'll look at a little later in this chapter.)

Exceptions that are *NOT* subclasses of RuntimeException are checked for by the compiler. They're called "checked exceptions"



RuntimeExceptions are NOT checked by the compiler. They're known as (big surprise here) "unchecked exceptions". You can throw, catch, and declare RuntimeExceptions, but you don't have to, and the compiler won't check.

Q: Wait just a minute! How come this is the FIRST time we've had to try/catch an Exception? What about the exceptions I've already gotten like NullPointerException and the exception for DivideByZero. I even got a NumberFormatException from the Integer.parseInt() method. How come we didn't have to catch those?

A: The compiler cares about all subclasses of Exception, *unless* they are a special type, RuntimeException. Any exception class that extends RuntimeException gets a free pass. RuntimeExceptions can be thrown anywhere, with or without throws declarations or try/catch blocks. The compiler doesn't bother checking whether a method declares that it throws a RuntimeException, or whether the caller acknowledges that they might get that exception at runtime.

Q: I'll bite. WHY doesn't the compiler care about those runtime exceptions? Aren't they just as likely to bring the whole show to a stop?

A: Most RuntimeExceptions come from a problem in your code logic, rather than a condition that fails at runtime in ways that you cannot predict or prevent. You *cannot* guarantee the file is there. You *cannot* guarantee the server is up. But you *can* make sure your code doesn't index off the end of an array (that's what the .length attribute is for).

You WANT RuntimeExceptions to happen at development and testing time. You don't want to code in a try/catch, for example, and have the overhead that goes with it, to catch something that shouldn't happen in the first place.

A try/catch is for handling exceptional situations, not flaws in your code. Use your catch blocks to try to recover from situations you can't guarantee will succeed. Or at the very least, print out a message to the user and a stack trace, so somebody can figure out what happened.

- 如存在风险，抛出异常。
- 异常是一个exception类型的对象。
- 编译器会忽视RuntimeException类型的异常。RuntimeException类型的异常不需被声明或者在try/catch中处理。
- 编译器只关心可检查异常，其必须在代码中声明和处理。
- 方法体中用throw关键词抛出异常对象 `throw new Exception ();`
- 抛出可检查异常的方法必须声明异常 `throws Exception。`
- 编译器会检查可检查异常是否被处理，即 `try-catch`
- 可以一直 `throws`

Flow control in try/catch blocks

If the try **succeeds**

(doRiskyThing() does *not* throw an exception)

First the try block runs, then the code below the catch runs.

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

The code in the catch block never runs.

```
File Edit Window Help RiskAll  
%java Tester  
We made it!
```

If the try fails

(because doRiskyThing()
does throw an exception)

The try block runs, but the
call to doRiskyThing() throws
an exception, so the rest of
the try block doesn't run.

The catch block runs, then
the method continues on.

```
try {  
  ① Foo f = x.doRiskyThing();  
    int b = f.getNum();  
  
  } catch (Exception ex) {  
    ② System.out.println("failed");  
  }  
  ③ System.out.println("We made it!");  
}
```

The rest of the try block nev-
er runs, which is a Good Thing
because the rest of the try
depends on the success of
the call to doRiskyThing().

```
File Edit Window Help RiskAll  
%java Tester  
failed  
We made it!
```


Finally

A finally block is where you put code that must run *regardless* of an exception.

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

Without finally, you have to put the `turnOvenOff()` in *both* the try and the catch because *you have to turn off the oven no matter what*. A finally block lets you put all your important cleanup code in *one* place instead of duplicating it like this:

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

Multiple exceptions

Catching multiple exceptions

The compiler will make sure that you've handled *all* the checked exceptions thrown by the method you're calling. Stack the *catch* blocks under the *try*, one after the other. Sometimes the order in which you stack the catch blocks matters, but we'll get to that a little later.

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



↖ ↗
This method declares two, count 'em,
TWO exceptions.

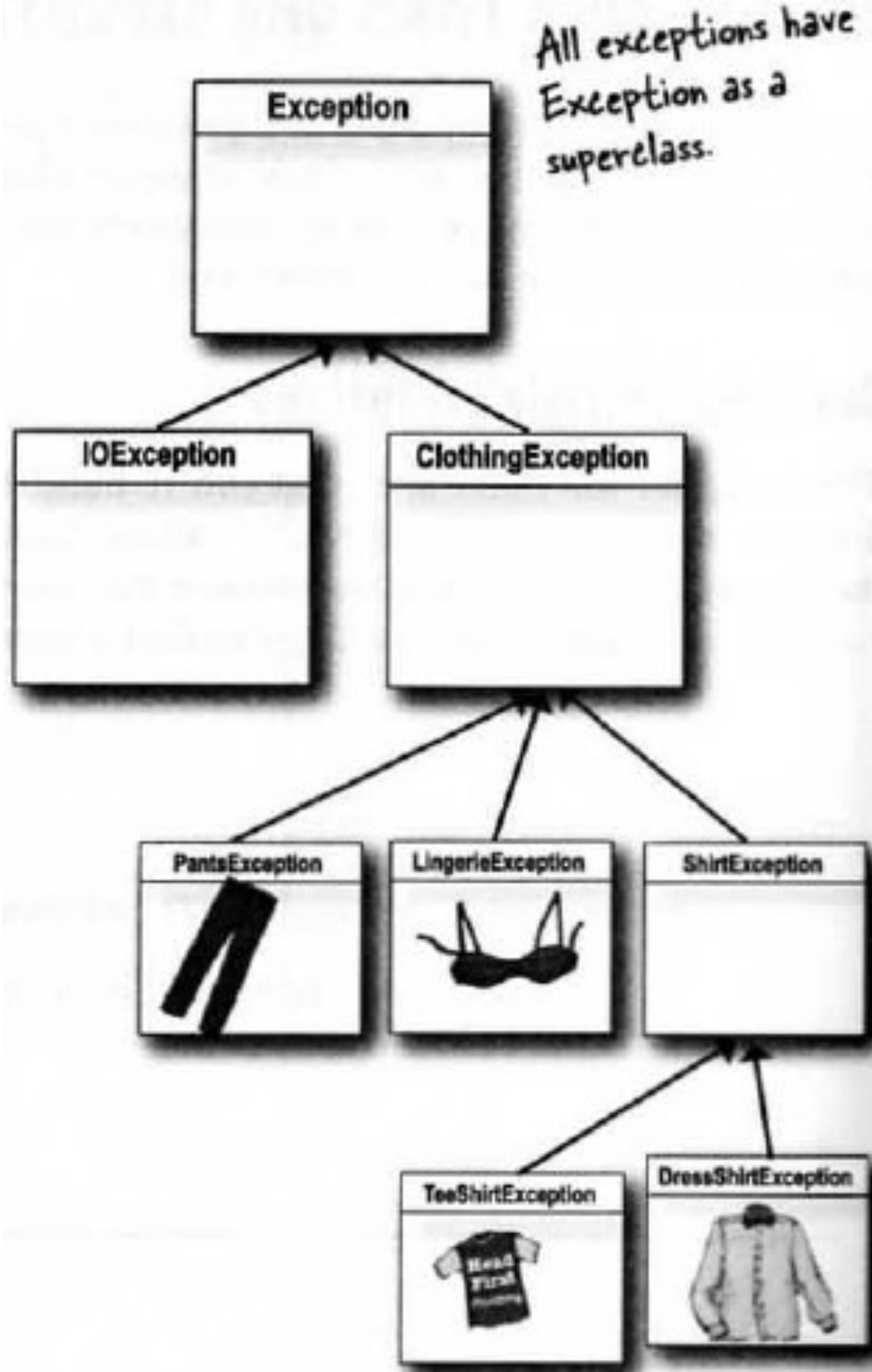

```
public class Foo {  
    public void go() {  
        Laundry laundry = new Laundry();  
        try {  
            laundry.doLaundry();  
        } catch (PantsException pex) {  
            // recovery code  
        } catch (LingerieException lex) {  
            // recovery code  
        }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

if doLaundry() throws a
PantsException, it lands in the
PantsException catch block.




if doLaundry() throws a
LingerieException, it lands in the
LingerieException catch block.

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.

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



Just because you CAN catch everything with one big super polymorphic catch, doesn't always mean you SHOULD.

You *could* write your exception-handling code so that you specify only *one* catch block, using the supertype Exception in the catch clause, so that you'll be able to catch *any* exception that might be thrown.

```
try {  
    laundry.doLaundry();  
} catch (Exception ex) {  
    // recovery code...  
}
```

Recovery from WHAT? This catch block will catch ANY and all exceptions, so you won't automatically know what went wrong.

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

For example, if your code deals with (or recovers from) a `TeeShirtException` differently than it handles a `LingerieException`, write a catch block for each. But if you treat all other types of `ClothingException` in the same way, then add a `ClothingException` catch to handle the rest.

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



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



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



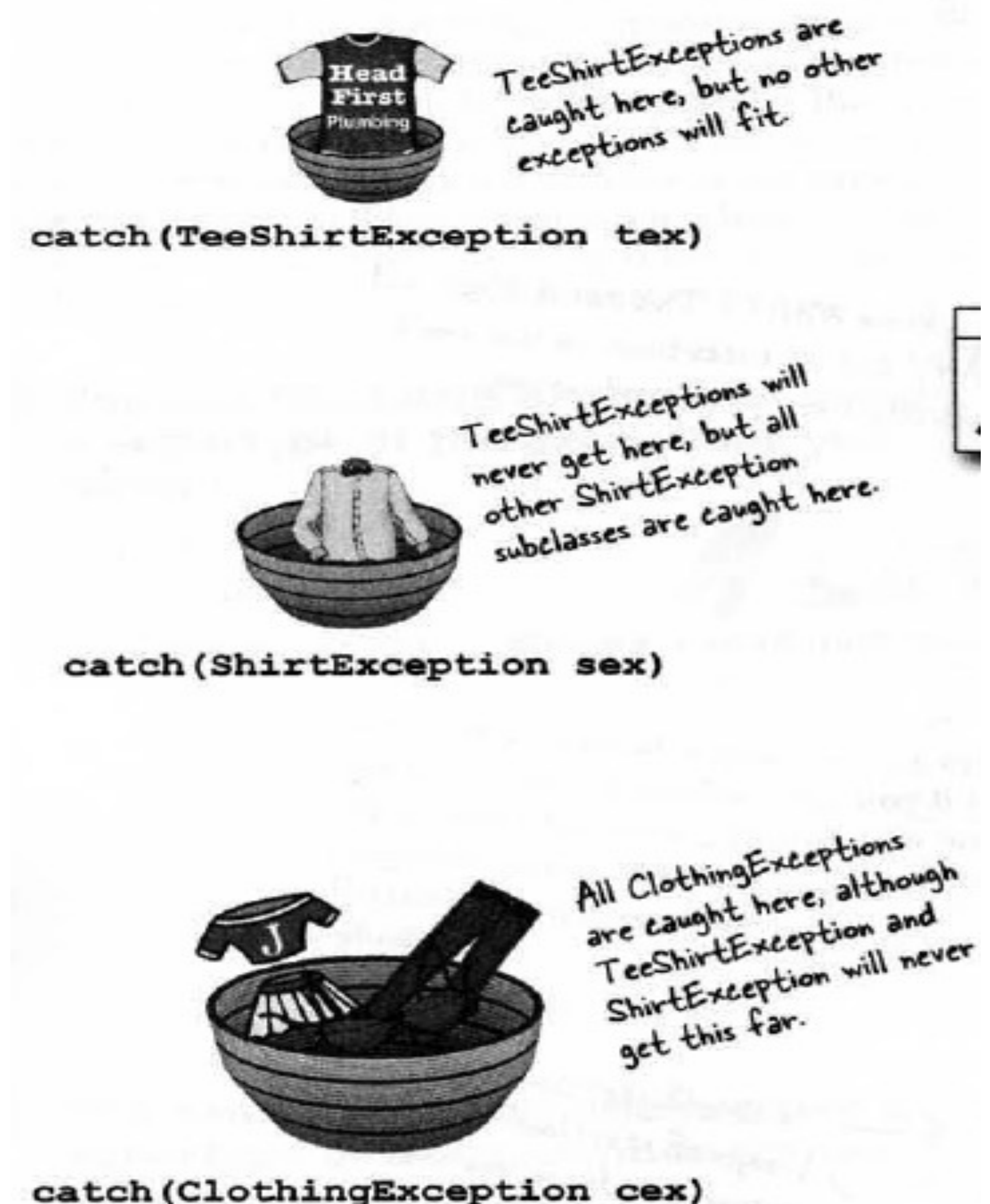
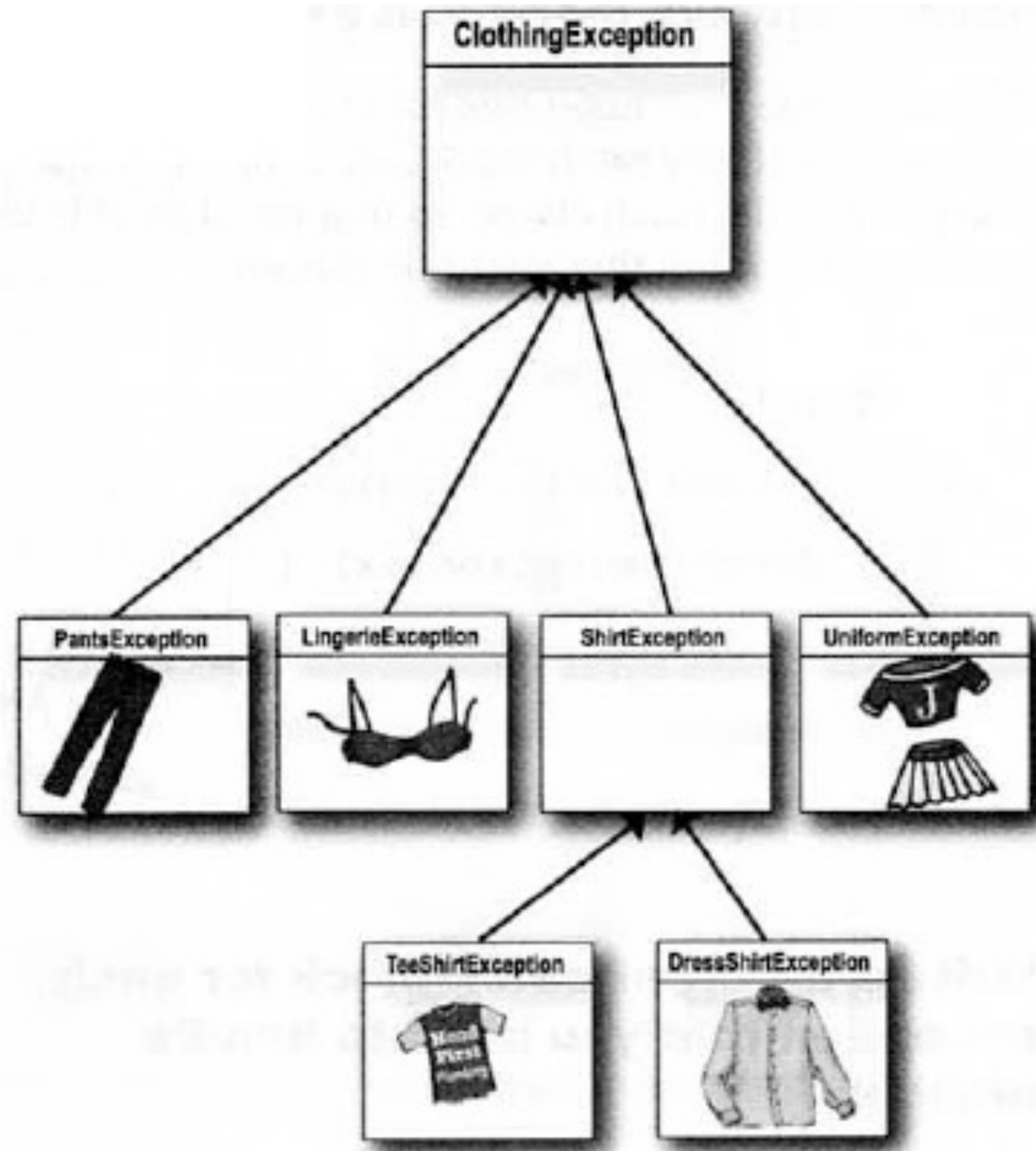
```
} catch (ClothingException cex) {  
    // recovery from all others
```

```
}
```

← 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



- 异常类位于继承树位置越高，能捕获的“篮子”就越大。
- Exception能够捕获所有的异常，包括运行时异常。

You can't put bigger baskets above smaller baskets.

Well, you *can* but it won't compile. Catch blocks are not like overloaded methods where the best match is picked. With catch blocks, the JVM simply starts at the first one and works its way down until it finds a catch that's broad enough (in other words, high enough on the inheritance tree) to handle the exception. If your first catch block is **catch (Exception ex)**, the compiler knows there's no point in adding any others—they'll never be reached.

Siblings can be in any order, because they can't catch one another's exceptions.

You could put `ShirtException` above `LingerieException` and nobody would mind. Because even though `ShirtException` is a bigger (broader) type because it can catch other classes (its own subclasses), `ShirtException` can't catch a `LingerieException` so there's no problem.

**When you don't want to handle
an exception...**

just duck it

**If you don't want to handle an
exception, you can **duck** it by
declaring it.**

When a method throws an exception, that method is popped off the stack immediately, and the exception is thrown to the next method down the stack—the *caller*. But if the *caller* is a *ducker*, then there's no catch for it so the *caller* pops off the stack immediately, and the exception is thrown to the next method and so on... where does it end? You'll see a little later.

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

← You don't REALLY throw it, but since you don't have a try/catch for the risky method you call, YOU are now the "risky method". Because now, whoever calls YOU has to deal with the exception.

Ducking (by declaring) only delays the inevitable

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



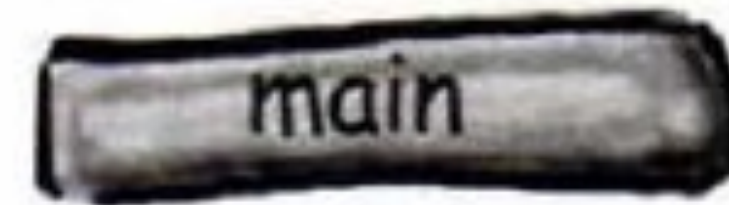
main() calls foo()
foo() calls doLaundry()
doLaundry() is running and throws a ClothingException

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

Handle or Declare. It's the law.

So now we've seen both ways to satisfy the compiler when you call a risky (exception-throwing) method.

🕒 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.

❶ 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.

But now this means that whoever calls the foo() method has to follow the Handle or Declare law. If foo() ducks the exception (by declaring it), and main() calls foo(), then main() has to deal with the exception.

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

TROUBLE!!

Now main() won't compile, and we get an "unreported exception" error. As far as the compiler's concerned, the foo() method throws an exception.

Because the foo() method ducks the ClothingException thrown by doLaundry(), main() has to wrap a.foo() in a try/catch, or main() has to declare that it, too, throws ClothingException!

Getting back to our music code...

Now that you've completely forgotten, we started this chapter with a first look at some JavaSound code. We created a Sequencer object but it wouldn't compile because the method `Midi.getSequencer()` declares a checked exception (`MidiUnavailableException`). But we can fix that now by wrapping the call in a try/catch.

```
public void play() {  
    try {  
  
        Sequencer sequencer = MidiSystem.getSequencer();  
        System.out.println("Successfully got a sequencer");  
  
    } catch (MidiUnavailableException ex) {  
        System.out.println("Bummer");  
    }  
} // close play
```

↙ No problem calling `getSequencer()`, now that we've wrapped it in a try/catch block.

↖ The catch parameter has to be the 'right' exception. If we said `catch(FileNotFoundException f)`, the code would not compile, because polymorphically a `MidiUnavailableException` won't fit into a `FileNotFoundException`. Remember it's not enough to have a catch block... you have to catch the thing being thrown!

Exception Rules

- ❶ You cannot have a catch or finally without a try

```
void go() {  
    Foo f = new Foo();  
    f.foo();  
    catch(FooException ex) { }  
}
```

NOT LEGAL!
Where's the try?

- ❷ You cannot put code between the try and the catch

```
try {  
    x.doStuff();  
}  
int y = 43;  
} catch(Exception ex) { }
```

NOT LEGAL! You can't put
code between the try and
the catch.

- ❸ A try MUST be followed by either a catch or a finally

```
try {  
    x.doStuff();  
} finally {  
    // cleanup  
}
```

LEGAL because you
have a finally, even
though there's no catch.
But you cannot have a
try by itself.

- ❹ A try with only a finally (no catch) must still declare the exception.

```
void go() throws FooException {  
    try {  
        x.doStuff();  
    } finally { }  
}
```

A try without a catch
doesn't satisfy the
handle or declare law

在继承中重写方法时抛出异常的问题

- 1，子类重写父类方法要抛出与父类一致的异常或异常子类，或者不抛出异常
- 2，子类重写父类方法所抛出的异常不能超过父类的范畴（仅指检查型异常）。
备注：子类重写的方法可以抛出任何运行期异常
- 3，子类在重写父类的具有异常声明的方法的同时，又去实现了具有相同方法名称的接口且该接口中的方法也具有异常声明，则子类中的重写的方法，要么不抛出异常，要么抛出父类中方法声明异常与接口中方法声明的异常的交集。

- 现有类
- class A{
- public void show() throws
 IOException,ClassNotFoundException,
 ArithmeticException {
- System.out.println("show");
- }
- }
- 以下类定义正确的是
- A. class Son extends A {
- public void show() throws
- IOException,ClassNotFoundException,
 ParseException {
- System.out.println("show");
- }
- }
- C. class Son extends A {
- public void show() throws
- IOException,ClassNotFoundException{
- System.out.println("show");
- }
- }
- D. class Son extends A {
- public void show() throws {
- System.out.println("show");
- }
- }
- [答案]ACD

- 先看例子1

- ```
public class TestRegularExperssion {
```
- ```
public void printA(float a,float b)throws  
SQLException
```
- ```
{
```
- ```
}
```
- ```
}
```
- ```
class TestRegularExpressionA extends  
TestRegularExperssion
```
- ```
{
```
- ```
public void printA(float a,float b)throws  
NullPointerException
```
- ```
{
```
- ```
}
```
- ```
} //编译通过
```

- 例子2

- ```
public class TestRegularExperssion {
```
- ```
public void printA(float a,float b)throws
NullPointerException
```
- ```
{
```
- ```
}
```
- ```
}
```
- ```
class TestRegularExpressionA extends
TestRegularExperssion
```
- ```
{
```
- ```
public void printA(float a,float b)throws
SQLException
```
- ```
{
```
- ```
}
```
- ```
} //编译不通过
```


使用异常机制的建议

- 异常的声明是API的一部分
- 异常处理不能代替简单的测试
- 不要过分地细化异常
- 利用异常层次结构
 - 不要只抛出RuntimeException异常，应该寻找更合适的子类或者创建自己的异常类
- 不要压制异常
- 在检测错误时，“苛刻”要比放任更好
 - 在出错的地方抛出一个EmptyStackException异常要比在后面抛出一个NullPointerException异常更好
- 不要羞于传递异常
 - 早抛出，晚捕获

创建自己的异常

- 精心设计异常的层次结构
- 异常类中包含足够的信息
- 异常与错误提示

代码清单 1-6 使用异常包装技术的示例

```
public class DataAccessGateway {  
    public void load() throws DataAccessException {  
        try {  
            FileInputStream input = new FileInputStream("data.txt");  
  
        }  
        catch (IOException e) {  
            throw new DataAccessException(e);  
        }  
    }  
}
```

异常的消失

代码清单 1-9 异常消失的示例

```
public class DisappearedException {  
    public void show() throws BaseException{  
        try {  
            Integer.parseInt("Hello");  
        }  
        catch (NumberFormatException nfe) {  
            throw new BaseException(nfe);  
        } finally {  
            try {  
                int result = 2 / 0;  
            } catch (ArithmeticException ae) {  
                throw new BaseException(ae);  
            }  
        }  
    }  
}
```


两种解决办法

- Solution 1
 - 抛出try语句块中阐述的原始异常，忽略在finally语句块中产生的异常
- Solution 2
 - 把产生的异常都记录下来

Solution 1

代码清单 1-10 抛出 try 语句块中产生的原始异常的示例

```
public class ReadFile {
    public void read(String filename) throws BaseException {
        FileInputStream input = null;
        IOException readException = null;
        try {
            input = new FileInputStream(filename);
        } catch (IOException ex) {
            readException = ex;
        } finally {
            if (input != null) {
                try {
                    input.close();
                } catch (IOException ex) {
                    if (readException == null) {
                        readException = ex;
                    }
                }
            }
        }
        if (readException != null) {
            throw new BaseException(readException);
        }
    }
}
```


Solution 2

代码清单 1-11 使用 addSuppressed 方法记录异常的示例

```
public class ReadFile {
    public void read(String filename) throws IOException {
        FileInputStream input = null;
        IOException readException = null;
        try {
            input = new FileInputStream(filename);
        } catch (IOException ex) {
            readException = ex;
        } finally {
            if (input != null) {
                try {
                    input.close();
                } catch (IOException ex) {
                    if (readException != null) {
                        readException.addSuppressed(ex);
                    }
                    else {
                        readException = ex;
                    }
                }
            }
        }
        if (readException != null) {
            throw readException;
        }
    }
}
```

Java 7的异常处理新特性

- 一个catch子句捕获多个异常
- 更加精确的异常抛出

代码清单 1-12 在 catch 子句中指定多种异常

```
public class ExceptionHandler {  
    public void handle() {  
        ExceptionThrower thrower = new ExceptionThrower();  
        try {  
            thrower.manyExceptions();  
        } catch (ExceptionA | ExceptionB ab) {  
        } catch (ExceptionC c) {  
        }  
    }  
}
```

每个异常类型之间使用“|”来分隔

注意：

在 catch 子句中声明捕获的这些异常类，不能出现重复的类型，也不允许其中的某个异常是另外一个异常的子类，否则会出现编译错误。

代码清单 1-17 精确的异常抛出的示例

```
public class PreciseThrowUse {  
    public void testThrow() throws ExceptionA {  
        try {  
            throw new ExceptionASub2();  
        }  
        catch(ExceptionA e) {  
            try {  
                throw e;  
            }  
            catch (ExceptionASub1 e2) { // 编译错误  
            }  
        }  
    }  
}
```

在上面的代码中,异常类 `ExceptionASub1` 和 `ExceptionASub2` 都是 `ExceptionA` 的子类。方法 `testThrow` 中首先抛出了 `ExceptionASub2` 异常,通过第一个 `catch` 子句捕获之后重新抛出。在这里,Java 编译器可以准确知道变量 `e` 表示的异常类型是 `ExceptionASub2`,接下来的第二个 `catch` 子句试图捕获 `ExceptionASub1` 类型的异常,这显然是不可能的,因此会产生编译错误。上面的代码在 Java 6 编译器上是可以通过编译的。对于 Java 6 编译器来说,第二个 `try` 子句中抛出的异常类型是前一个 `catch` 子句中声明的 `ExceptionA` 类型,因此在第二个 `catch` 子句中尝试捕获 `ExceptionA` 的子类型 `ExceptionASub1` 是合法的。

Java 7中引入try-with-resources语句

代码清单 1-18 读取磁盘文件内容的示例

```
public class ResourceBasicUsage {  
    public String readFile(String path) throws IOException {  
        try (BufferedReader reader = new BufferedReader(new FileReader(path))) {  
            StringBuilder builder = new StringBuilder();  
            String line = null;  
            while ((line = reader.readLine()) != null) {  
                builder.append(line);  
                builder.append(String.format("%n"));  
            }  
            return builder.toString();  
        }  
    }  
}
```

代码清单 1-19 自定义资源使用 AutoCloseable 接口的示例

```
public class CustomResource implements AutoCloseable {
    public void close() throws Exception {
        System.out.println(" 进行资源释放。");
    }

    public void useCustomResource() throws Exception {
        try (CustomResource resource = new CustomResource()) {
            System.out.println(" 使用资源。");
        }
    }
}
```

能够被 try 语句所管理的资源需要满足一个条件,那就是其 Java 类要实现 java.lang. AutoCloseable 接口,否则会出现编译错误。当需要释放资源的时候,该接口的 close 方法会被自动调用。Java 类库中已有不少接口或类继承或实现了这个接口,使得它们可以用在 try 语句中。在这些已有的常见接口或类中,最常用的就是与 I/O 操作和数据库相关的接口。与 I/O 相关的 java.io.Closeable 继承了 AutoCloseable,而与数据库相关的 java.sql.Connection、java.sql.ResultSet 和 java.sql.Statement 也继承了该接口。如果希望自己开发的类也能利用 try 语句的自动化资源管理,只需要实现 AutoCloseable 接口即可。

代码清单 1-20 使用 try-with-resources 语句管理两个资源的示例

```
public class MultipleResourcesUsage {  
    public void copyFile(String fromPath, String toPath) throws IOException {  
        try (InputStream input = new FileInputStream(fromPath);  
            OutputStream output = new FileOutputStream(toPath)) {  
            byte[] buffer = new byte[8192];  
            int len = -1;  
            while ((len = input.read(buffer)) != -1) {  
                output.write(buffer, 0, len);  
            }  
        }  
    }  
}
```

当finally子句包含return 语句

```
public class MyExceptionExample{

    public static int f(int n){
        int r;
        try{
            r = n*n;
            return r;
        }
        finally{
            r =2;
            return r;
        }
    }
    public static void main(String[] args){
        System.out.println(f(2));
    }
}
```


athrow指令

- throw语句
 - 由athrow指令实现
 - ; Throw an IOException. This is equivalent to the Java code:
 - ;
 - ; throw new java.io.IOException();
 - ;
 - new java/io/IOException ; 1) create and initialize an IOException instance
 - dup
 - invokespecial java/io/IOException/<init>()V
 - athrow ; 2) throw the IOException instance on the stack
- 运行时异常
 - idiv或ldiv指令中抛出ArithmeticException

```
promote:~ qinliu$ javap -verbose MyExceptionExample
Classfile /Users/qinliu/MyExceptionExample.class
  Last modified 2015-5-27; size 425 bytes
  MD5 checksum 1b6dbb4173666b6fb0eb54381864e541
  Compiled from "MyExceptionExample.java"
public class MyExceptionExample
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Methodref          #4.#16          //
java/lang/Object."<init>":()V
  #2 = Class               #17             // java/lang/Exception
  #3 = Class               #18             // MyExceptionExample
  #4 = Class               #19             // java/lang/Object
  #5 = Utf8                <init>
  #6 = Utf8                ()V
  #7 = Utf8                Code
  #8 = Utf8                LineNumberTable
  #9 = Utf8                inc
  #10 = Utf8               ()I
  #11 = Utf8               StackMapTable
  #12 = Class              #17             // java/lang/Exception
  #13 = Class              #20             // java/lang/Throwable
  #14 = Utf8               SourceFile
  #15 = Utf8               MyExceptionExample.java
  #16 = NameAndType        #5:#6          // "<init>":()V
  #17 = Utf8               java/lang/Exception
  #18 = Utf8               MyExceptionExample
  #19 = Utf8               java/lang/Object
  #20 = Utf8               java/lang/Throwable
{
  public MyExceptionExample();
    descriptor: ()V
    flags: ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
      0: aload_0
      1: invokespecial #1          // Method
java/lang/Object."<init>":()V
      4: return
    LineNumberTable:
      line 1: 0

  public int inc();
    descriptor: ()I
```

```
flags: ACC_PUBLIC
Code:
  stack=1, locals=5, args_size=1
   0: iconst_1
   1: istore_1
   2: iload_1
   3: istore_2
   4: iconst_3
   5: istore_1
   6: iload_2
   7: ireturn
   8: astore_2
   9: iconst_2
  10: istore_1
  11: iload_1
  12: istore_3
  13: iconst_3
  14: istore_1
  15: iload_3
  16: ireturn
  17: astore           4
  19: iconst_3
  20: istore_1
  21: aload           4
  23: athrow

Exception table:
   from    to  target type
     0      4      8  Class java/lang/Exception
     0      4     17   any
     8     13     17   any
    17     19     17   any

LineNumberTable:
 line 5: 0
 line 6: 2
 line 12: 4
 line 7: 8
 line 8: 9
 line 9: 11
 line 12: 13
StackMapTable: number_of_entries = 2
 frame_type = 72 /* same_locals_1_stack_item */
  stack = [ class java/lang/Exception ]
 frame_type = 72 /* same_locals_1_stack_item */
  stack = [ class java/lang/Throwable ]
}
SourceFile: "MyExceptionExample.java"
```

1. public class MyExceptionExample{
2. public int inc(){
3. int x;
4. try{
5. x=1;
6. return x;
7. }catch(Exception e){
8. x=2;
9. return x;
10. }
11. finally{
12. x=3;
13. }
14. }
15.}

catch中抛出异常

- 下面代码的输出结果是什么？

```
public class ZeroTest {  
    public static void main(String[] args) {  
        try {  
            int i = 100 / 0;  
            System.out.print(i);  
        } catch (Exception e) {  
            System.out.print(1);  
            throw new RuntimeException();  
        } finally {  
            System.out.print(2);  
        }  
        System.out.print(3);  
    }  
}
```

- 在catch中又抛出了异常，这时除了执行finally中的代码，往下的代码都不会执行了。