

# Exception Handling: Risky Behavior

Samkeun Kim <skim@hknu.ac.kr>

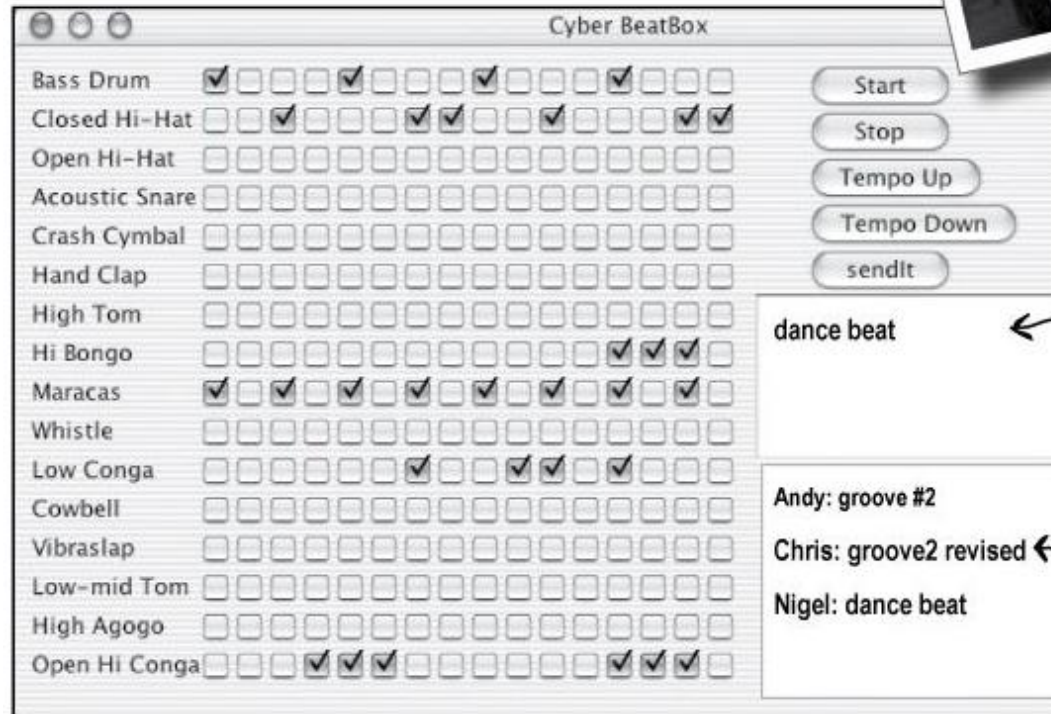
<http://cyber.hankyong.ac.kr>

# 비트박스 드럼 머신을 만들어보자

앞으로 몇 장에 걸쳐서 비트박스 드럼 머신을 포함한 몇 가지 사운드 애플리케이션을 만들어 본다.

The finished BeatBox looks something like this:

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



*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.*

# 기초부터 시작하자

## JavaSound API

**JavaSound**는 자바 버전 1.3부터 추가된 클래스 및 인터페이스의 컬렉션이다.

단순히 애드온 된 것이 아니라 표준 **J2SE** 클래스 라이브러리의 일부분으로 포함되었다.

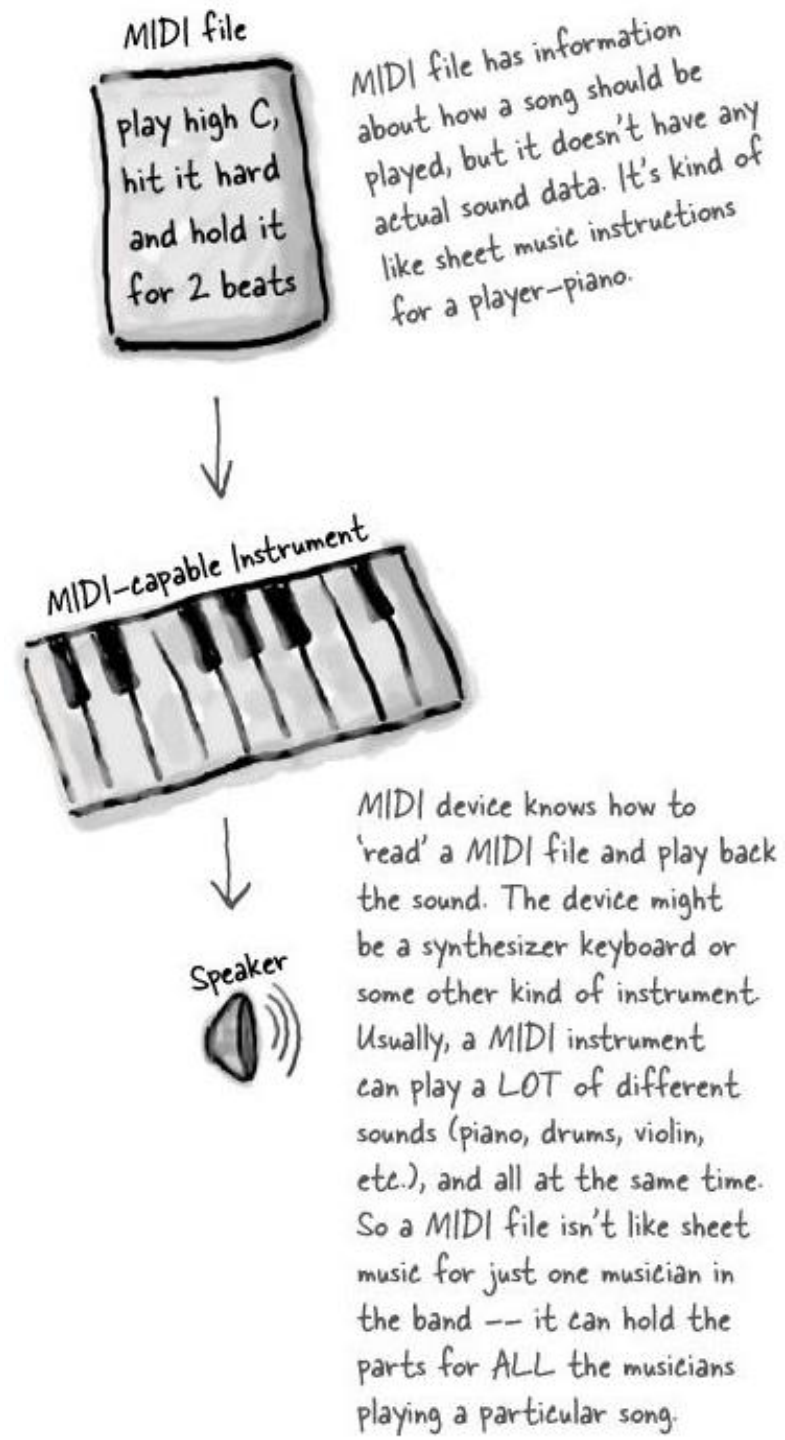
두 파트로 나뉜다: **MIDI**와 **Sampled**.

이 책에서는 **MIDI**만 사용하도록 한다.

- ✓ **MIDI**: Musical Instrument Digital Interface
- ✓ 다양한 전자 사운드 장비와 통신하기 위한 표준 프로토콜

**MIDI** 파일 vs. **악보**:

- ✓ **악보** => 밴드의 한 연주자를 위한 악보
- ✓ **MIDI** 파일 => 밴드의 모든 연주자를 위한 악보



# First we need a Sequencer

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

← import the javax.sound.midi package

```
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
```

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.

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
```

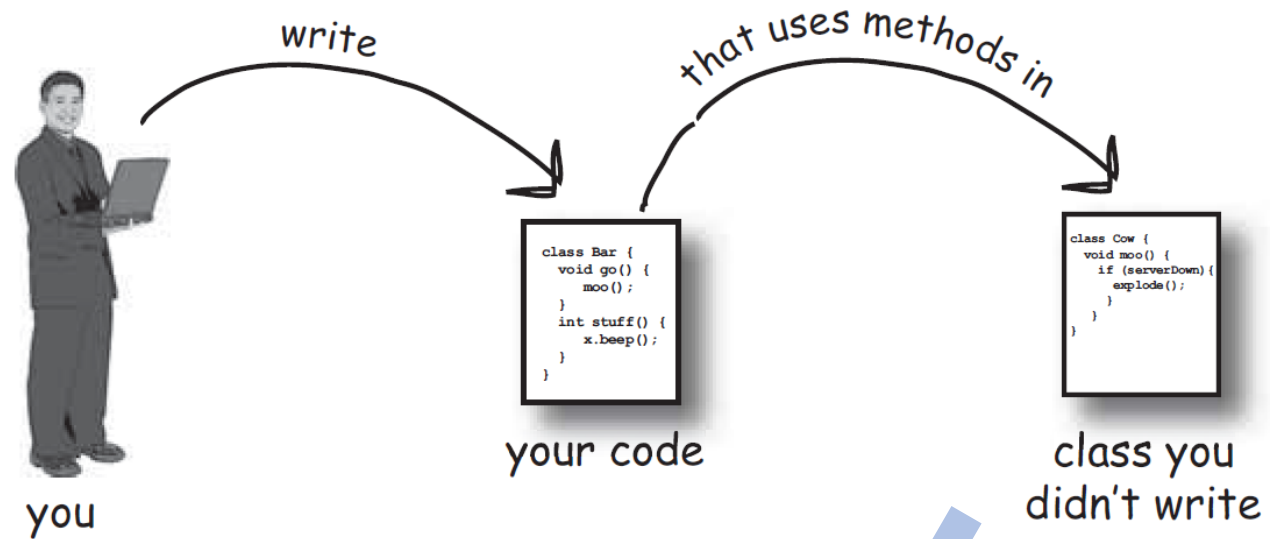
```
        Sequencer sequencer = MidiSystem.getSequencer();
```

```
1 errors
```

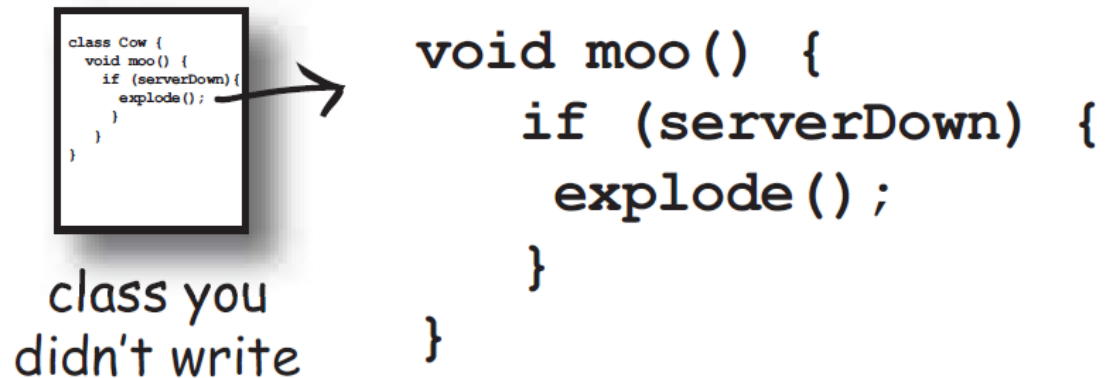
Something's wrong!

# 위험스런 메소드를 호출할 때 어떤 일이?

1. 다른 누군가 작성한 메소드를 호출한다고 가정해 보자.



2. 그 메소드는 실행 시에 동작하지 않을 수도 있는 뭔가 위험스런 일을 할 수도 있다.



3. 호출하고자 하는 메소드에 위험 요소가 있다는 것을 알 필요가 있다.



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. 다음은 실제로 실패했을 경우, 그 상황을 처리할 수 있는 코드를 작성한다. 실패했을 경우를 대비해서 미리 준비되어 있어야만 한다.



Now that I know, I can take precautions.

write safely

```
class Bar {  
  void go() {  
    try {  
      moo();  
    } catch (MX m) {  
      cry();  
    }  
  }  
}
```

your code

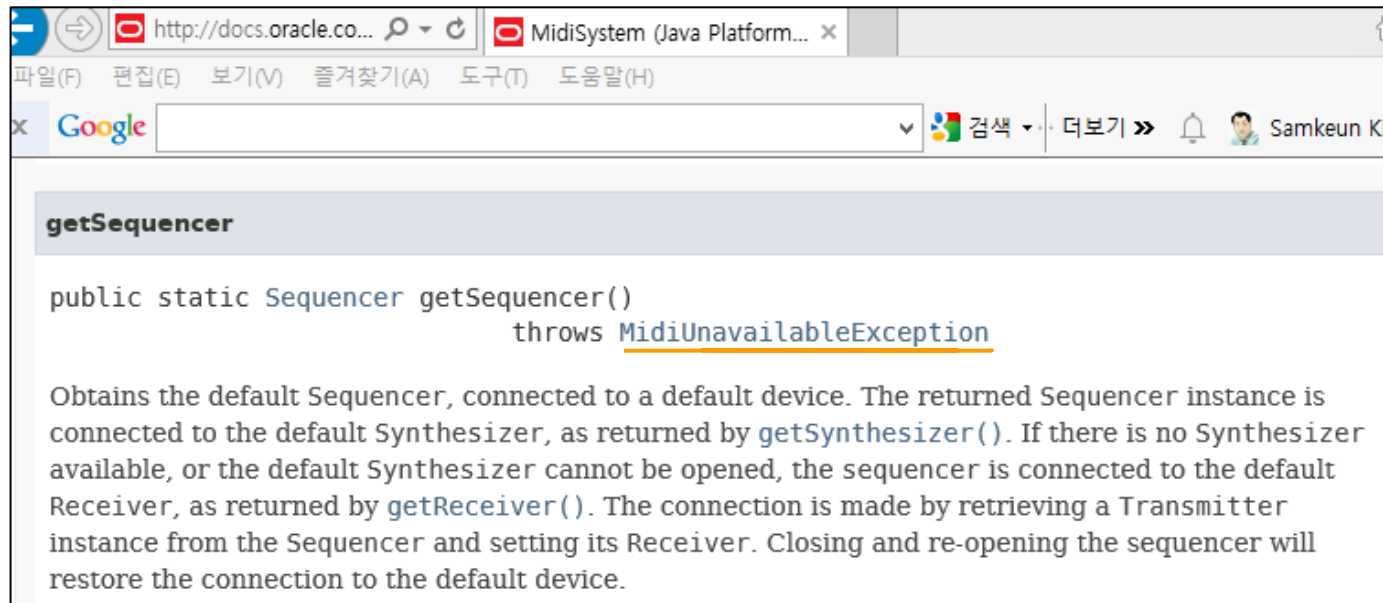


## 자바 메소드에서는 문제가 발생하면 예외를 사용하여 호출한 코드에 그 사실을 알려준다

자바의 예외 처리 메커니즘은 실행 시에 발생할 수 있는 “예외 상황” 을 처리할 수 있는 깔끔하고 부담이 없는 방법을 제공한다.

이 방법은 호출하고 있는 메소드가 위험하다는 것을 알고 있다는 사실에 기반하고 있다.

**getSequencer()** 메소드는 위험 요소가 있다. 실행 시에 실패할 수도 있다. 따라서 그것을 호출했을 때 발생할 수 있는 위험 요인을 ‘선언’ 해줘야만 한다.



<http://docs.oracle.com/javase/9/docs/api/javax/sound/midi/MidiSystem.html>

# 우리가 위험스런 메소드를 호출하고 있는 사실을 알고 있다는 것을 컴파일러는 알아야 한다

위험스런 코드를 **try/catch**로 감싸주기만 하면 컴파일러는 안심할 것이다.

**Try/catch** 블록은 프로그래머가 현재 호출하고 있는 메소드에서 뭔가 예외 상황이 발생할 수 있음을 알고 있고, 또 그것을 처리할 준비가 되어 있음을 컴파일러에게 알려준다.

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

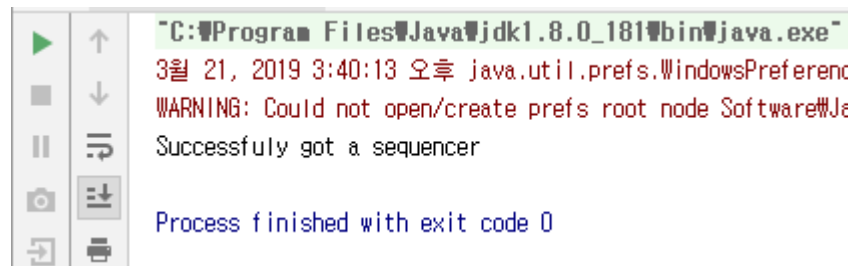
```
public class MusicTest1 {  
    public void play() {
```

```
        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().



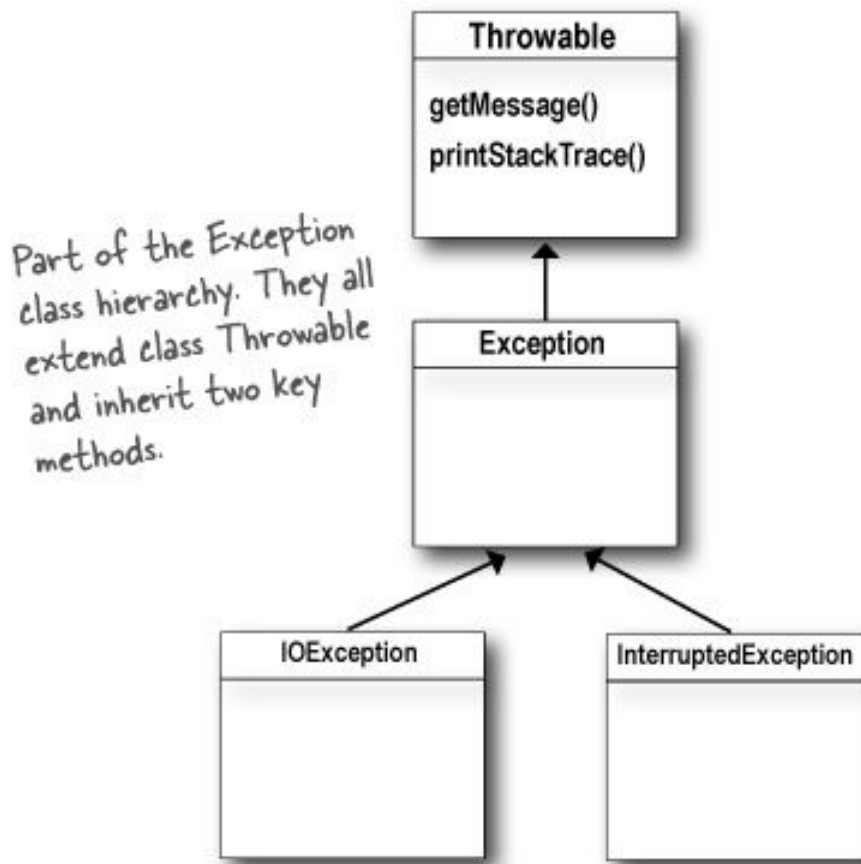
# 예외도 객체다...

예외는 **Exception** 타입의 객체다!

**Exception**이 객체이므로 '잡아야'  
하는 것도 객체다!

**Catch**의 인자는 **Exception** 타입으로  
선언되고, 매개변수인 레퍼런스 변수  
는 **ex**이다.

*catch 블록에 들어가는 것은 던져  
지는 예외에 따라 달라진다!*



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

*it's just like declaring a method argument.*

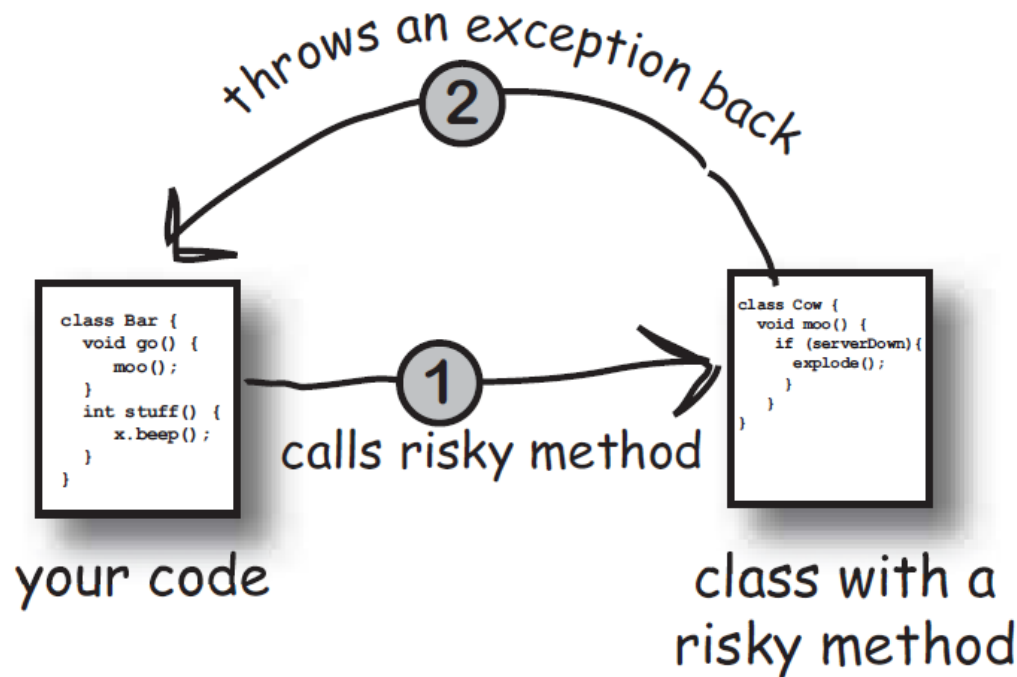
*This code only runs if an Exception is thrown.*

# 우리가 작성하는 코드가 예외를 잡는다면 그 예외를 던지는 코드는?

위험스런 메소드를 호출할 때 호출한 쪽으로 예외를 던지는 것은 바로 그 위험스런 메소드이다.

직접 그 위험스런 메소드를 만들어 보자.

중요한 것은 그 코드를 누가 만드느냐가 아니라 어느 메소드가 예외를 던지고 어느 메소드가 예외를 잡아내는지가 중요하다.



누군가 예외를 던질 수도 있는 코드를 작성한다면 반드시 해당 예외를 선언해야 한다!

## 1. 예외를 던지는 위험한 코드

```
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. 위험한 메소드를 호출하는 코드

```
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.

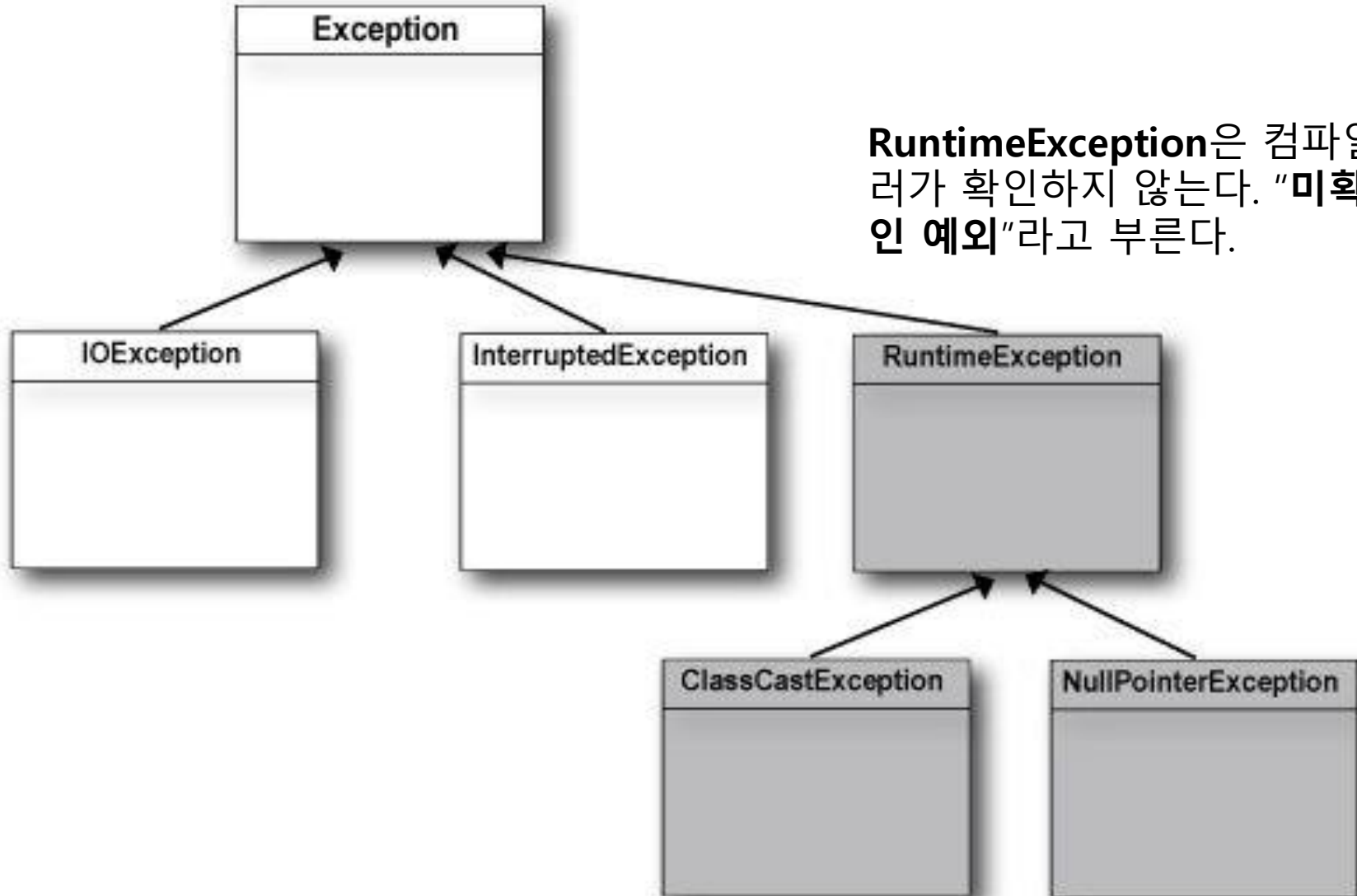
# 컴파일러에서는 RuntimeException을 제외한 모든 것을 확인한다

The compiler guarantees:

1. 작성하는 코드에서 예외를 던진다면 반드시 **throws**라는 키워드를 사용하여 예외를 선언해야만 한다.
2. 예외를 던지는 메소드를 호출한다면 예외 발생 가능성을 인지하고 있다는 것을 알려줘야 한다

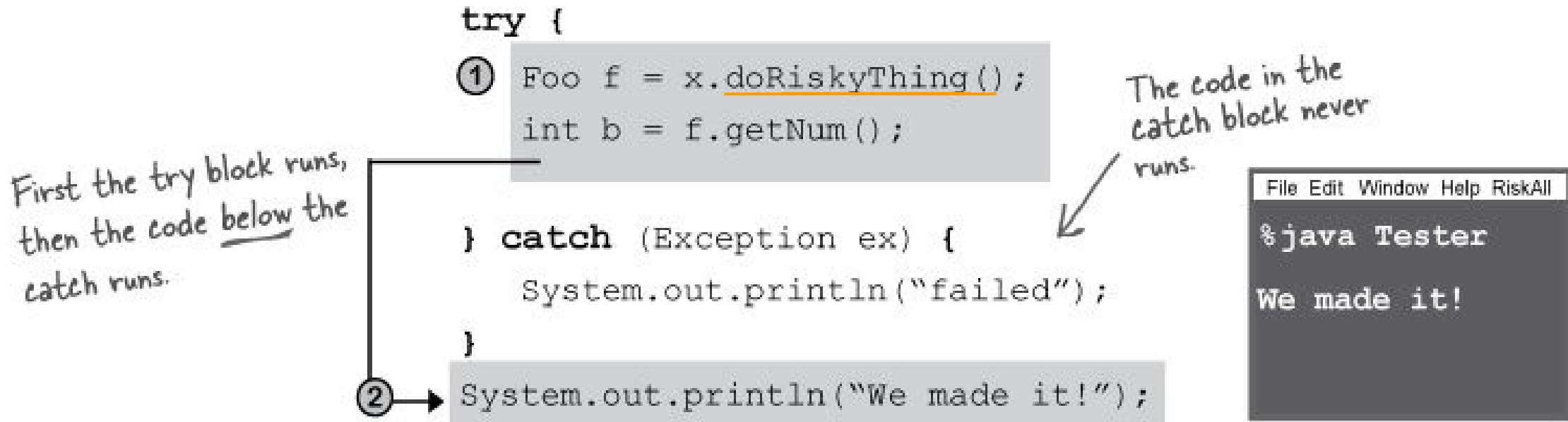
⇒ **try/catch** 문으로 감싸는 것이다.

**RuntimeException**의 서브클래스가 아닌 예외는 컴파일러가 반드시 확인한다. "**확인 예외**"라고 부른다.



# Flow control in try/catch blocks

If the **try succeeds** (doRiskyThing() does not throw an exception)





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 never 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: 무조건 실행할 내용을 지정하는 방법

예외 발생 여부와 상관없이 무조건 실행할 코드는 **finally** 블록에 집어넣으면 된다.

**Finally**가 없다면 **turnOvenOff()**를 **try** 블록과 **catch** 블록 모두에 집어넣어야 한다.

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

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



# 실습과제 11-1 Sharpen your pencil

```
public class TestExceptions {

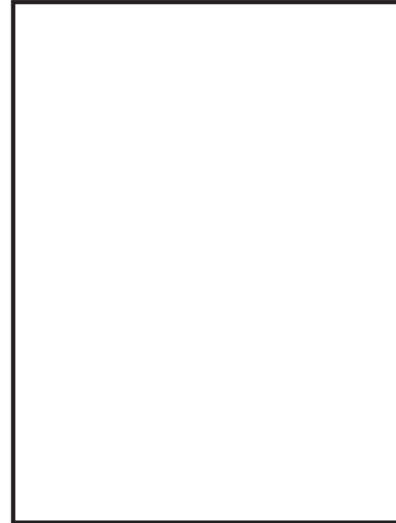
    public static void main(String [] args) {

        String test = "no";
        try {
            System.out.println("start try");
            doRisky(test);
            System.out.println("end try");
        } catch ( ScaryException se) {
            System.out.println("scary exception");
        } finally {
            System.out.println("finally");
        }
        System.out.println("end of main");
    }

    static void doRisky(String test) throws ScaryException {
        System.out.println("start risky");
        if ("yes".equals(test)) {

            throw new ScaryException();
        }
        System.out.println("end risky");
        return;
    }
}
```

Output when test = "no"




Output when test = "yes"




Look at the code to the left.  
What do you think the output of this program would be?  
What do you think it would be if the third line of the program were changed to: **String test = "yes";** ? Assume ScaryException extends Exception.

# 메소드가 2개 이상의 예외를 던질 수도 있다




```
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

예외도 객체다!

던져질 수 있다는 것만 제외하고 특별한 것은 아무 것도 없다!

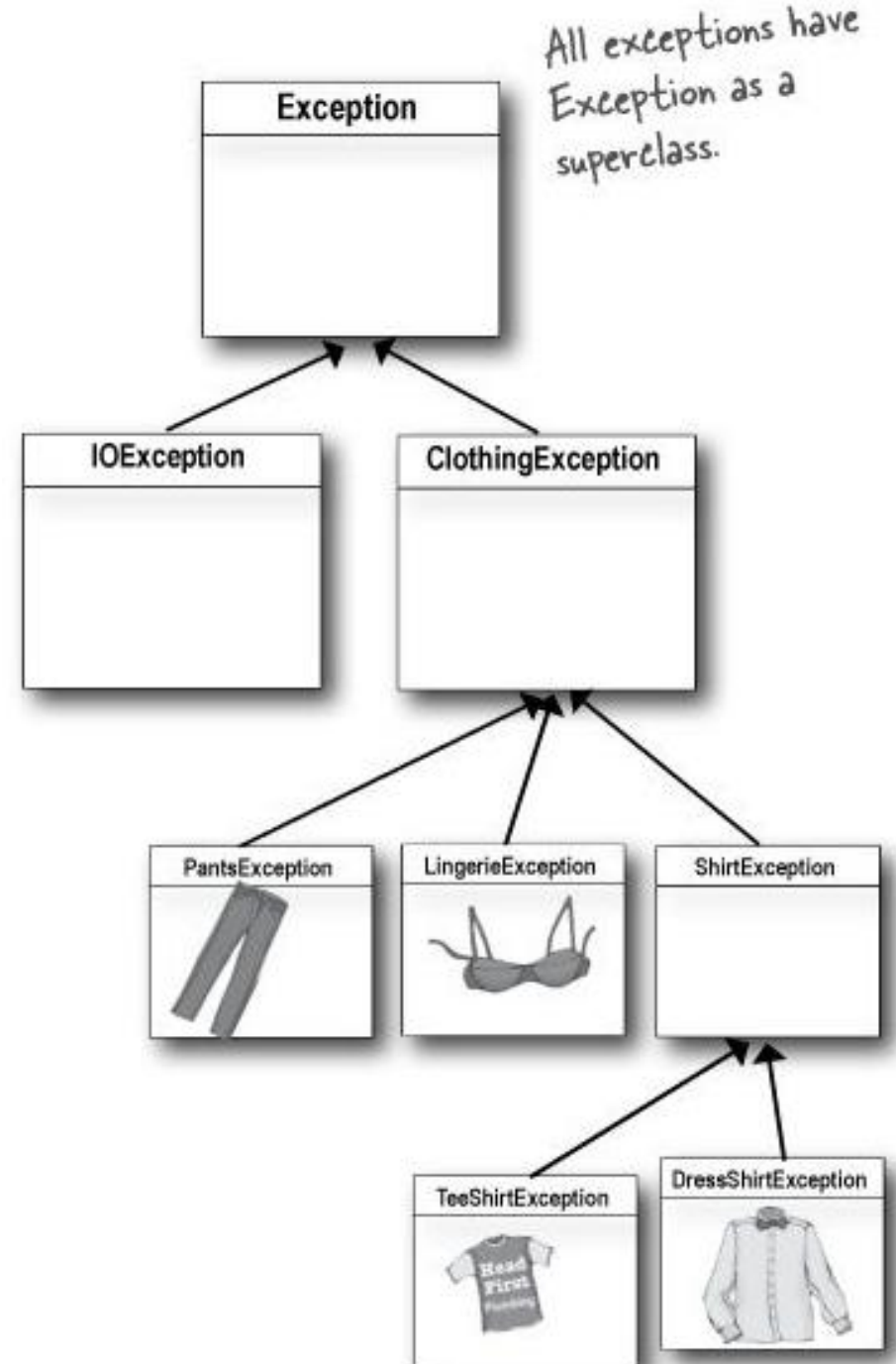
다른 모든 객체와 마찬가지로 Exception도 **다형적으로 참조**될 수 있다.

예를 들어, **PantsException** 객체가 **Exception** 레퍼런스에 할당될 수 있다.

예외에서 장점 중 하나:

던질 수도 있는 모든 가능한 예외를 모두 명시적으로 선언할 필요가 없다.

⇒ 그냥 그 예외의 상위클래스만 선언해도 된다!!





1. Throw하는 예외의 상위 타입을 사용하여 예외를 선언할 수 있다.



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

↑  
ClothingException으로 선언 하면 ClothingException의 모든  
서브클래스도 throw할 수 있게 해준다.  
즉 doLaundry()는 PantsException, LingerieException,  
TeeShirtException, DressShirtException을 명시적으로 선  
언하지 않고도 throw 할 수 있다!

2. 던져진 예외의 상위 타입을 사용하여 예외를 **CATCH**할 수 있다.

```
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

다형적인 catch 블록 하나로 모든 예외를 잡을 수 있다고 해서 꼭 그렇게 해야 하는 것은 아니다.

하나의 커다란 슈퍼 다형성 캐치로 모든 것을 잡을 수 있다고 해서 항상 그래야만 한다는 것을 의미하지는 않는다.

```
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.

특별히 처리해야 하는 각각의 예외에 대해서 서로 다른 **catch** 블록을 사용하라.

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

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

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

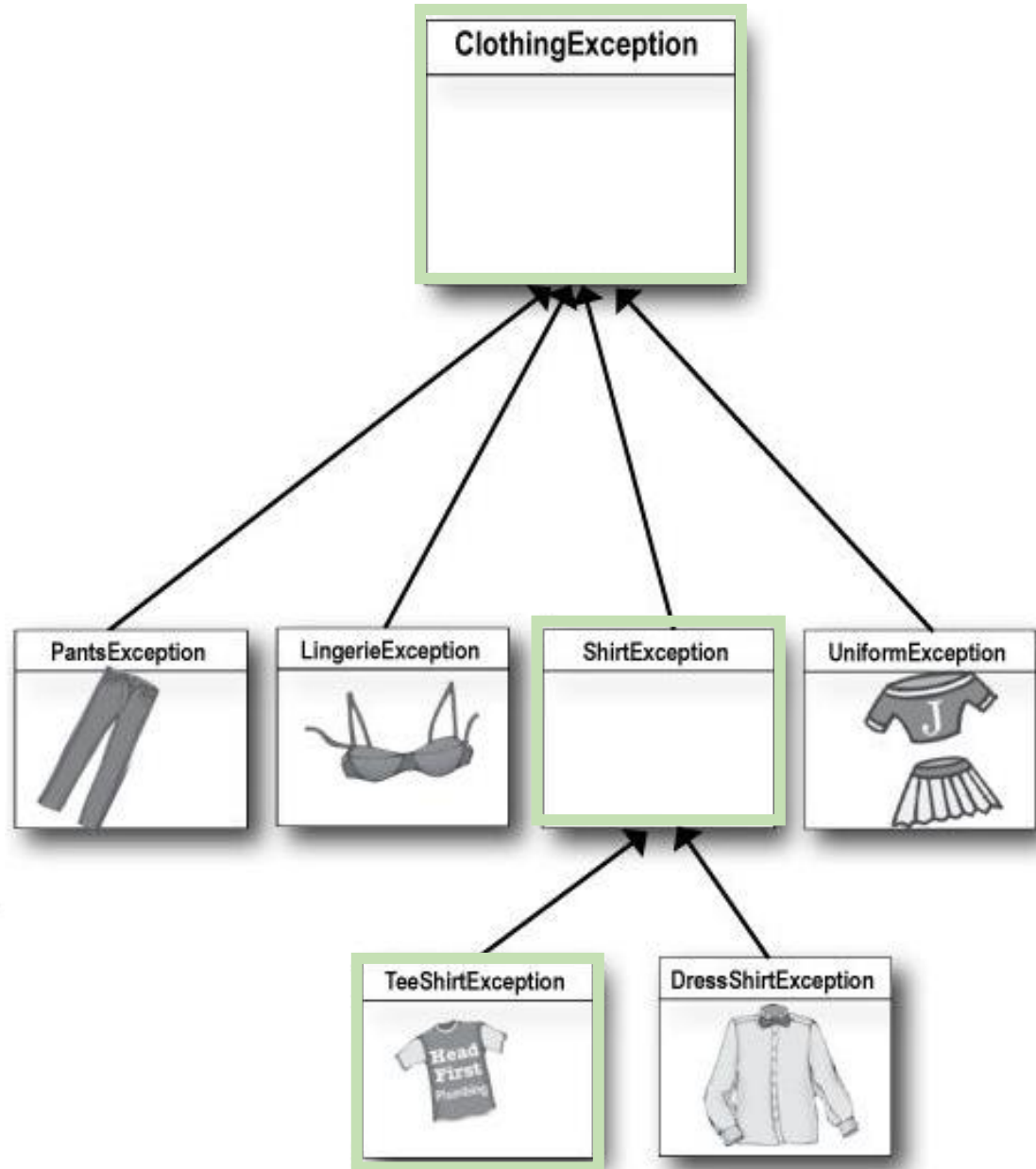
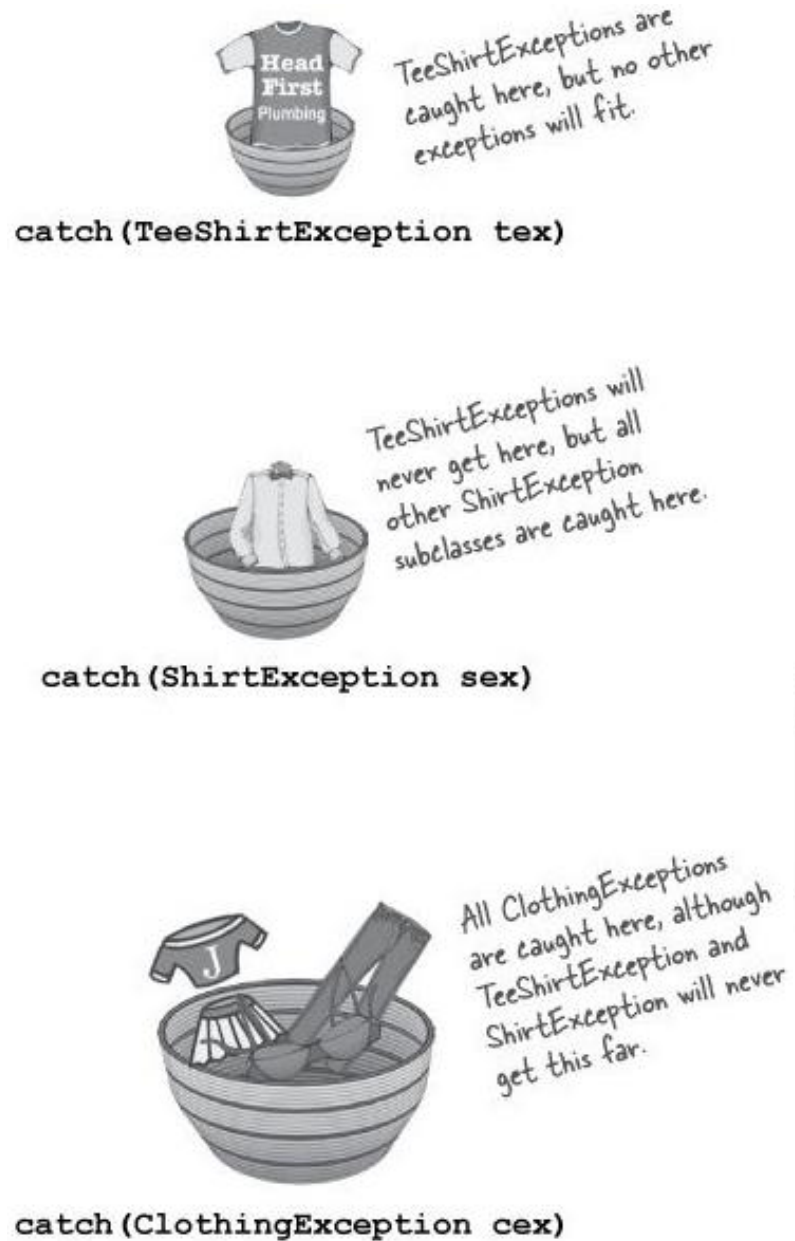
```
    } catch (ClothingException cex) {  
        // recovery from all others
```

```
}
```

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

All other ClothingExceptions are caught here.

# catch 블록을 여러 개 사용할 때는 작은 것부터 큰 것 순서로 나열하라



# 작은 바구니 위에 큰 바구니를 놓을 수 없다

Don't do this!

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



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



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



Size matters when you have multiple catch blocks. The one with the biggest basket has to be on the bottom. Otherwise, the ones with smaller baskets are useless.





# 예외를 처리하고 싶지 않을 때 . . .

just duck it

예외를 처리하고 싶지 않으면 예외를 선언함으로써  
예외 처리를 피할 수 있다.

```
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.



What the...?

There is NO way I'm catching that thing. I'm gettin' out of the way-- somebody behind me can handle it.

# Ducking (by declaring) only delays the inevitable

조만간 누군가는 그것을 처리해야 한다.

**main()**에서도 예외를 회피해 버리면 어떻게 될까?

```
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.

## 1. HANDLE

위험한 콜을 **try/catch**로 감싸라.

```
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)

사용자가 호출하는 위험한 메소드와 동일한 예외를 **throw**한다고 선언하라.

```
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.

그러나 이제 **foo()** 메소드를 누가 호출하든지 간에 **Handle** or **Declare** 법칙을 따라야 한다.

만일 **foo()**가 예외를 회피하고(예외를 선언함으로써) **main()**이 **foo()**를 호출하게 되면 **main()**은 반드시 그 예외를 처리해야만 한다.

```
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 . . .

시작부분에서 **JavaSound** 코드에서 **Sequencer** 객체를 만들었다.

그러나 **Midi.getSequencer()** 메소드가 확인 예외 (**MidiUnavailableException**)를 선언하기 때문에 컴파일시킬 수 없었다.

⇒ 그러나 이제 콜을 **try/catch**로 감싸서 문제를 해결할 수 있다.

```
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!



# 예외 규칙

1. try없이 catch 또는 finally만을 사용할 수 없다.

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

2. try와 catch 사이에 코드를 집어넣을 수 없다.

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

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

3. try 뒤에는 반드시 catch 또는 finally가  
와야 한다.

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

4. try 뒤에 (no catch) finally만 있다면 예  
외를 선언해야 한다.

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

코드 키친



# 실제로 소리를 만들어보자

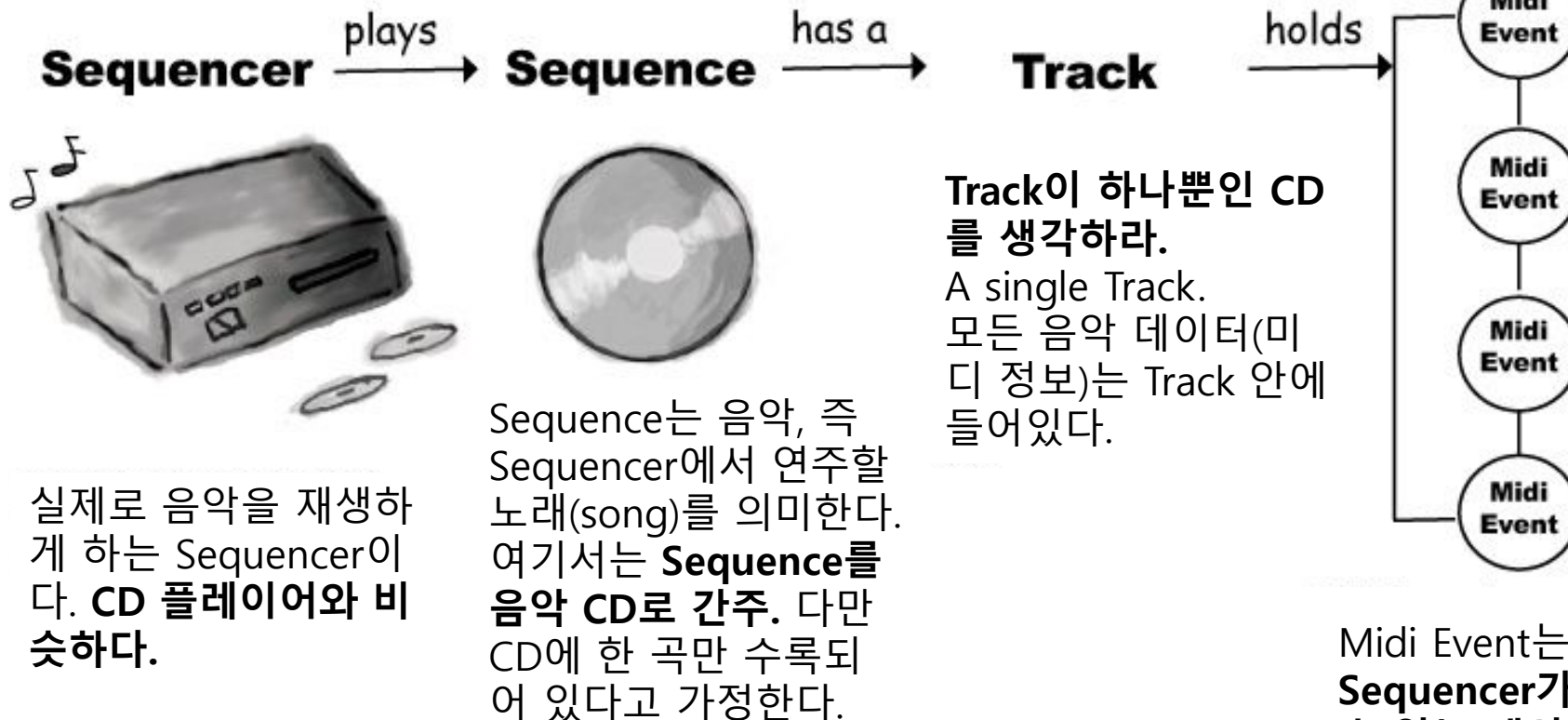
You need **FOUR** things:

① 음악을 재생하는 것

② 재생할 음악

③ Sequence에서  
실제 정보가 들어  
있는 부분

④ 실제 음악 정보: 연주  
할 음표, 지속 시간 등



# And you need **FIVE** steps:

- ① Get a **Sequencer** and open it

```
Sequencer player = MidiSystem.getSequencer();  
player.open();
```

- ② Make a new **Sequence**

```
Sequence seq = new Sequence(timing, 4);
```

- ③ Get a new **Track** from the Sequence

```
Track t = seq.createTrack();
```

- ④ Fill the Track with **MidiEvents** and give the Sequence to the Sequencer

```
t.add(myMidiEvent1);  
player.setSequence(seq);
```



# 첫 번째 사운드 플레이어 앱

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

← Don't forget to import the midi package

```
public class MiniMiniMusicApp {
```

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

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

```
        try {
```

```
            ① Sequencer player = MidiSystem.getSequencer();  
               player.open();
```

```
            ② Sequence seq = new Sequence(Sequence.PPQ, 4);
```

← get a Sequencer and open it  
(so we can use it... a Sequencer  
doesn't come already open)

← Don't worry about the arguments to the  
Sequence constructor. Just copy these (think  
of 'em as Ready-bake arguments).

③ `Track track = seq.createTrack();`

← Ask the Sequence for a Track. Remember, the Track lives in the Sequence, and the MIDI data lives in the Track.

④ `ShortMessage a = new ShortMessage();  
a.setMessage(144, 1, 44, 100);  
MidiEvent noteOn = new MidiEvent(a, 1);  
track.add(noteOn);`

`ShortMessage b = new ShortMessage();  
b.setMessage(128, 1, 44, 100);  
MidiEvent noteOff = new MidiEvent(b, 16);  
track.add(noteOff);`

Put some MidiEvents into the Track. This part is mostly Ready-bake code. The only thing you'll have to care about are the arguments to the `setMessage()` method, and the arguments to the `MidiEvent` constructor. We'll look at those arguments on the next page.

`player.setSequence(seq);` ← Give the Sequence to the Sequencer (like putting the CD in the CD player)

`player.start();` ← `start()` the Sequencer (like pushing PLAY)

```
} catch (Exception ex) {  
    ex.printStackTrace();  
}
```

```
} // close play
```

```
} // close class
```



## 실습과제 11-2

'Your very first sound player app'을 다양하게 변형시켜 적어도 3개의 소리를 듣고 그냥 글로 표현해 보시오.

두윽~웅!



# Making a MidiEvent (song data)

MidiEvent는 노래의 한 부분에 대한 지시사항이다.

- 일련의 MidiEvent는 종이에 쓰여진 **악보**와 같다.
- 우리가 사용하게 되는 대부분의 MidiEvent는 '**할 일**'과 '**그 일을 할 순간**'을 지정하는 것이다.
- '**그 일을 할 순간**'을 지정하는 것이 더 중요하다. 음악에 타이밍이 중요하듯이.
- "이 음표 다음에는 이 음표가 오고 ..." 하는 식으로.

MidiEvent는 연주를 시작하는 시기(**NOTE ON**)와 끝내는 시기(**NOTE OFF**)도 지정해줘야만 한다.

MidiEvent = Message + 메시지가 '시작'되어야 할 순간

Message => 무엇(What)을 해야 할지를 말해준다.

MidiEvent => 그 일을 언제(When) 할지를 말해준다:

1. **Message**를 만든다:

```
ShortMessage a = new ShortMessage();
```

2. **Message**에 지시사항을 집어넣는다.

```
a.setMessage(144, 1, 44, 100);
```

3. **Message**를 이용하여 새로운 **MidiEvent**를 만든다.

```
MidiEvent noteOn = new MidiEvent(a, 1);
```

4. **MidiEvent**를 **Track**에 추가한다.

```
track.add(noteOn);
```

# MIDI message: the heart of a MidiEvent

## Anatomy of a message

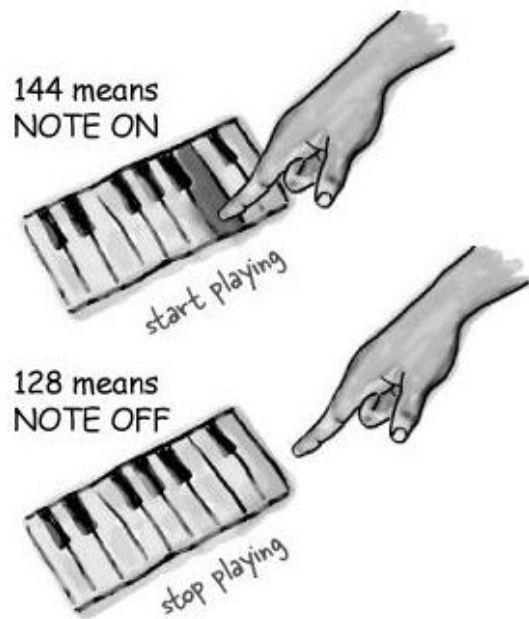
setMessage()의 첫 번째 인자는 항상 메시지 '타입'이 온다.

- 나머지 세 개의 인자는 이 메시지 타입에 따라 내용이 달라진다.

*message type*  
*channel*      *note to play*  
*velocity*  
`a.setMessage(144, 1, 44, 100);`

The last 3 args vary depending on the message type. This is a **NOTE ON** message, so the other args are for things the Sequencer needs to know in order to play a note.

## 1. Message type



## 2. Channel

밴드에서 **연주자**처럼 간주하라.

Channel 1 -> 키보드 연주자

Channel 9 -> 드러머

...

## 3. Note to play

A number from 0 to 127, going from low to high notes.

0에서 127 사이의 숫자로, 낮은 음조에서 높은 음조로 올라간다.



## 4. Velocity

키를 얼마나 빨리 그리고 얼마나 세게 눌렀나?

0 -> 매우 부드럽게

100 -> 적절



# Change a message

1. 음을 바꿔보자: 0 ~ 127 사이의 숫자

**a.setMessage(144, 1, 20, 100);**



2. 지속시간을 바꿔보자.

먼저 연주 끝내기 이벤트로 고친 후:

**b.setMessage( 128, 1, 44, 100);**

**MidiEvent noteOff = new MidiEvent( b, 3);**



3. 악기를 바꿔보자.

악기 변경 메시지는 '192'.  
세 번째 인자는 실제 악기를 나타낸다 (0에서 127 사이의 숫자를 시도해보라)



**first.setMessage(192, 1, 102, 0);**

change-instrument message  
in channel 1 (musician 1)  
to instrument 102

# 실습과제 11-3

Using command-line args to experiment with sounds

```
1 package ch11_3;
2 import javax.sound.midi.*;
3
4
5 public class MiniMiniMusicCmdLine { // this is the first one
6
7     public static void main(String[] args) {
8         MiniMiniMusicCmdLine mini = new MiniMiniMusicCmdLine();
9         if (args.length < 2) {
10             System.out.println("Don't forget the instrument and note args");
11         } else {
12             int instrument = Integer.parseInt(args[0]);
13             int note = Integer.parseInt(args[1]);
14             mini.play(instrument, note);
15         }
16     }
17 }
18
19 public void play(int instrument, int note) {
20
21     try {
22
23         Sequencer player = MidiSystem.getSequencer();
24         player.open();
25
26         Sequence seq = new Sequence(Sequence.PPQ, 4);
27         Track track = seq.createTrack();
28
```

```
29         MidiEvent event = null;
30
31         ShortMessage first = new ShortMessage();
32         first.setMessage(192, 1, instrument, 0);
33         MidiEvent changeInstrument = new MidiEvent(first, 1);
34         track.add(changeInstrument);
35
36         ShortMessage a = new ShortMessage();
37         a.setMessage(144, 1, note, 100);
38         MidiEvent noteOn = new MidiEvent(a, 1);
39         track.add(noteOn);
40
41         ShortMessage b = new ShortMessage();
42         b.setMessage(128, 1, note, 100);
43         MidiEvent noteOff = new MidiEvent(b, 16);
44         track.add(noteOff);
45         player.setSequence(seq);
46         player.start();
47         // new
48         Thread.sleep(5000);
49         player.close();
50         System.exit(0);
51
52     } catch (Exception ex) {ex.printStackTrace();}
53 } // close play
54
55 } // close class
```

# (1) 명령 프롬프트에서 실행

```
명령 프롬프트
D:\practice\java\Chapter11\src>cd ch11_3
D:\practice\java\Chapter11\src\ch11_3>dir
D 드라이브의 볼륨: SAMKEUN1901
볼륨 일련 번호: 2071-8E4E

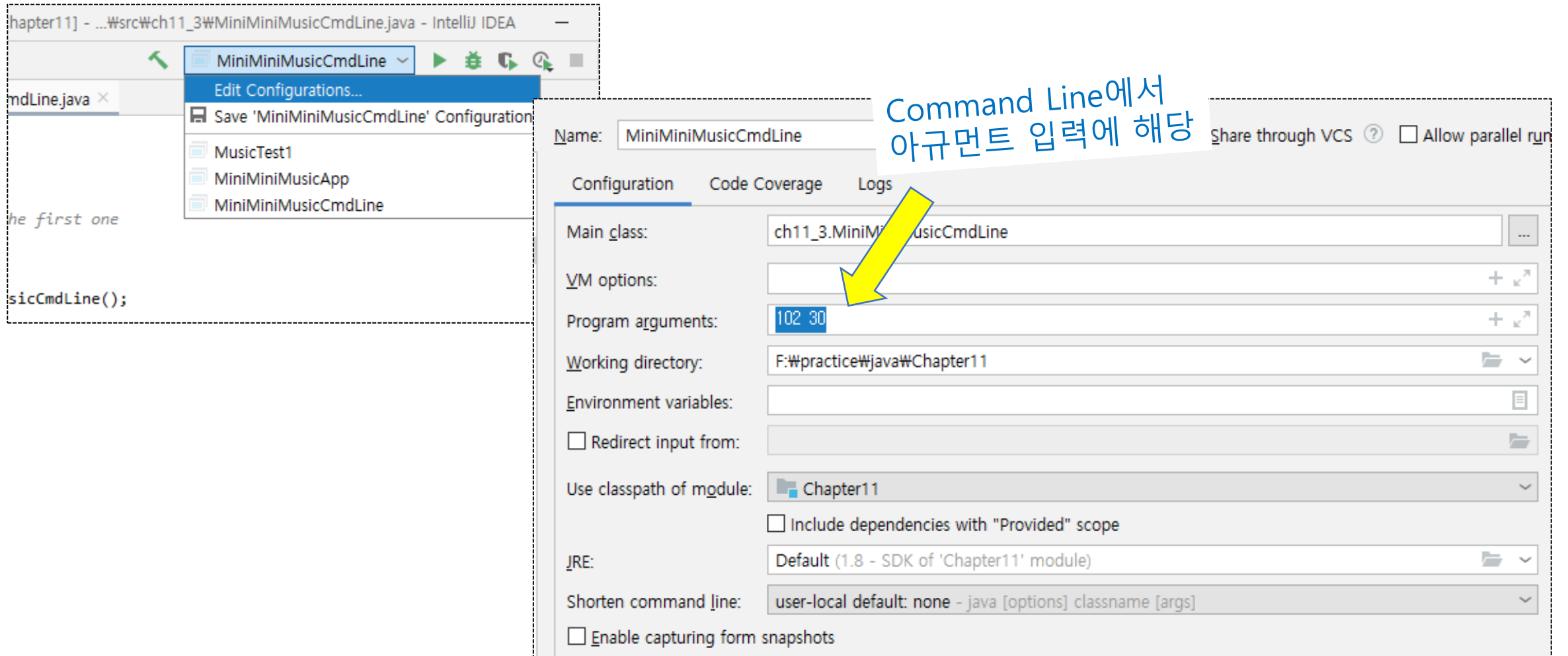
D:\practice\java\Chapter11\src\ch11_3 디렉터리

2019-03-21 오후 07:45 <DIR>          .
2019-03-21 오후 07:45 <DIR>          ..
2019-03-21 오후 07:48                1,687 MiniMiniMusicCmdLine.java
                1개 파일                1,687 바이트
                2개 디렉터리    5,264,457,728 바이트 남음

D:\practice\java\Chapter11\src\ch11_3>cd ..
D:\practice\java\Chapter11\src>javac -d . ch11_3\MiniMiniMusicCmdLine.java
D:\practice\java\Chapter11\src>java ch11_3.MinMiniMusicCmdLine 80 20
D:\practice\java\Chapter11\src>
```

Command Line에서 아규먼트  
를 입력해야 하는 프로그램

## 또는 (2) IntelliJ IDEA에서 실행





# 실습과제 11-4 Code Magnets

A working Java program is scrambled up on the fridge. Can you reconstruct all the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

```
File Edit Window Help ThrowUp
% java ExTestDrive yes
thaws

% java ExTestDrive no
throws
```

```
System.out.print("r");
try {
    System.out.print("t");
    doRisky(test);
    System.out.println("s");
    System.out.print("o");
} finally {

class MyEx extends Exception { }

public class ExTestDrive {

    System.out.print("w");
    if ("yes".equals(t)) {

        System.out.print("a");
        throw new MyEx();
    } catch (MyEx e) {

static void doRisky(String t) throws MyEx {
    System.out.print("h");

public static void main(String [] args) {
    String test = args[0];
```

