中原大學
Chung Yuan Christian University

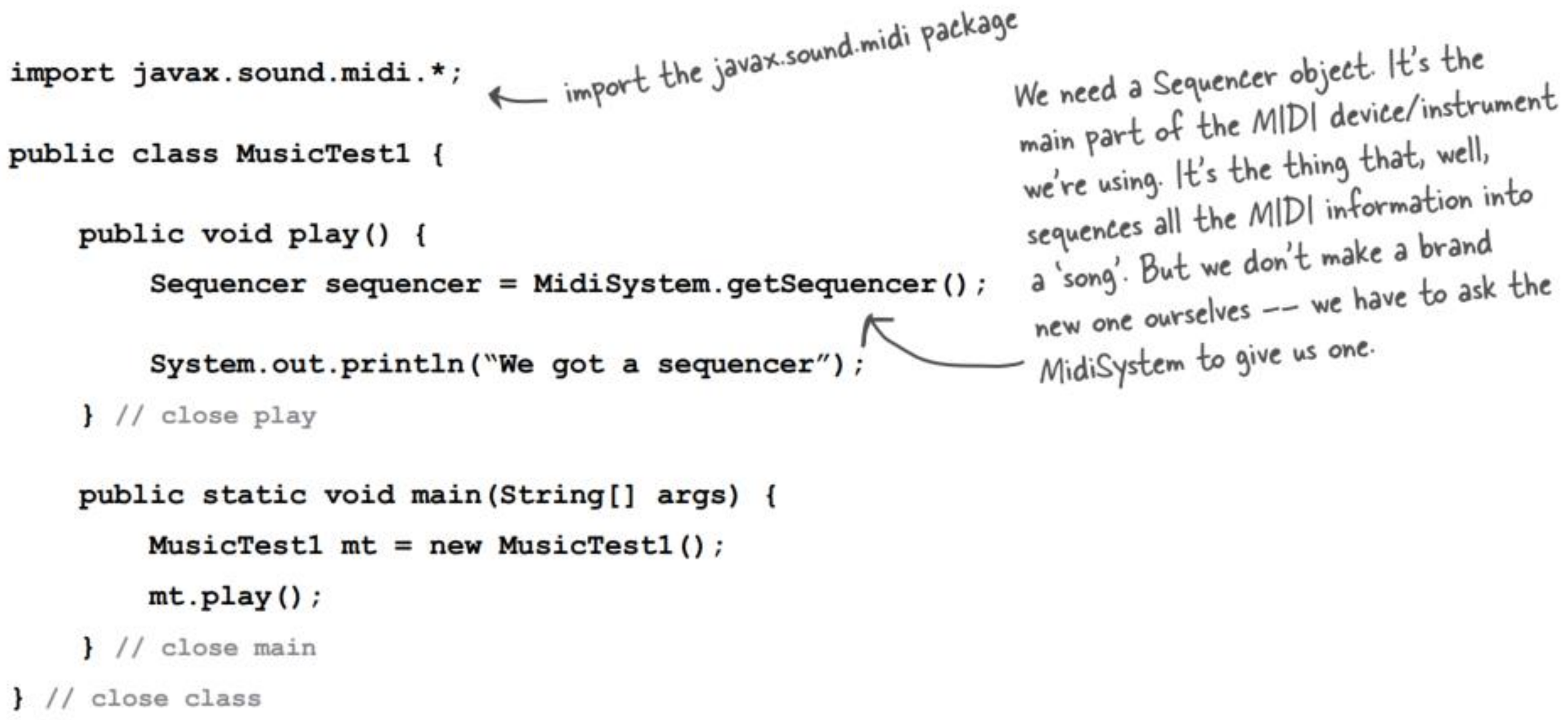
Chapter 11

Exception handling

- No matter how good a programmer you are, you can't control everything. Things can go wrong.
- You want to load a file but the file isn't there. You want to messaging others but the server is down.
- When you invoke methods written by others, how do you know that method is risky or not?
- When you write a method (or class) that might be used by others, how to let them know that bad things might happen?
- Some codes that you just can't guarantee will work at runtime

- Here we try to play MIDI for demonstration
- To play MIDI, you can use JavaSound API
- JavaSound API is a collection of classes and interfaces to play sound.
- MIDI stands for Musical Instrument Digital Interface, and is a standard protocol for getting different kinds of electronic sound equipment to communicate.

- Before we can get any sound to play, we need a Sequencer object
- A sequencer can do a lot of different things, but here we're using it strictly as a playback device
- The Sequencer class is in the javax.sound.midi package
- https://docs.oracle.com/javase/8/docs/api/ search MidiSystem

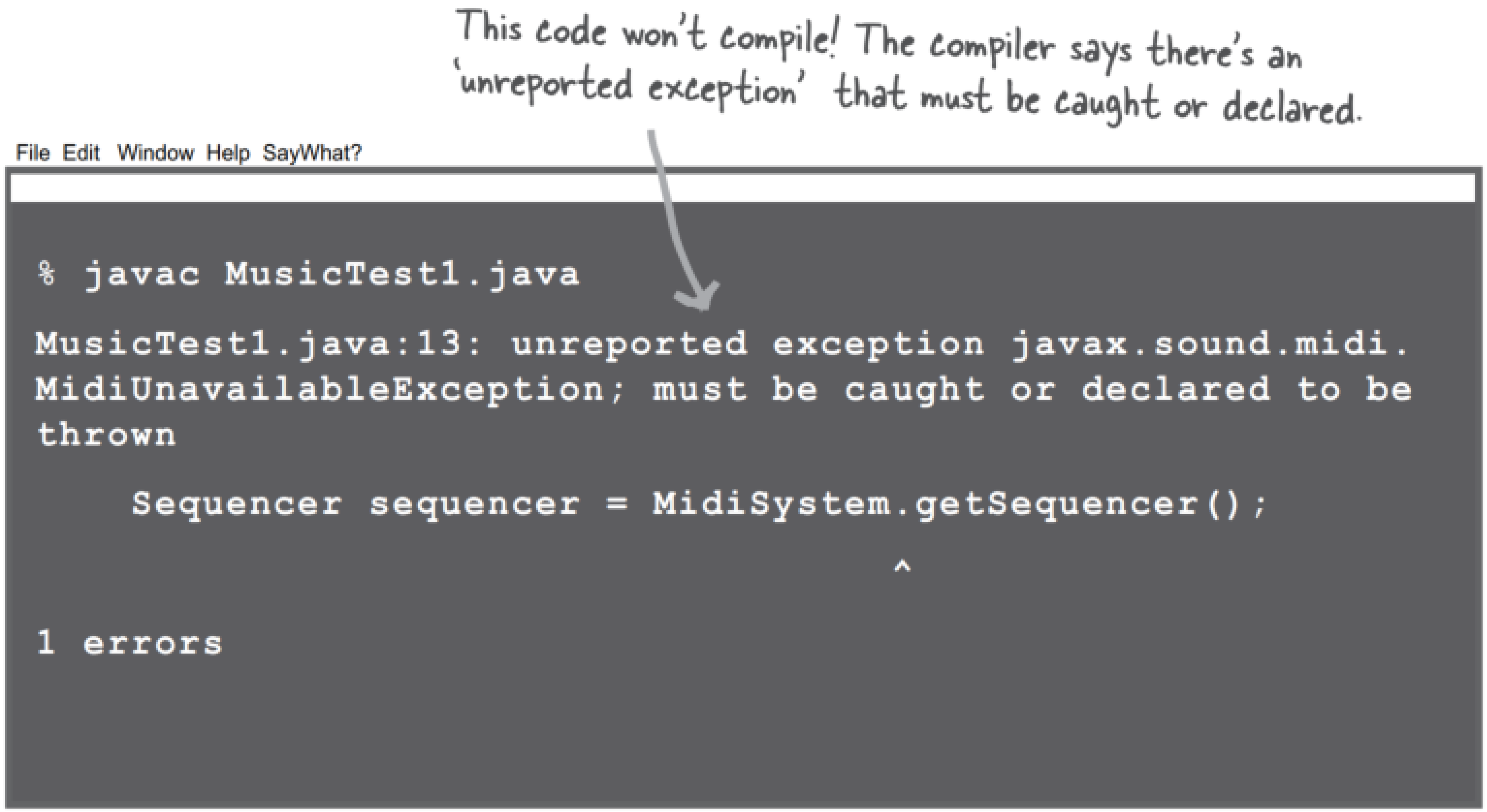


```
import javax.sound.midi.*;

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.

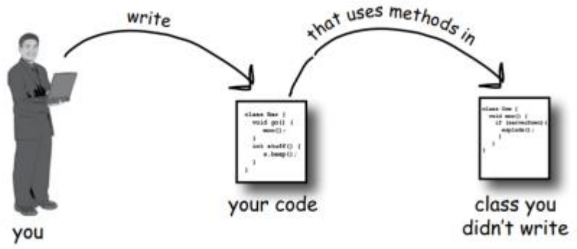
- Oops! Something wrong!


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 is risky?

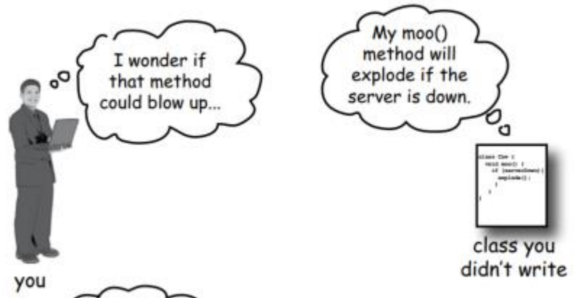
① **Let's say you want to call a method in a class that you didn't write.**



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.**


I wonder if that method could blow up...
My moo() method will explode if the server is down.
you
class you didn't write

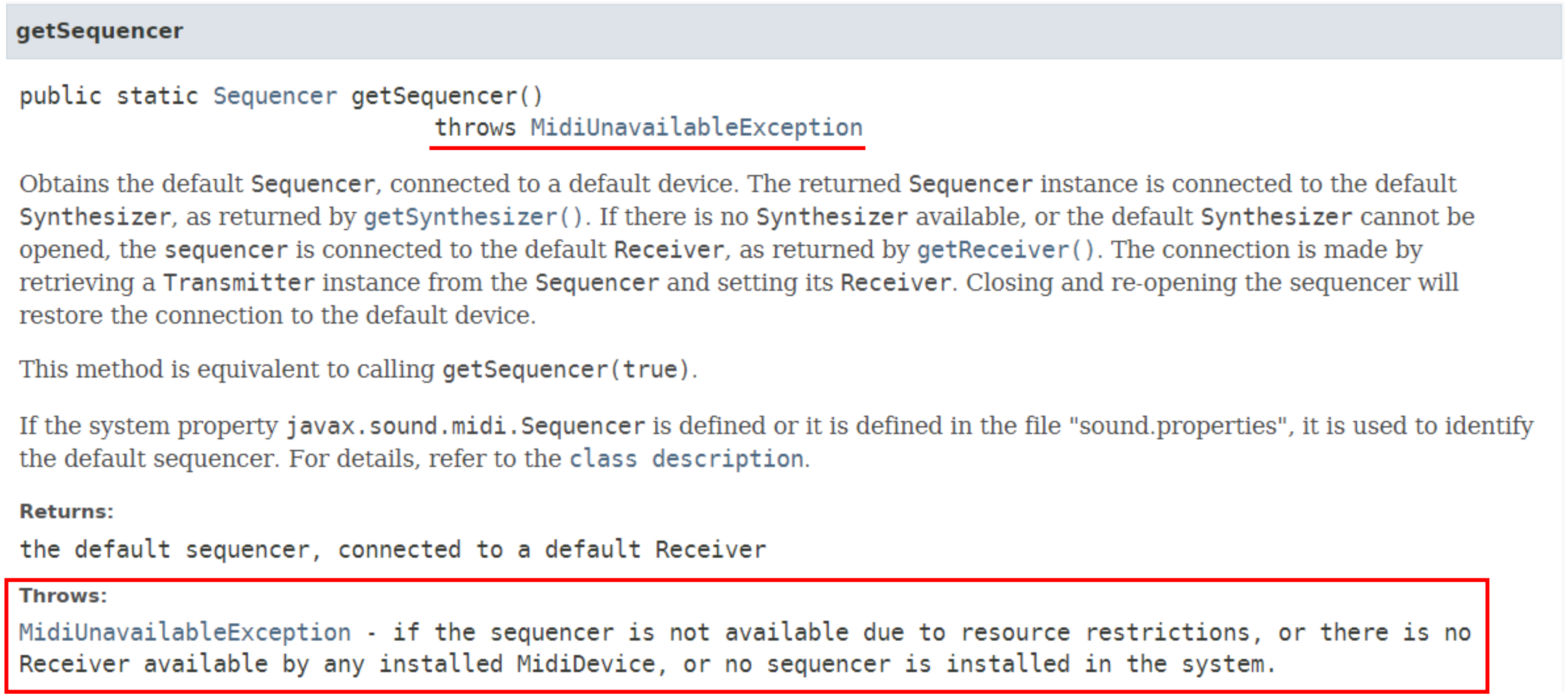
④ **You then write code that can handle the failure if it *does* happen. You need to be prepared, just in case.**

Now that I know, I can take precautions.
write safely
you
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's based on you knowing that the method you're calling is risky, 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.

- A frequently asked question is: why we need exception-handling?
- For example, we can use a simple "if condition" to check possible failures.
- Exception support is not necessary to write valid programs. It just makes doing so a lot easier.
- Suppose you have method1 calling method2 with some input. Suddenly, method2 fails for some reason. What will you do? You might handle the failure within method2, and then return that error to method1.
- You see the problem - a developer can ignore (or not be aware of) your return and go on.
- An exception needs to be acknowledged in some way - it can't be silently ignored without actively putting something in place to do so. You must deal with them at **some** level, or they will terminate your program

- How do you know if a method throws an exception? In Java API documentation, you'll find a ***throws*** clause in the risky method's declaration.



getSequencer

```
public static Sequencer getSequencer()
                              throws MidiUnavailableException
```

Obtains the default `Sequencer`, connected to a default device. The returned `Sequencer` instance is connected to the default `Synthesizer`, as returned by `getSynthesizer()`. If there is no `Synthesizer` available, or the default `Synthesizer` cannot be opened, the `sequencer` is connected to the default `Receiver`, as returned by `getReceiver()`. The connection is made by retrieving a `Transmitter` instance from the `Sequencer` and setting its `Receiver`. Closing and re-opening the sequencer will restore the connection to the default device.

This method is equivalent to calling `getSequencer(true)`.

If the system property `javax.sound.midi.Sequencer` is defined or it is defined in the file "sound.properties", it is used to identify the default sequencer. For details, refer to the `class description`.

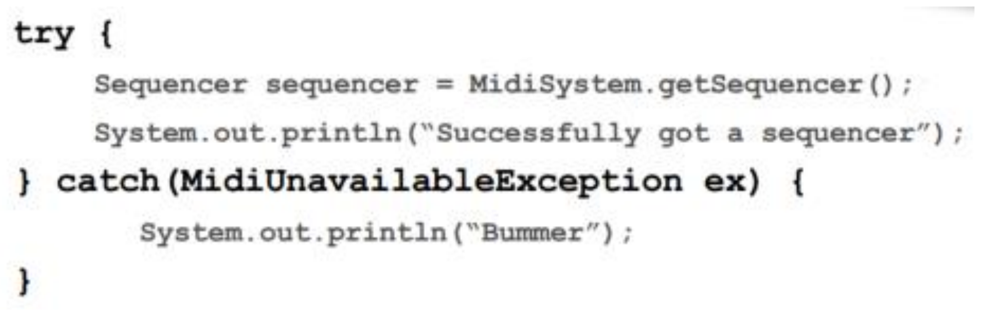
Returns:

the default sequencer, connected to a default Receiver

Throws:

`MidiUnavailableException` - if the sequencer is not available due to resource restrictions, or there is no Receiver available by any installed MidiDevice, or no sequencer is installed in the system.

- For those methods which throw exceptions, the compiler complains if you do not handle them.
- 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



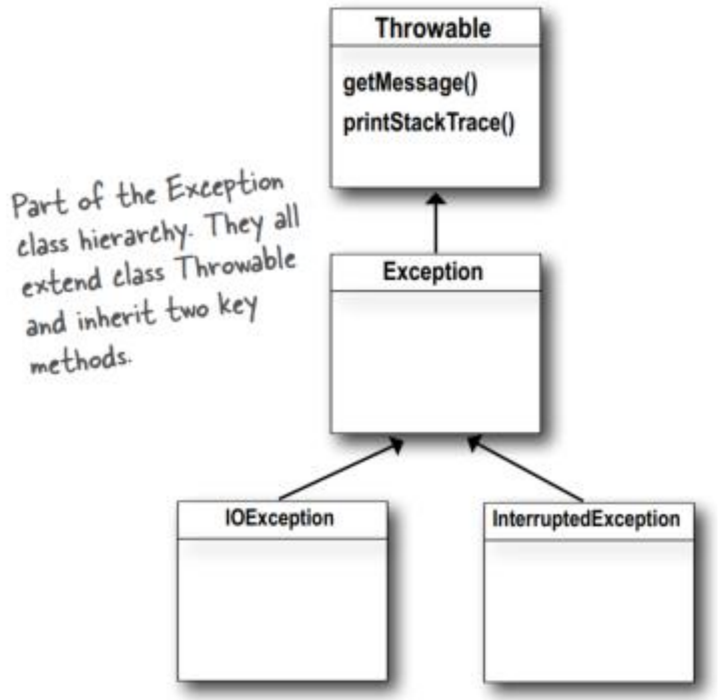
```
try {
    Sequencer sequencer = MidiSystem.getSequencer();
    System.out.println("Successfully got a sequencer");
} catch(MidiUnavailableException ex) {
      System.out.println("Bummer");
}
```

- An exception is an object of type Exception
- An object of type Exception can be an instance of any subclass of Exception.
- What you catch is an ***object***

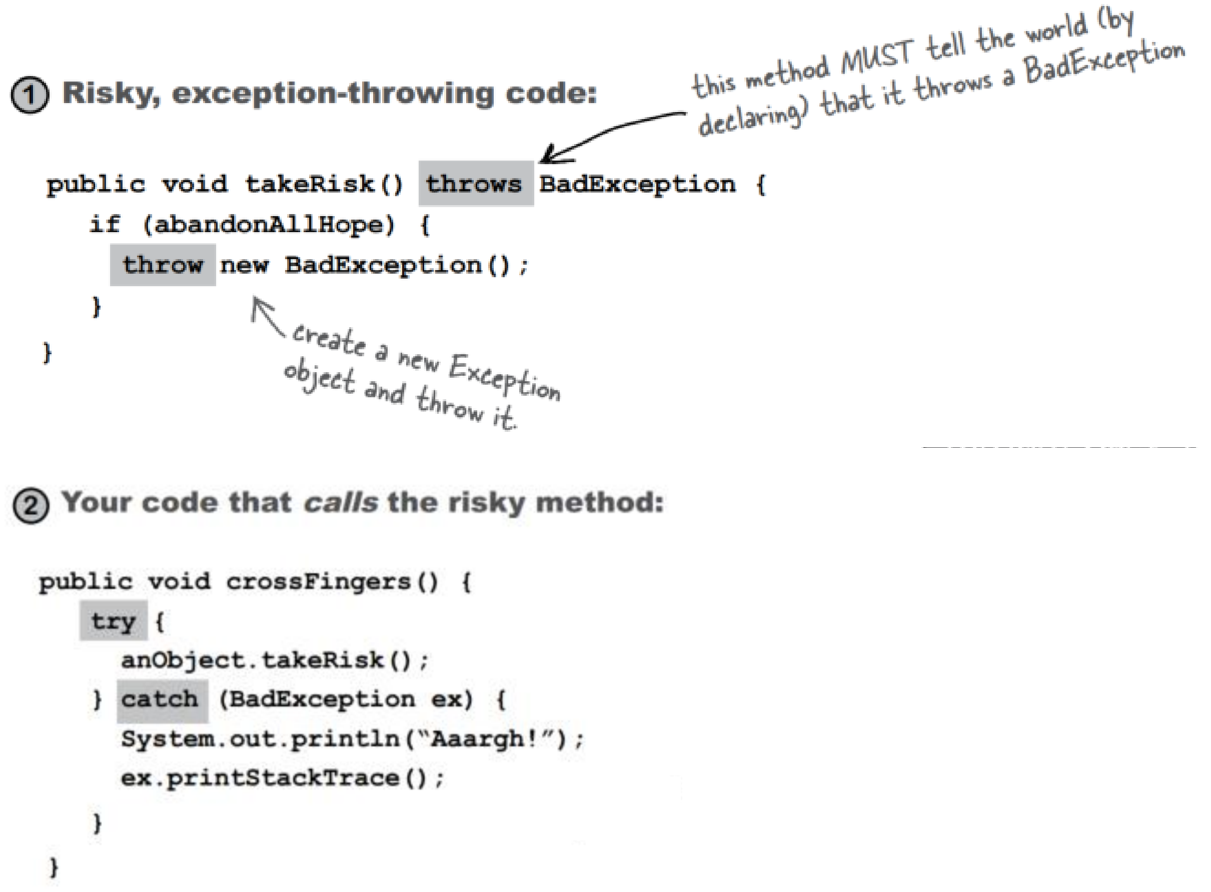
```
try {
    // do risky thing

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

it's just like declaring a method argument

This code only runs if an Exception is thrown.


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

- When your code calls a risky method—a method that declares an exception—it's the risky method that throws the exception back to the caller
- When somebody writes code that could throw an exception, they must declare the exception.



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();
    }
}
```

- The compiler guarantees:
- If you throw an exception in your code you must declare it using the throws keyword in your method declaration.
- 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.
- Exceptions that are subclasses of RuntimeException are NOT checked by the compiler.

* Exception/RuntimeException are also called checked/unchecked exceptions respectively.

- RuntimeExceptions can be thrown anywhere, with or without throws declarations or try/catch blocks.
- Why the compiler ignores RuntimeExceptions?
- Most RuntimeExceptions are ***unpredictable***. They 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 WANT RuntimeExceptions to happen at development and testing time.
- A try/catch is for handling exceptional situations, not flaws in your code. At least print out a message to the user and a stack trace, so somebody can figure out what happened.
- Quiz: can you come up with some RuntimeExceptions?

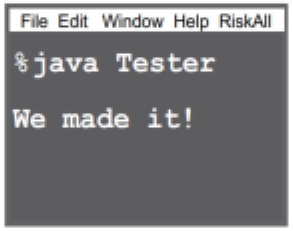
If the try succeeds

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

```
try {
①   Foo f = x.doRiskyThing();
    int b = f.getNum();

} catch (Exception ex) {
    System.out.println("failed");
}
② → System.out.println("We made it!");
```

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

The code in the catch block never runs.



```
%java Tester
We made it!
```

If the try fails

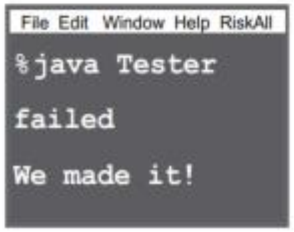
(because doRiskyThing() *does* throw an exception)

```
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 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.

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().



```
%java Tester
failed
We made it!
```

- Summary
- A method can throw an exception when something fails at runtime
- An exception is an object of type Exception
- An exception of type RuntimeException does not have to be wrapped in a try/catch block.
- Put the keywords throw followed by a new exception object when declaring a method.
- The catch block is used to put codes that are handling/recovering that exception.

- finally: for the things you want to do no matter what happened in the try/catch block
- 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. That makes your code ugly.

- If the try or catch block has a return statement, finally will still run! Flow jumps to the finally, then back to the return.

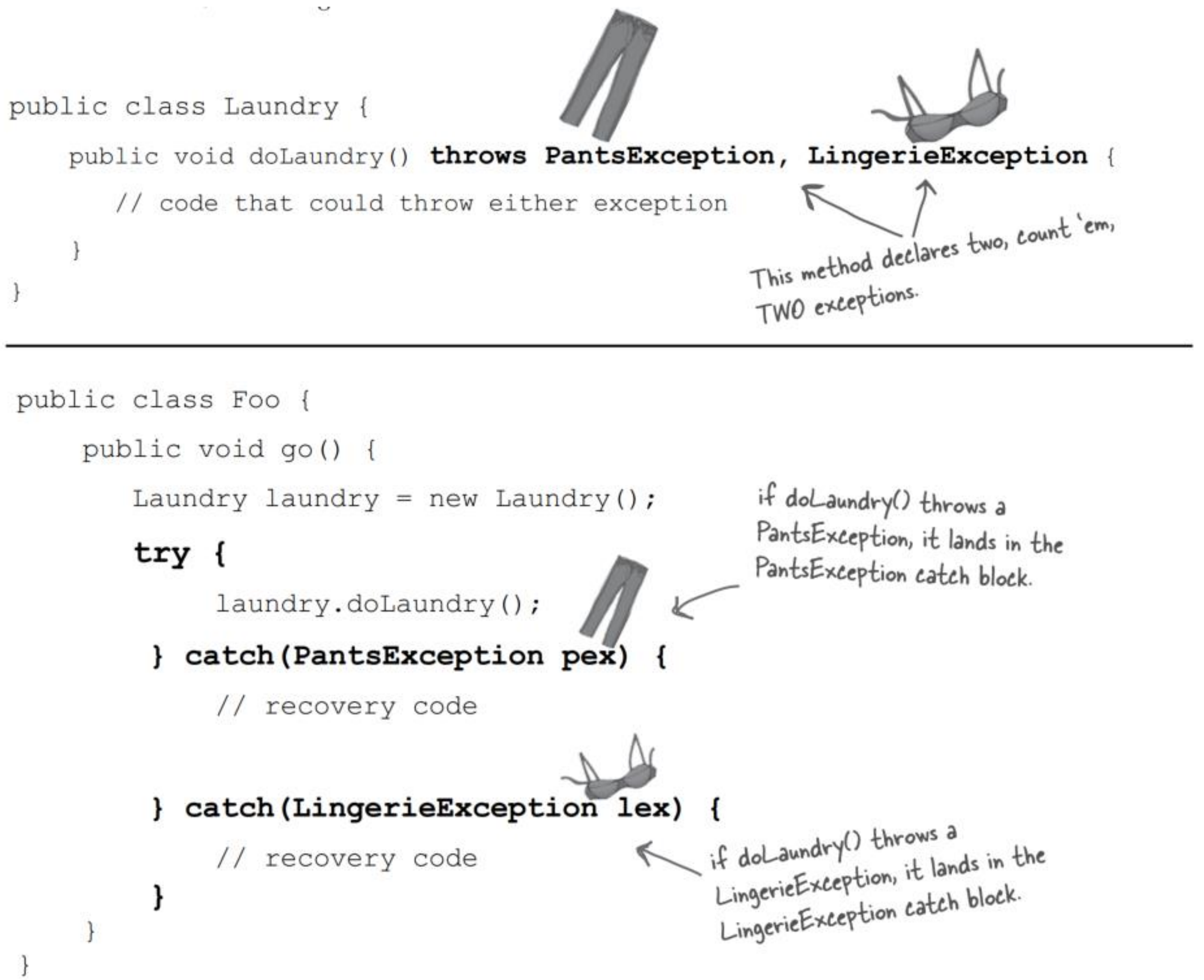
```java
public static void main(String[] args) {
    System.out.println(Test.test());
}

public static int test() {
    try {
        return 0;
    }
    finally {
        System.out.println("finally trumps return.");
    }
}
```

- See https://stackoverflow.com/a/2902578 for more explanations.

- What’s the output when test = “no”? What’s the output when test = “yes”?

```
public class TestExceptions {

  public static void main(String [] args) {

    String test = “no”;
    try {
1      System.out.println(“start try”);
       doRisky(test);
       System.out.println(“end try”);  2
    } catch ( ScaryException se) {
        System.out.println(“scary exception”);  3
    } finally {
        System.out.println(“finally”);  4
    }
    System.out.println(“end of main”); 5
  }

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

      throw new ScaryException();
    }
    System.out.println(“end risky”);  7
    return;
  }
}
```

- A method can throw more than one exception
- You have to handle all exceptions thrown by that method



```
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.

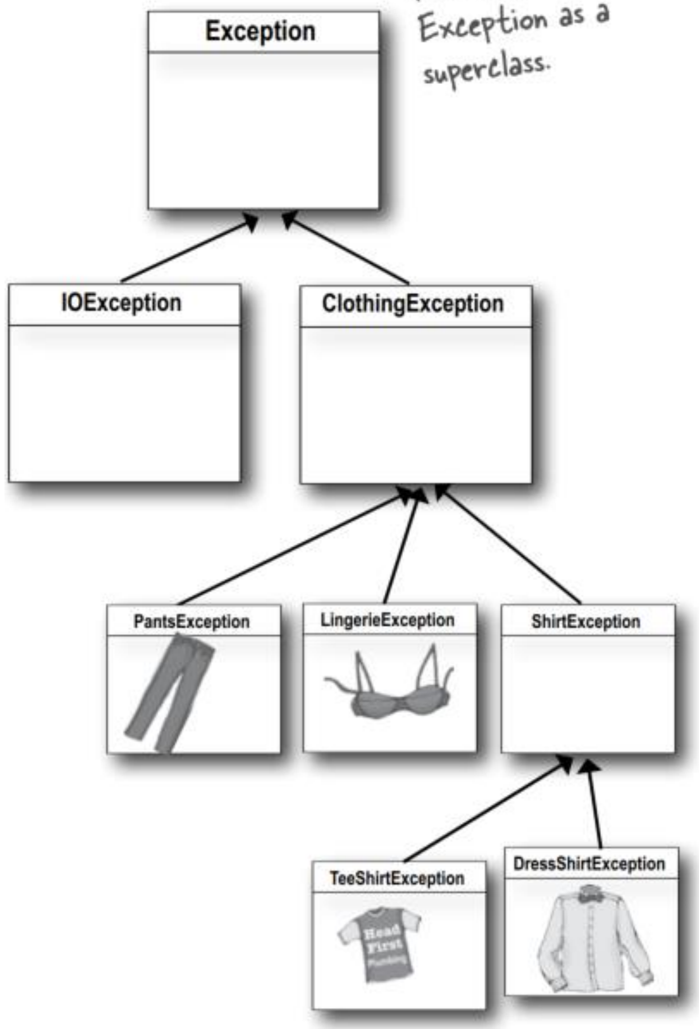
- Exceptions are polymorphic.

```
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.

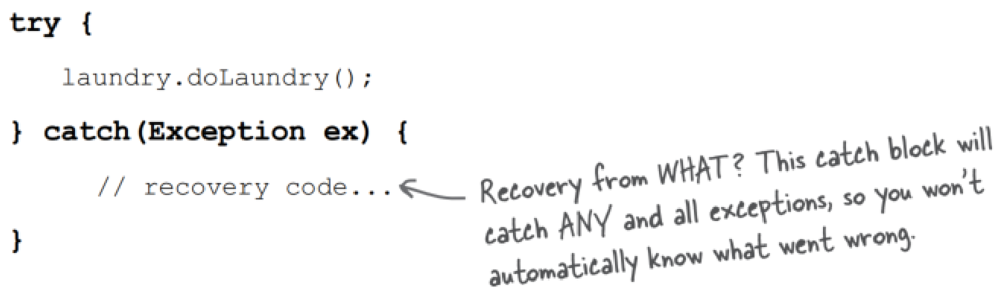
```
try {

    laundry.doLaundry();

} catch(ClothingException cex) {

     // recovery code

}
```

can catch any ClothingException subclass



All exceptions have Exception as a superclass.

- You CAN catch everything with one big super polymorphic catch doesn't mean you SHOULD.



```
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.

- You can write a different catch block for each exception that you need to handle uniquely (if they have to be handled differently)

- Multiple catch blocks must be ordered from smallest to biggest

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

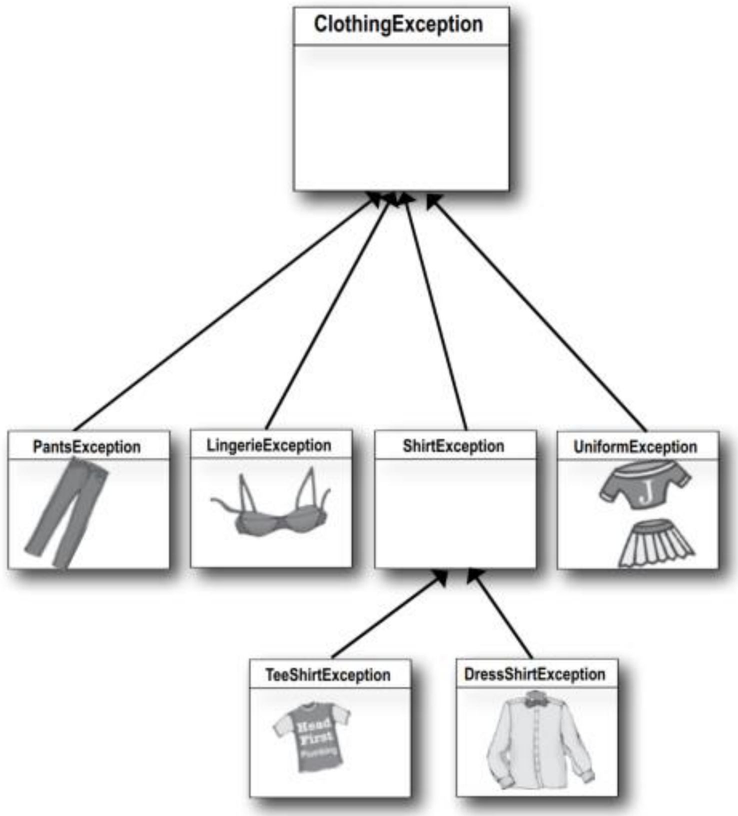
`catch(TeeShirtException tex)`

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

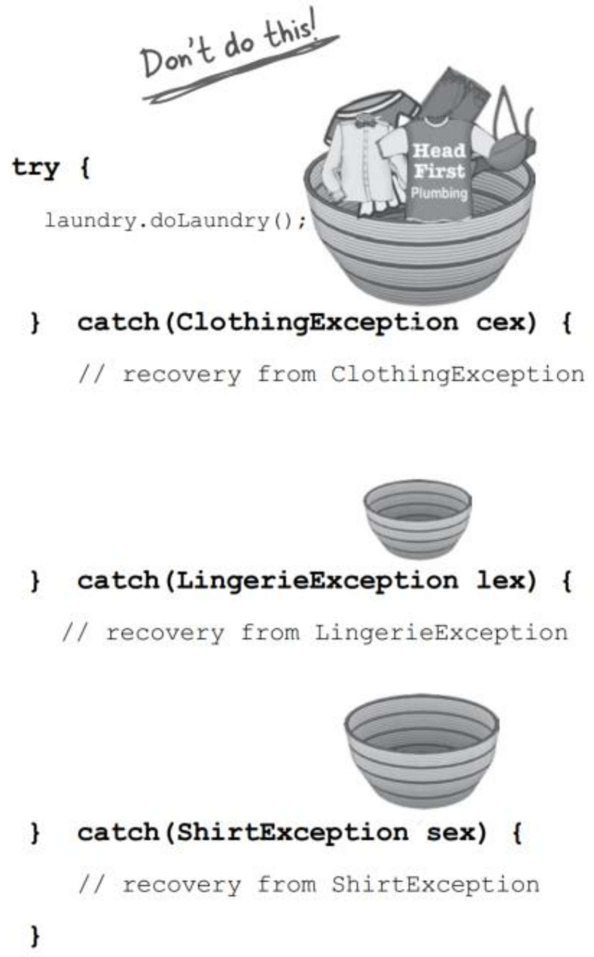
`catch(ShirtException sex)`

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

`catch(ClothingException cex)`



- 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. (The compiler will complain)



```
try {
   laundry.doLaundry();

 }  catch(ClothingException cex) {
    // recovery from ClothingException

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

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

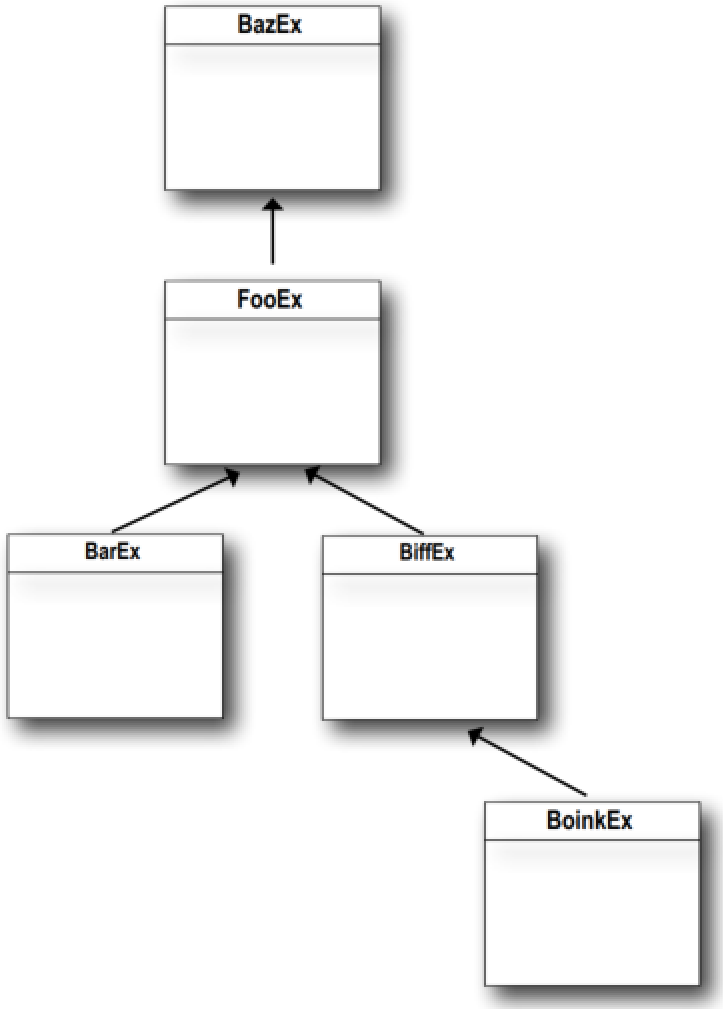
- If you define a try block you have to define
- One finally block, or
- One or more catch blocks, or
- One or more catch blocks and one finally block

* After Java 7, in some situation you can write a try block alone. It is called ***try-with-resources***. You can study this part on your own.

- Exercise 1: suppose the following code is legal. Draw two different class diagrams that can accurately reflect the Exception classes.

```
try {
  x.doRisky();
}  catch(AlphaEx a) {
     // recovery from AlphaEx
}  catch(BetaEx b) {
   // recovery from BetaEx
}  catch(GammaEx c) {
     // recovery from GammaEx
}  catch(DeltaEx d) {
     // recovery from DeltaEx
}
```

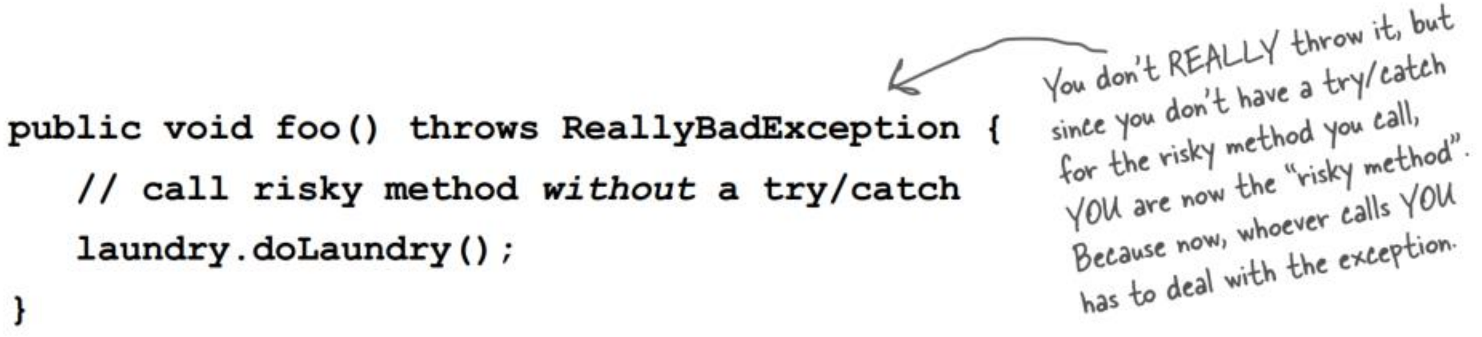
- Exercise 2: create two different legal try / catch structures to accurately represent the class diagram



- After Java 7, it supports multi-catch exceptions. That means you can catch multiple exceptions with a single catch block. This can prevent duplicate codes.

```
try{
    ...
} catch (parseException | IOException e){
    ...
}
```

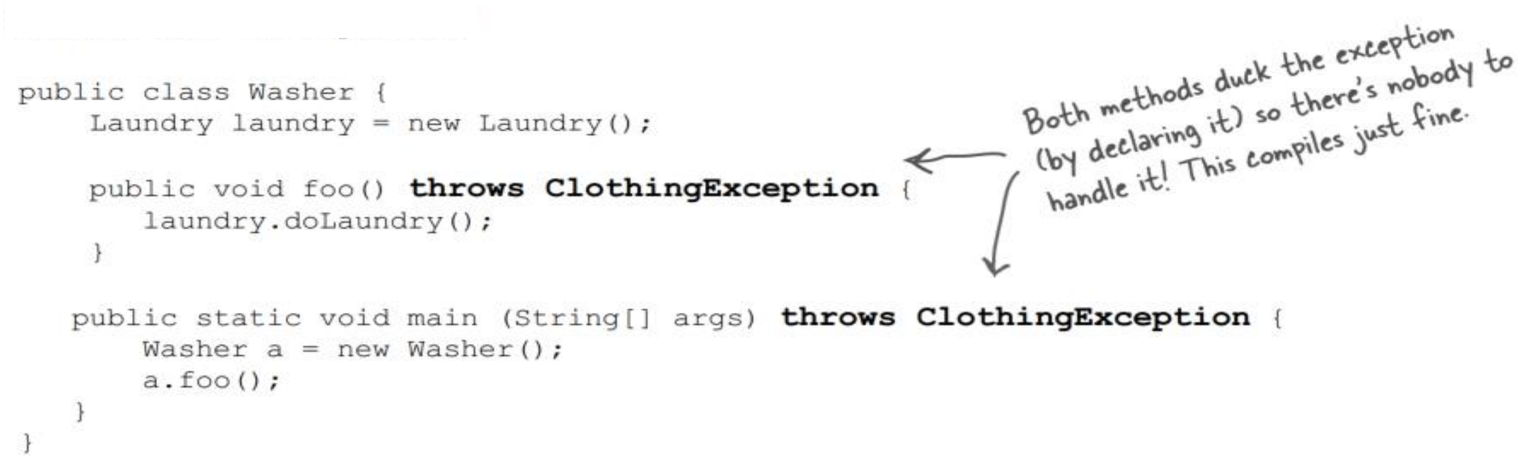
- If you don’t want to handle an exception, you can duck it by declaring it.
- Let the method that called you catch the exception.
- In other words, if you don’t handle the risky call, you become a risky method.



```
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.

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



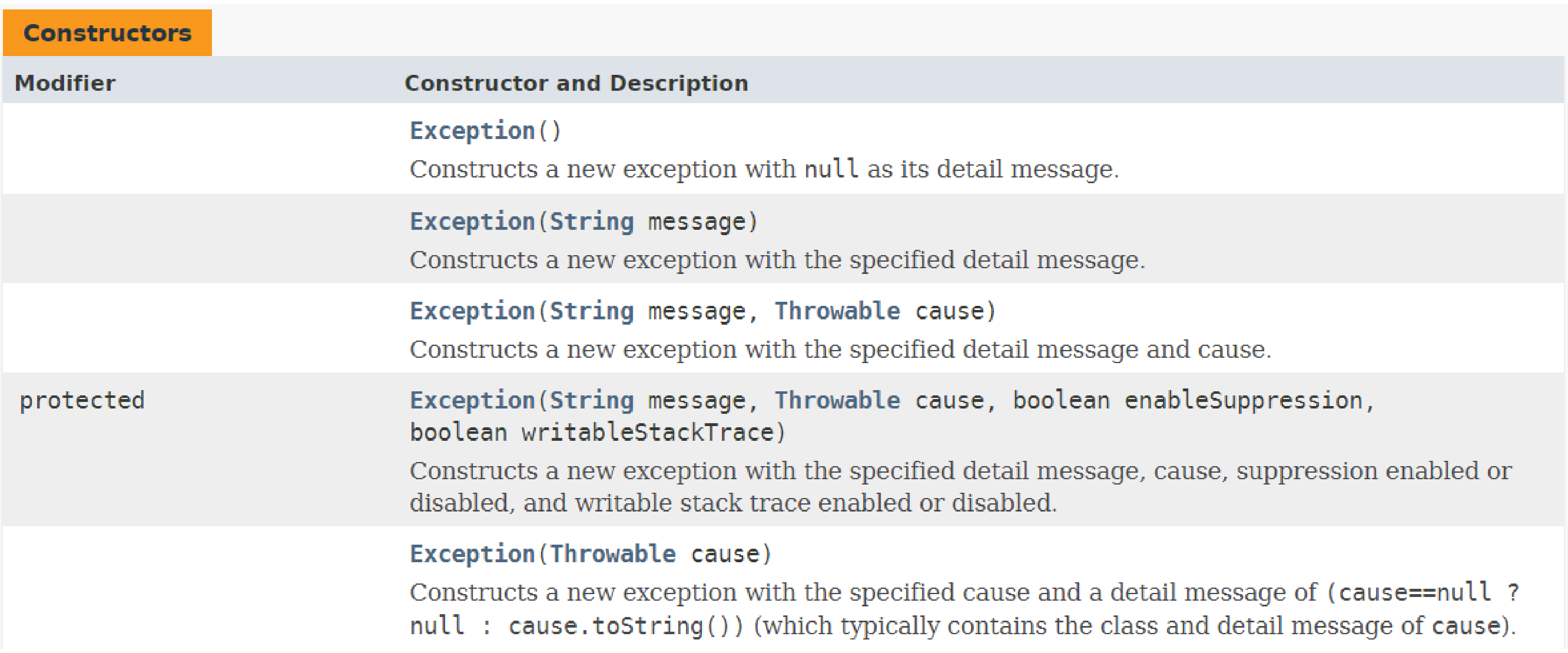
```
public class Washer {
    Laundry laundry = new Laundry();

    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.

- The JVM explodes!

- Java exceptions cover almost all general exceptions that are bound to happen in programming.
- However, sometimes we still have to build some exceptions with our own.
- Business logic exceptions – Exceptions that are specific to the business logic and workflow. These help the application users or the developers understand what the exact problem is
- For example, you want to merge two files into a single file, but the merge may fail. In that case you may want to throw an exception that is telling others why it fails.
- To catch and provide specific treatment to a subset of existing Java exceptions

- To create a custom exception, we have to extend the java.lang.Exception class.



Constructors

Modifier	Constructor and Description
	`Exception()` Constructs a new exception with `null` as its detail message.
	`Exception(String message)` Constructs a new exception with the specified detail message.
	`Exception(String message, Throwable cause)` Constructs a new exception with the specified detail message and cause.
`protected`	`Exception(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace)` Constructs a new exception with the specified detail message, cause, suppression enabled or disabled, and writable stack trace enabled or disabled.
	`Exception(Throwable cause)` Constructs a new exception with the specified cause and a detail message of `(cause==null ? null : cause.toString())` (which typically contains the class and detail message of `cause`).

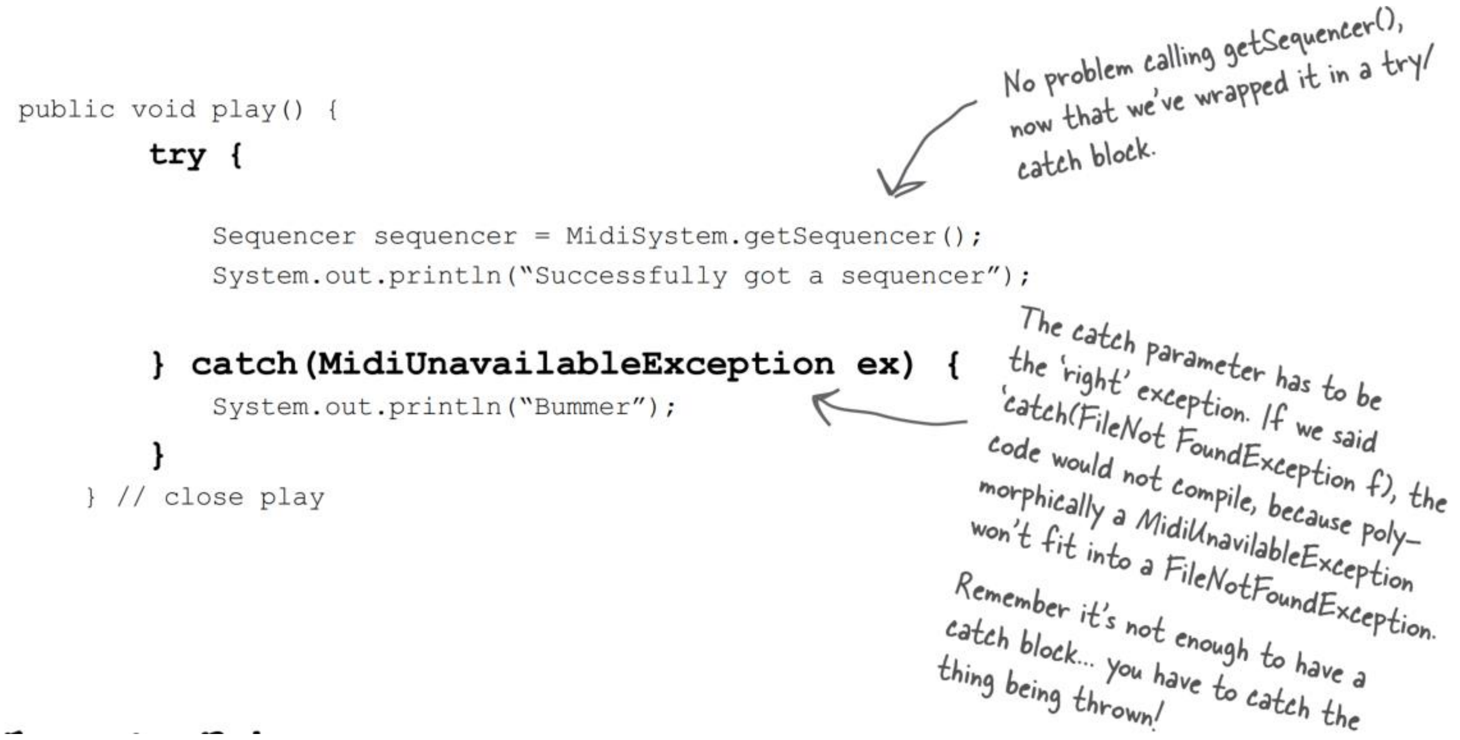
```
public class MyCustomException extends Exception
{
    public MyCustomException(String errorMessage)
    {
        super(errorMessage);
    }
}
```

Create a custom exception.

```
public class testException
{

    public void throwIt() throws MyCustomException
    {
        throw new MyCustomException("custom error!");
    }

    public static void main(String[] args)
    {
        testException t = new testException();

        try{
            t.throwIt();
        } catch(MyCustomException e){
            e.printStackTrace();
        }

    }
}
```

Use try/catch to handle the custom exception.

- Getting back to our music code
- We created a Sequencer object but it wouldn’t compile because the method Midi.getSequencer() declares a checked exception (MidiUnavailableException).
- Now we can fix that now by wrapping the call in a try/catch.

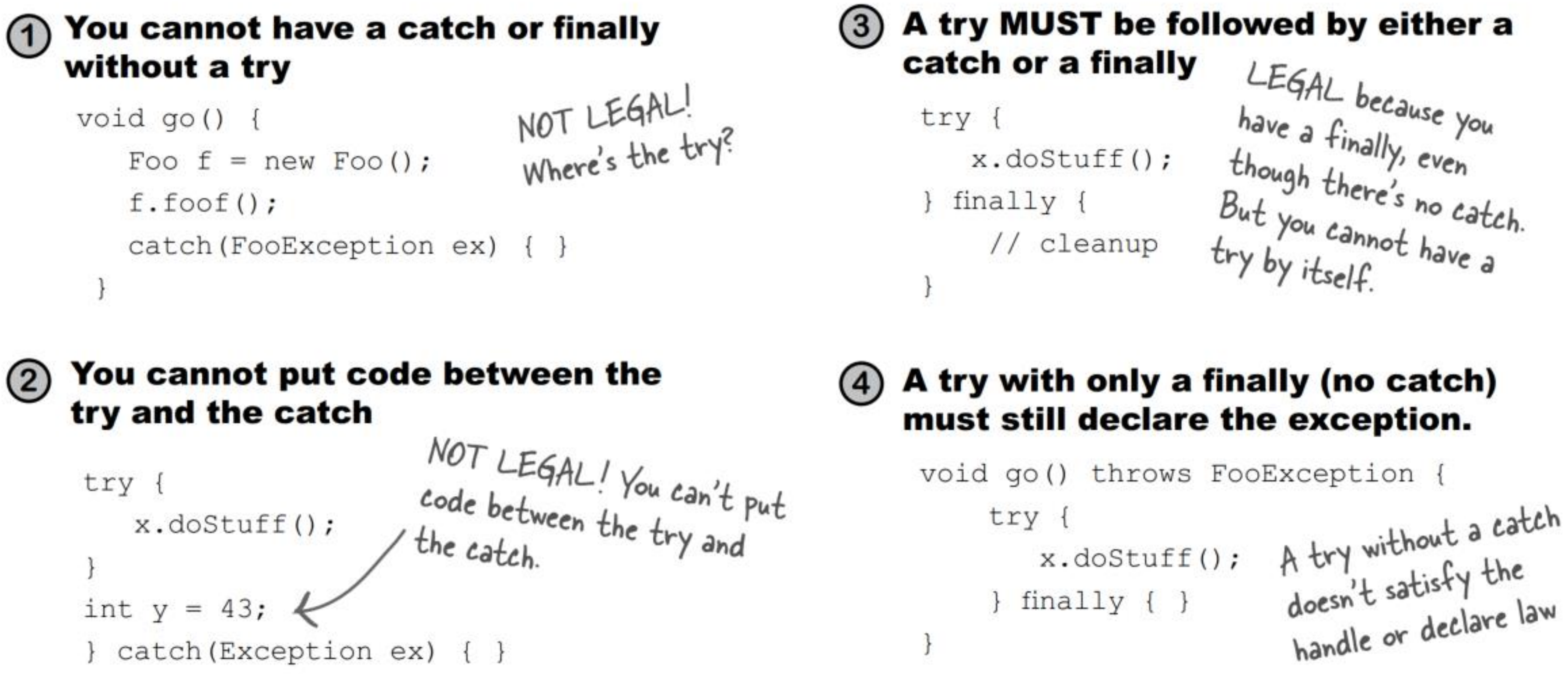


```
public void play() {
        try {

            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(FileNot FoundException f), the code would not compile, because polymorphically a MidiUnavilableException 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



1 You cannot have a catch or finally without a try

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

NOT LEGAL! Where's the try?

2 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.

3 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.

4 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

- Extension
- Exception handling with method overriding
- If superclass does not declare an exception, then the subclass can only declare unchecked exceptions, but not the checked exceptions.
- If superclass declares an exception, then the subclass can only declare the child exceptions of the exception declared by the superclass
- You can throw exceptions with abstract methods (that mean this behavior MAY be dangerous rather than WILL be dangerous)
- What if you tried to throw exceptions not declared by the interface? That would be "more restrictive" and clients who coded to the promises of the interface would be surprised in a bad way when you threw the (checked) exception.

- (Self-study)
- Essential components of the MidiSystem
- Synthesizer: This is the device that plays the midi soundtrack. It can either be a software synthesizer or a real midi compatible instrument.
- Sequencer: A sequencer takes in Midi data(via a sequence) and commands different instruments to play the notes. It arranges events according to start time, duration and channel to be played on.
- Channel: Midi supports up to 16 different channels. We can send off a midi event to any of those channels which are later synchronized by the sequencer.
- Track: It is a sequence of Midi events.
- Sequence: It is a data structure containing multiple tracks and timing information. The sequencer takes in a sequence and plays it.

- Get a Sequencer and open it
- Make a new Sequence
- Get a new Track from the Sequence
- Fill the track with MidiEvents and give the Sequence to the Sequencer
- Start playing

```
public void play() {

  try {

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

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

③     Track track = seq.createTrack();

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);

       player.setSequence(seq);

       player.start();

  } catch (Exception ex) {
        ex.printStackTrace();
  }
 } // close play
} // close class
```

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).

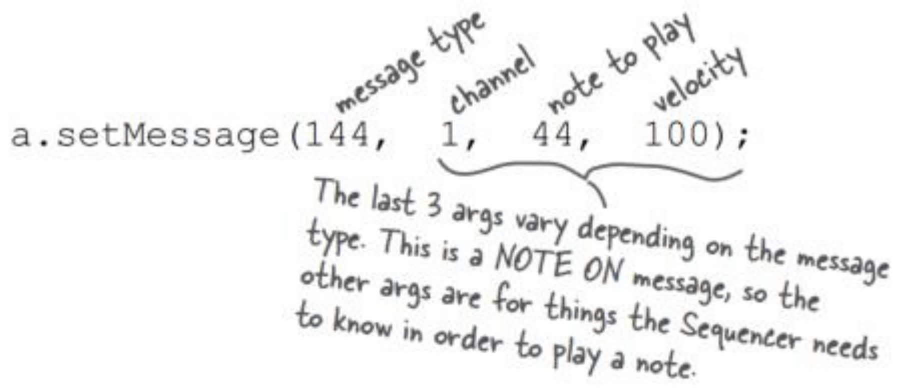
Ask the Sequence for a Track. Remember, the Track lives in the Sequence, and the MIDI data lives in the Track.

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.

Give the Sequence to the Sequencer (like putting the CD in the CD player)

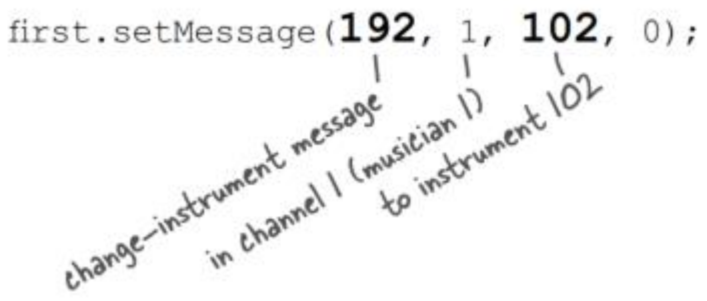
Start() the Sequencer (like pushing PLAY)

- A MIDI message holds the part of the event that says what to do.
- To make a MIDI message, make a ShortMessage instance and invoke setMessage(), passing in the four arguments for the message.
- The Message says what to do, the MidiEvent says when to do it.


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

- Type: 144 is start, 128 means stop
- Channel: 1 is the keyboard, channel 9 is the drummer.
- Note to play: 44 is a low C (Do~)
- Velocity: How hard and fast you press

- Exercise: try to make different sound by changing the parameters.
- Change the note: Try a number between 0 and 127 in the note on and note off messages.
- Change the duration: change the note off event (not the message) so that it happens at an earlier or later beat
- Change the instrument: add a new message, BEFORE the note-playing message, that sets the instrument in channel 1 to something other than the default piano. The change-instrument message is '192', and the third argument represents the actual instrument (try a number between 0 and 127)

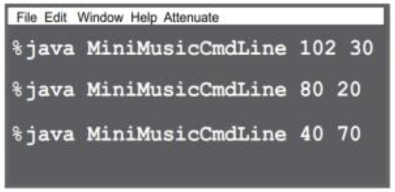


```
first.setMessage(192, 1, 102, 0);
```

- Using command-line args to experiment with sounds

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

public class MiniMusicCmdLine {    // this is the first one

   public static void main(String[] args) {
        MiniMusicCmdLine mini = new MiniMusicCmdLine();
        if (args.length < 2) {
            System.out.println("Don't forget the instrument and note args");
        } else {
            int instrument = Integer.parseInt(args[0]);
            int note = Integer.parseInt(args[1]);
            mini.play(instrument, note);
        }
   } // close main

   public void play(int instrument, int note) {

      try {

         Sequencer player = MidiSystem.getSequencer();
         player.open();
         Sequence seq = new Sequence(Sequence.PPQ, 4);
         Track track = seq.createTrack();

         MidiEvent event = null;

         ShortMessage first = new ShortMessage();
         first.setMessage(192, 1, instrument, 0);
         MidiEvent changeInstrument = new MidiEvent(first, 1);
         track.add(changeInstrument);

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

         ShortMessage b = new ShortMessage();
         b.setMessage(128, 1, note, 100);
         MidiEvent noteOff = new MidiEvent(b, 16);
         track.add(noteOff);
         player.setSequence(seq);
         player.start();

      } catch (Exception ex) {ex.printStackTrace();}
   } // close play
} // close class
```

Run it with two int args from 0 to 127. Try these for starters:



```
%java MiniMusicCmdLine 102 30
%java MiniMusicCmdLine 80 20
%java MiniMusicCmdLine 40 70
```

TRUE OR FALSE

1. A try block must be followed by a catch *and* a finally block.

2. If you write a method that might cause a compiler-checked exception, you *must* wrap that risky code in a try / catch block.

3. Catch blocks can be polymorphic.

4. Only 'compiler checked' exceptions can be caught.

5. If you define a try / catch block, a matching finally block is optional.

6. If you define a try block, you can pair it with a matching catch or finally block, or both.

7. If you write a method that declares that it can throw a compiler-checked exception, you must also wrap the exception throwing code in a try / catch block.

8. The main( ) method in your program must handle all unhandled exceptions thrown to it.

9. A single try block can have many different catch blocks.

10. A method can only throw one kind of exception.

11. A finally block will run regardless of whether an exception is thrown.

12. A finally block can exist without a try block.

13. A try block can exist by itself, without a catch block or a finally block.

14. Handling an exception is sometimes referred to as 'ducking'.

15. The order of catch blocks never matters.

16. A method with a try block and a finally block, can optionally declare the exception.

17. Runtime exceptions must be *handled* or *declared*.