Risky Behavior

Sure it's risky, but I can **handle** it if something goes wrong.

Stuff happens. The file isn't there. The server is down. No matter how good a programmer you are, you can't control everything. Things can go wrong. *Very* wrong. When you write a risky method, you need code to handle the bad things that might happen. But how do you *know* when a method is risky? And where do you put the code to *handle* the *exceptional* situation? So far in this book, we haven't *really* taken any risks. We've certainly had things go wrong at runtime, but the problems were mostly flaws in our own code. Bugs. And those we should fix at development time. No, the problem-handling code we're talking about here is for code that you *can't* guaranatee will work at runtime. Code that expects the file to be in the right directory, the server to be running, or the Thread to stay asleep. And we have to do this *now*. Because in *this* chapter, we're going to build something that uses the risky JavaSound API. We're going to build a MIDI Music Player.

Let's make a Music Machine

Over the next three chapters, we'll build a few different sound applications, including a BeatBox Drum Machine. In fact, before the book is done, we'll have a multi-player version so you can send your drum loops to another player, kind of like a chat room. You're going to write the whole thing, although you can choose to use Ready-bake code for the GUI parts. OK, so not every IT department is looking for a new BeatBox server, but we're doing this to *learn* more about *Java*. Building a BeatBox is just a way to have fun *while* we're learning Java.

The finished BeatBox looks something like this:

You make a beatbox loop (a 16-beat drum pattern) by putting checkmarks in the boxes.
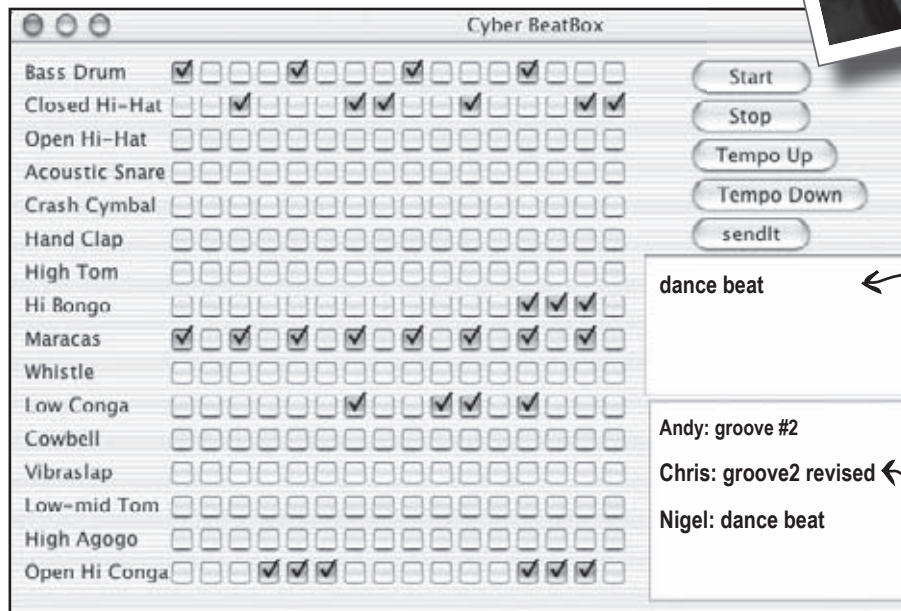
your message, that gets sent to the other players, along with your current beat pattern, when you hit "SendIt"

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

Put checkmarks in the boxes for each of the 16 'beats'. For example, on beat 1 (of 16) the Bass drum and the Maracas will play, on beat 2 nothing, and on beat 3 the Maracas and Closed Hi-Hat... you get the idea. When you hit 'Start', it plays your pattern in a loop until you hit 'Stop'. At any time, you can "capture" one of your own patterns by sending it to the BeatBox server (which means any other players can listen to it). You can also load any of the incoming patterns by clicking on the message that goes with it.

We'll start with the basics

Obviously we've got a few things to learn before the whole program is finished, including how to build a Swing GUI, how to *connect* to another machine via networking, and a little I/O so we can *send* something to the other machine.
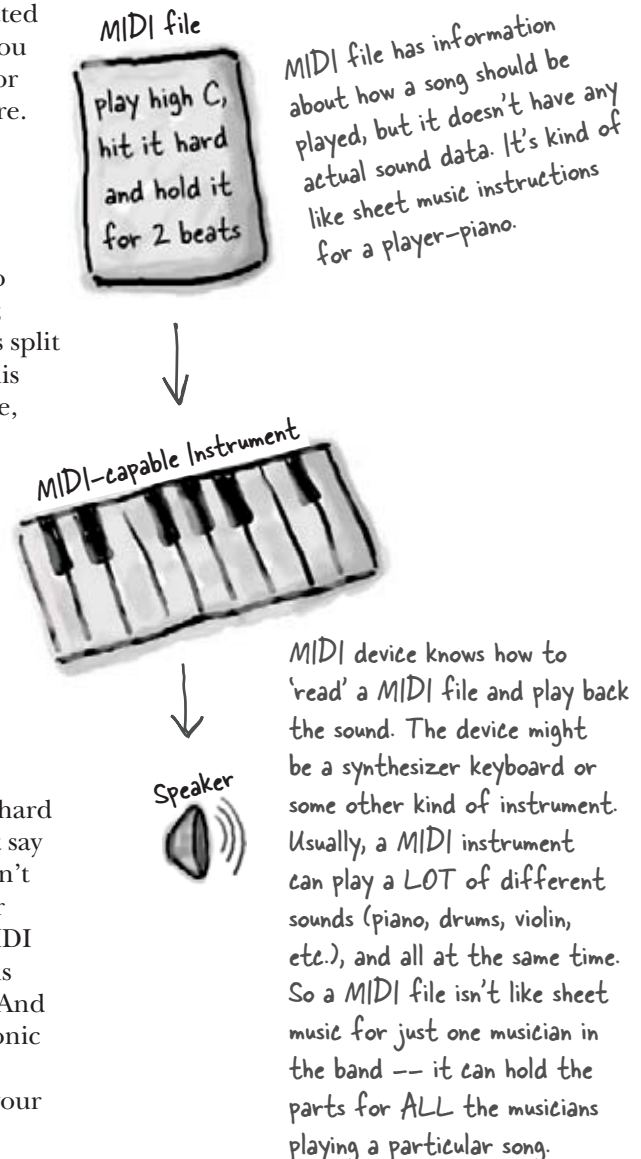
Oh yeah, and the JavaSound API. *That's* where we'll start in this chapter. For now, you can forget the GUI, forget the networking and the I/O, and focus only on getting some MIDI-generated sound to come out of your computer. And don't worry if you don't know a thing about MIDI, or a thing about reading or making music. Everything you need to learn is covered here. You can almost smell the record deal.

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. JavaSound is split into two parts: MIDI and Sampled. We use only MIDI in this book. MIDI stands for Musical Instrument Digital Interface, and is a standard protocol for getting different kinds of electronic sound equipment to communicate. But for our BeatBox app, you can think of MIDI as *a kind of sheet music* that you feed into some device you can think of like a high-tech 'player piano'. In other words, MIDI data doesn't actually include any *sound*, but it does include the *instructions* that a MIDI-reading instrument can play back. Or for another analogy, you can think of a MIDI file like an HTML document, and the instrument that renders the MIDI file (i.e. *plays* it) is like the Web browser.

MIDI data says *what* to do (play middle C, and here's how hard to hit it, and here's how long to hold it, etc.) but it doesn't say anything at all about the actual *sound* you hear. MIDI doesn't know how to make a flute, piano, or Jimmy Hendrix guitar sound. For the actual sound, we need an instrument (a MIDI device) that can read and play a MIDI file. But the device is usually more like an *entire band or orchestra* of instruments. And that instrument might be a physical device, like the electronic keyboard synthesizers the rock musicians play, or it could even be an instrument built entirely in software, living in your computer.

For our BeatBox, we use only the built-in, software-only instrument that you get with Java. It's called a *synthesizer* (some folks refer to it as a *software synth*) because it *creates* sound. Sound that you *hear*.

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.

```java
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.

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?
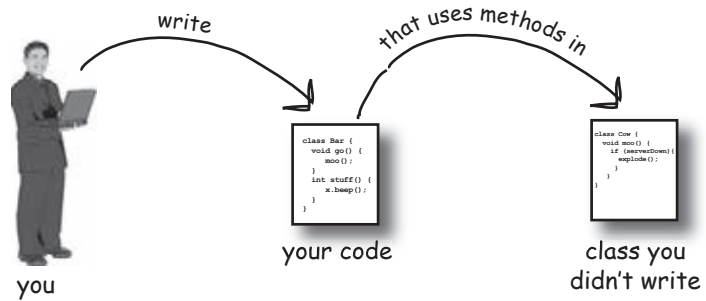
```
% javac MusicTest1.java

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

    Sequencer sequencer = MidiSystem.getSequencer();
                                    ^

1 errors
```
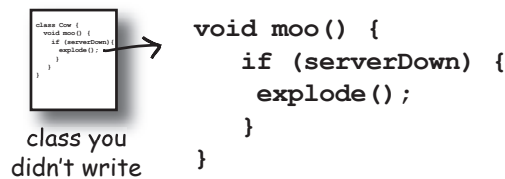
What happens when a method you want to call (probably in a class you didn't write) is risky?
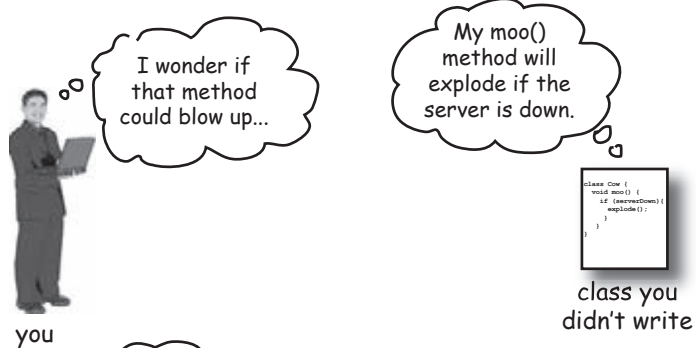
write • *that uses methods in*

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

```
class Bar {
    void go() {
        moo();
    }
    int stuff() {
        x.beep();
    }
}
```
your code

```
class Cow {
    void moo() {
        if (serverDown){
            explode();
        }
    }
}
```
class you didn't write

you

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

```
class Cow {
    void moo() {
        if (serverDown){
            explode();
        }
    }
}
```
class you didn't write

```
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.

```
class Cow {
    void moo() {
        if (serverDown){
            explode();
        }
    }
}
```
class you didn't write

you

Now that I know, I can take precautions.

write safely

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

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

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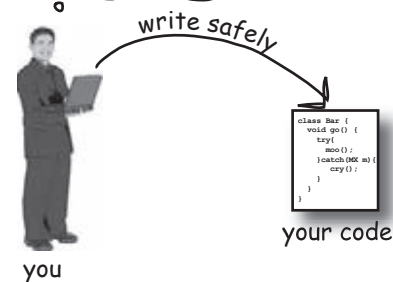
Methods in Java use *exceptions* to tell the calling code, "Something Bad Happened. I failed."

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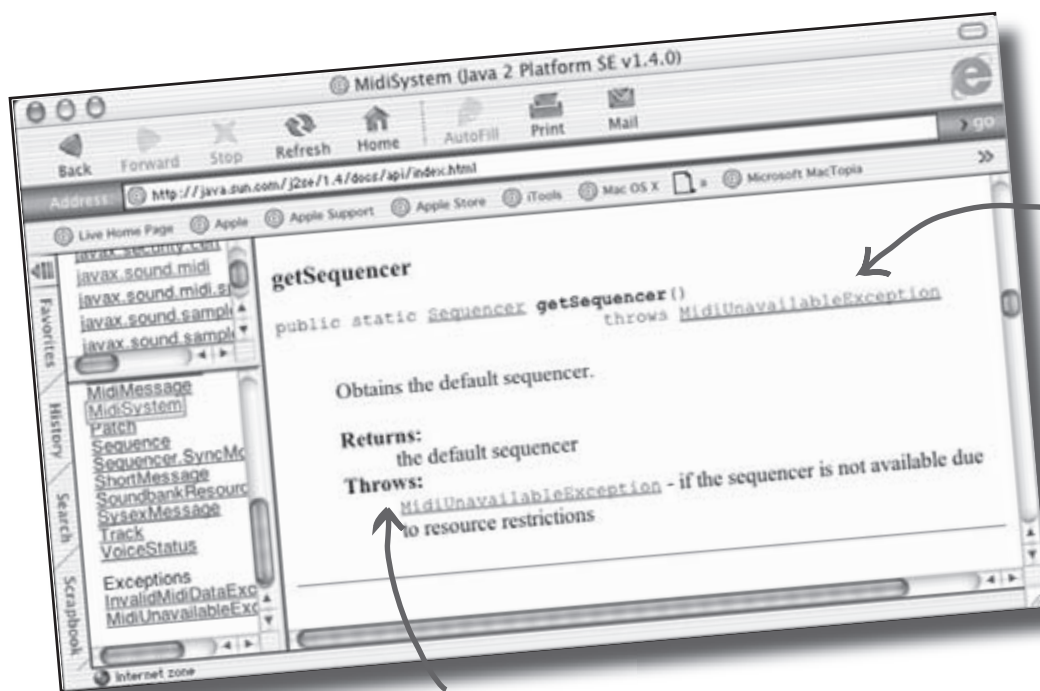
The API docs tell you that getSequencer() can throw an exception: MidiUnavailableException. A method has to declare the exceptions it might throw.

This part tells you WHEN you might get that exception -- in this case, because of resource restrictions (which could just means 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.

Dear Compiler,

I know I'm taking a risk here, but don't you think it's worth it? What should I do?

signed, geeky in Waikiki

Dear geeky,

Life is short (especially on the heap). Take the risk. try it. But just in case things don't work out, be sure to catch any problems before all hell breaks loose.

```java
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().
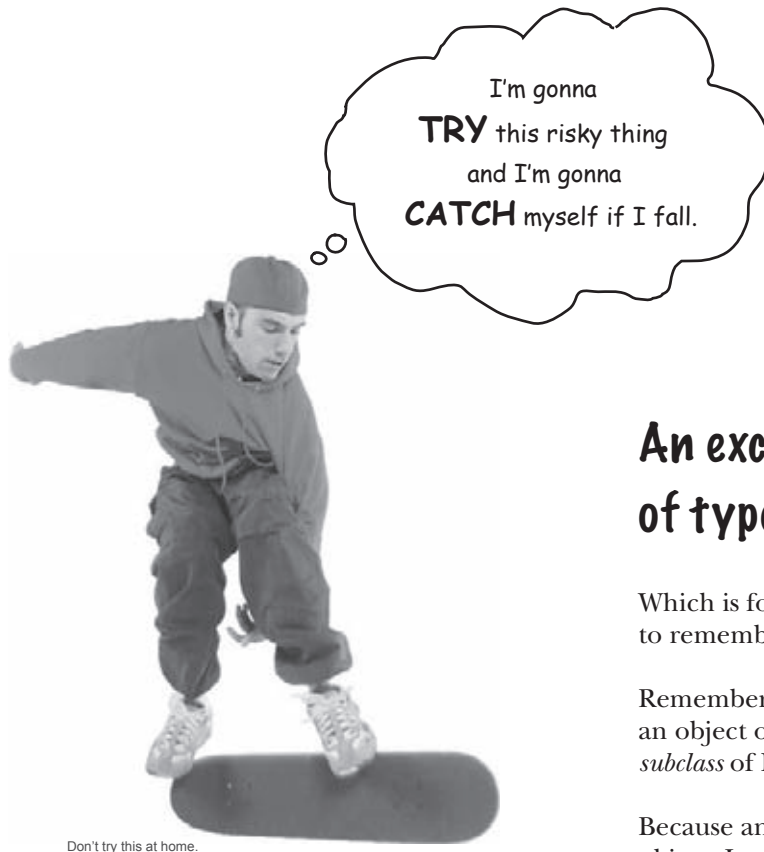
I'm gonna **TRY** this risky thing and I'm gonna **CATCH** myself if I fall.

Don't try this at home.

Throwable

getMessage()
printStackTrace()

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

Exception

IOException

InterruptedException

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

Which is fortunate, because it would be much harder to remember if exceptions were of type Broccoli.

Remember from your polymorphism chapters that an object of type Exception *can* be an instance of any *subclass* of Exception.
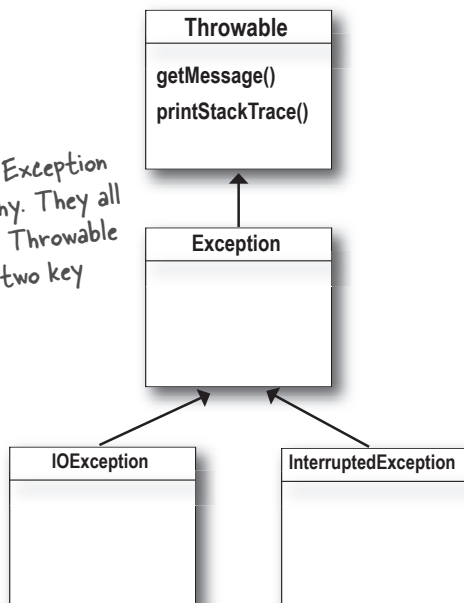
Because an *Exception* is an object, what you *catch* is an object. In the following code, the **catch** argument is declared as type Exception, and the parameter reference variable is *ex*.

```
try {

    // do risky thing

} catch(Exception ex) {

    // try to recover

}
```

it's just like declaring a method argument.

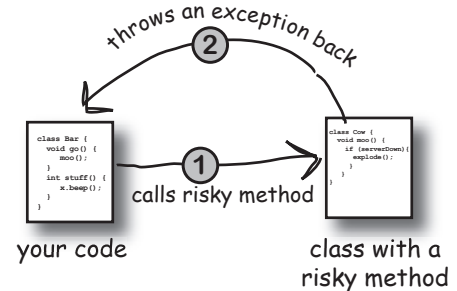*This code only runs if an Exception is thrown.*

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?

You'll spend much more of your Java coding time *handling* exceptions than you'll spend *creating* and *throwing* them yourself. For now, just know that when your code *calls* a risky method—a method that declares an exception—it's the risky method that *throws* the exception back to *you*, the caller.

In reality, it might be you who wrote both classes. It really doesn't matter who writes the code... what matters is knowing which method *throws* the exception and which method *catches* it.

When somebody writes code that could throw an exception, they must *declare* the exception.

throws an exception back

②

```
class Bar {
    void go() {
        moo();
    }
    int stuff() {
        x.beep();
    }
}
```

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

①

calls risky method

your code

class with a risky method

---

① **Risky, exception-throwing code:**

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

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

create a new Exception object and throw it.

> One method will catch what another method throws. An exception is always thrown back to the caller.
>
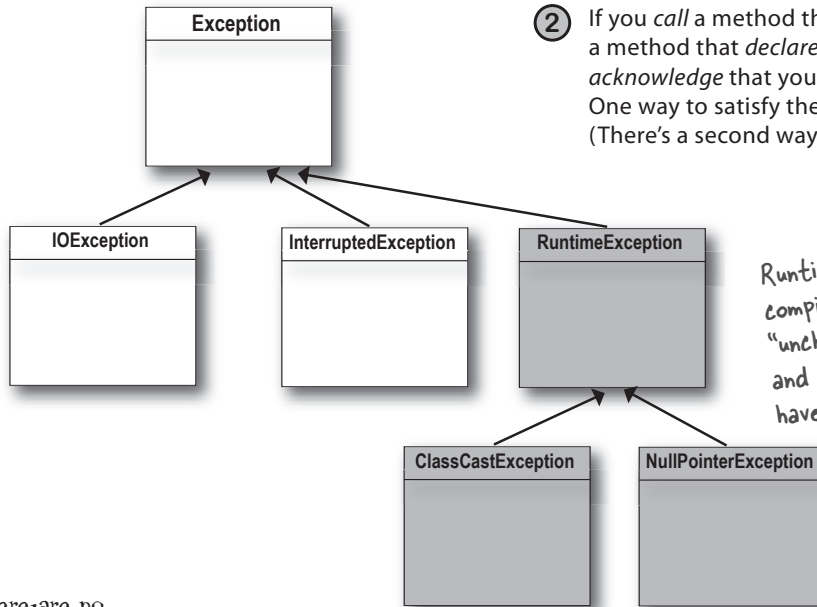> The method that throws has to declare that it might throw the exception.

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

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

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

The compiler checks for everything except RuntimeExceptions.

The compiler guarantees:

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

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.)

Exception

IOException

InterruptedException

RuntimeException

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.

ClassCastException

NullPointerException

there are no Dumb Questions

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.

BULLET POINTS

- A method can throw an exception when something fails at runtime.

- An exception is always an object of type Exception. (Which, as you remember from the polymorphism chapters means the object is from a class that has Exception somewhere up its inheritance tree.)

- The compiler does NOT pay attention to exceptions that are of type **RuntimeException**. A RuntimeException does not have to be declared or wrapped in a try/catch (although you're free to do either or both of those things)

- All Exceptions the compiler cares about are called 'checked exceptions' which really means *compiler*-checked exceptions. Only RuntimeExceptions are excluded from compiler checking. All other exceptions must be acknowledged in your code, according to the rules.

- A method throws an exception with the keyword **throw**, followed by a new exception object:

  ```
  throw new NoCaffeineException();
  ```

- Methods that *might* throw a checked exception ***must*** announce it with a **throws Exception** declaration.

- If your code calls a checked-exception-throwing method, it must reassure the compiler that precautions have been taken.

- If you're prepared to handle the exception, wrap the call in a try/catch, and put your exception handling/recovery code in the catch block.

- If you're not prepared to handle the exception, you can still make the compiler happy by officially 'ducking' the exception. We'll talk about ducking a little later in this chapter.

metacogNitive tiP

If you're trying to learn something new, make that the *last* thing you try to learn before going to sleep. So, once you put this book down (assuming you can tear yourself away from it) don't read anything else more challenging than the back of a Cheerios™ box. Your brain needs time to process what you've read and learned. That could take a few *hours*. If you try to shove something new in right on top of your Java, some of the Java might not 'stick.'

Of course, this doesn't rule out learning a physical skill. Working on your latest Ballroom KickBoxing routine probably won't affect your Java learning.

For the best results, read this book (or at least look at the pictures) right before going to sleep.

Sharpen your pencil

Things you want to do

What might go wrong

Which of these do you think might throw an exception that the compiler would care about? We're only looking for the things that you can't control in your code. We did the first one.

(Because it was the easiest.)

✔ connect to a remote server   <u>the server is down</u>

___ access an array beyond its length   _____

___ display a window on the screen   _____

___ retrieve data from a database   _____

___ see if a text file is where you *think* it is   _____

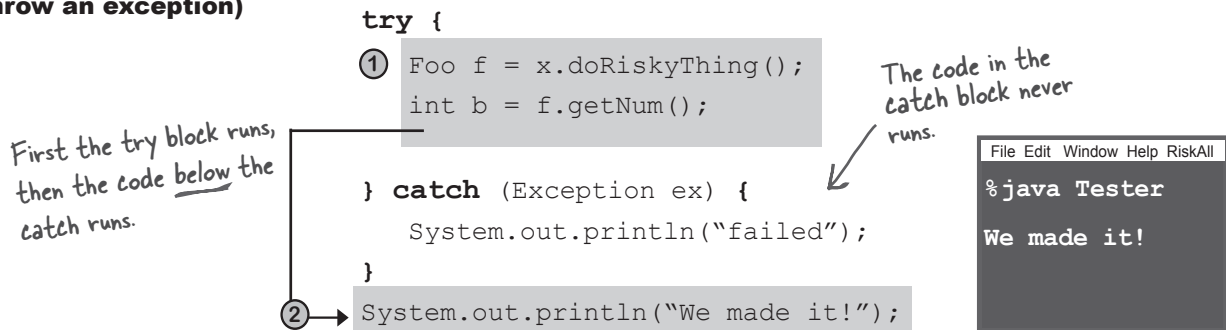___ create a new file   _____

___ read a character from the command-line   _____

Flow control in try/catch blocks

When you call a risky method, one of two things can happen. The risky method either succeeds, and the try block completes, or the risky method throws an exception back to your calling method.
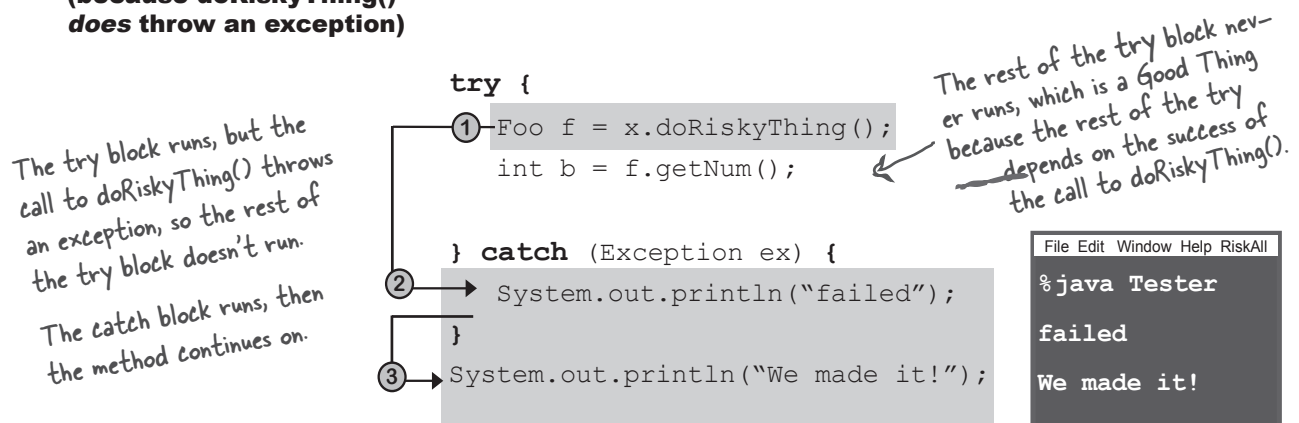
If the try **succeeds**

(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.

File Edit Window Help RiskAll

%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 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().

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.

File Edit Window Help RiskAll

%java Tester

failed

We made it!

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

If you try to cook something, you start by turning on the oven.

If the thing you try is a complete **failure**, *you have to turn off the oven.*

If the thing you try **succeeds**, *you have to turn off the oven.*

You have to turn off the oven no matter what!

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();
}
```

Are you sure you want to try this?

No matter what, do NOT let me forget to turn off the oven! Last time I torched half the neighborhood.

If the try block fails (an exception), flow control immediately moves to the catch block. When the catch block completes, the finally block runs. When the finally block completes, the rest of the method continues on.

If the try block succeeds (*no* exception), flow control skips over the catch block and moves to the finally block. When the finally block completes, the rest of the method continues on.

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

Sharpen your pencil

Flow Control

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.

```java
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"`

When test = "yes": start try - start risky - scary exception - finally - end of main
When test = "no": start try - start risky - end risky - end try - finally - end of main

Did we mention that a method can throw more than one exception?

A method can throw multiple exceptions if it darn well needs to. But a method's declaration must declare *all* the checked exceptions it can throw (although if two or more exceptions have a common superclass, the method can declare just the superclass.)

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

Exceptions are objects, remember. There's nothing all that special about one, except that it is *a thing that can be **thrown***. So like all good objects, Exceptions can be referred to polymorphically. A LingerieException *object*, for example, could be assigned to a ClothingException *reference*. A PantsException could be assigned to an Exception reference. You get the idea. The benefit for exceptions is that a method doesn't have to explicitly declare every possible exception it might throw; it can declare a superclass of the exceptions. Same thing with catch blocks—you don't have to write a catch for each possible exception as long as the catch (or catches) you have can handle any exception thrown.

All exceptions have Exception as a superclass.

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

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

Declaring a ClothingException lets you throw any subclass of ClothingException . That means doLaundry() can throw a PantsException, LingerieException, TeeShirtException, and DressShirtException without explicitly declaring them individually.
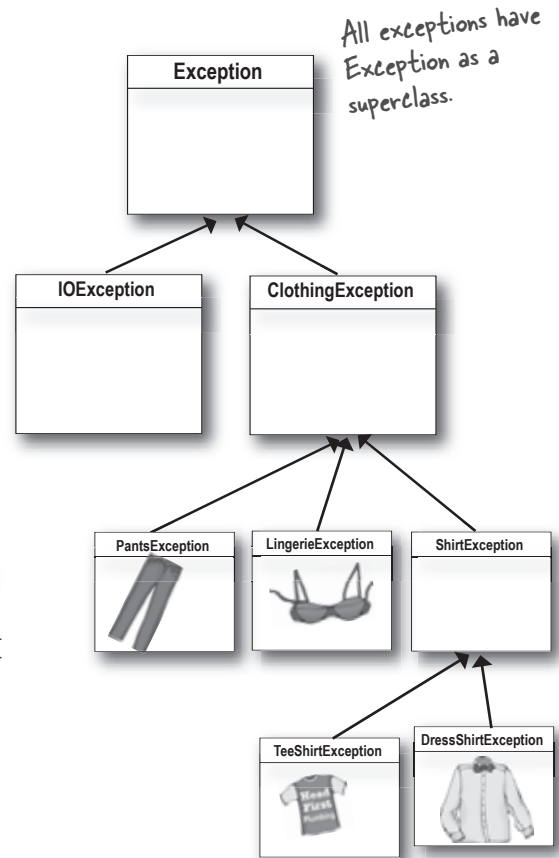
(2) **You can CATCH exceptions using a supertype of the exception thrown.**

can catch any ClothingException subclass

```
try {

    laundry.doLaundry();

} catch(ClothingException cex) {

    // recovery code

}
```

can catch only TeeShirtException and DressShirtException

```
try {

    laundry.doLaundry();

} catch(ShirtException sex) {

    // recovery code

}
```

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

ClothingException

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

`catch(TeeShirtException tex)`

PantsException

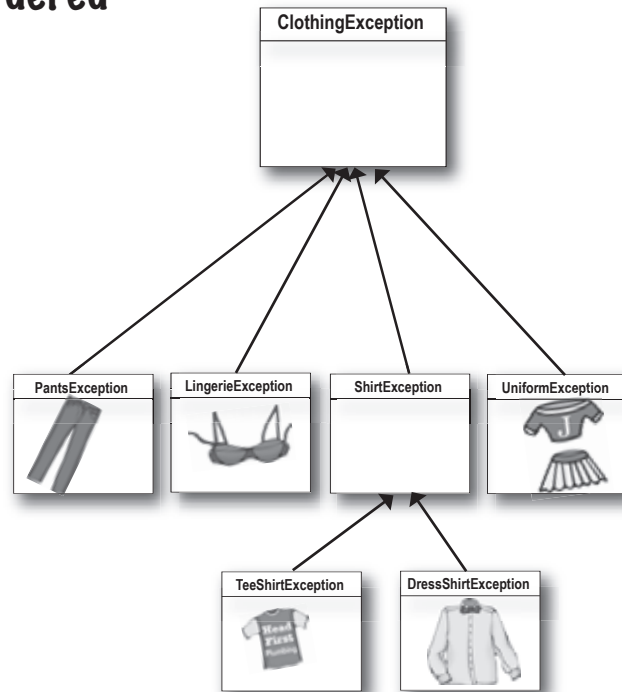**LingerieException**

ShirtException

UniformException

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

`catch(ShirtException sex)`

TeeShirtException

DressShirtException

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

`catch(ClothingException cex)`

The higher up the inheritance tree, the bigger the catch 'basket'. As you move down the inheritance tree, toward more and more specialized Exception classes, the catch 'basket' is smaller. It's just plain old polymorphism.

A ShirtException catch is big enough to take a TeeShirtException or a DressShirtException (and any future subclass of anything that extends ShirtException). A ClothingException is even bigger (i.e. there are more things that can be referenced using a ClothingException type). It can take an exception of type ClothingException(duh), and any ClothingException subclasses: PantsException, UniformException, LingerieException, and ShirtException. The mother of all catch arguments is type **Exception**; it will catch *any* exception, including runtime (unchecked) exceptions, so you probably won't use it outside of testing.

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.

> 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.

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

}
```

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.

polymorphic puzzle

Sharpen your pencil

Assume the try/catch block here is legally coded. Your task is to draw two different class diagrams that can accurately reflect the Exception classes. In other words, what class inheritance structures would make the try/catch blocks in the sample code legal?

```
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
}
```

Your task is to create two different *legal* try / catch structures (similar to the one above left), to accurately represent the class diagram shown on the left. Assume ALL of these exceptions might be thrown by the method with the try block.

BazEx

FooEx

BarEx

BiffEx

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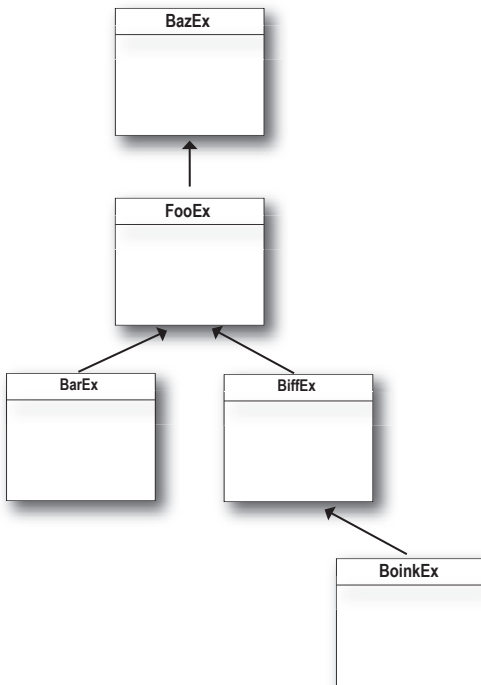
When you don't want to handle an exception...

just duck it

If you don't want to handle an exception, you can **duck** it by **declaring** it.
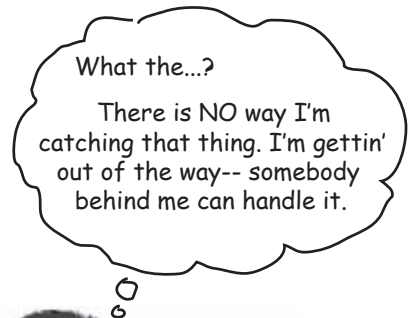
When you call a risky method, the compiler needs you to acknowledge it. Most of the time, that means wrapping the risky call in a try/catch. But you have another alternative, simply *duck* it and let the method that called *you* catch the exception.

It's easy—all you have to do is *declare* that *you* throw the exceptions. Even though, technically, *you* aren't the one doing the throwing, it doesn't matter. You're still the one letting the exception whiz right on by.

But if you duck an exception, then you don't have a try/catch, so what happens when the risky method (doLaundry()) *does* throw the exception?

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.

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

```
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?

```java
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

2 foo() ducks the exception

3 main() ducks the exception

4 The JVM shuts down

doLaundry
foo
main

foo
main

main

main() calls foo()

foo() calls doLaundry()

doLaundry() is running and throws a ClothingException

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

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

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."

We're using the tee-shirt to represent a Clothing Exception. We know, we know... you would have preferred the blue jeans.

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.**

① HANDLE

Wrap the risky call in a try/catch

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

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

② DECLARE (duck it)

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

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

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

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.

```java
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 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

```java
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

```java
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

```java
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.

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

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

Code Kitchen

You don't have to do it yourself, but it's a lot more fun if you do.
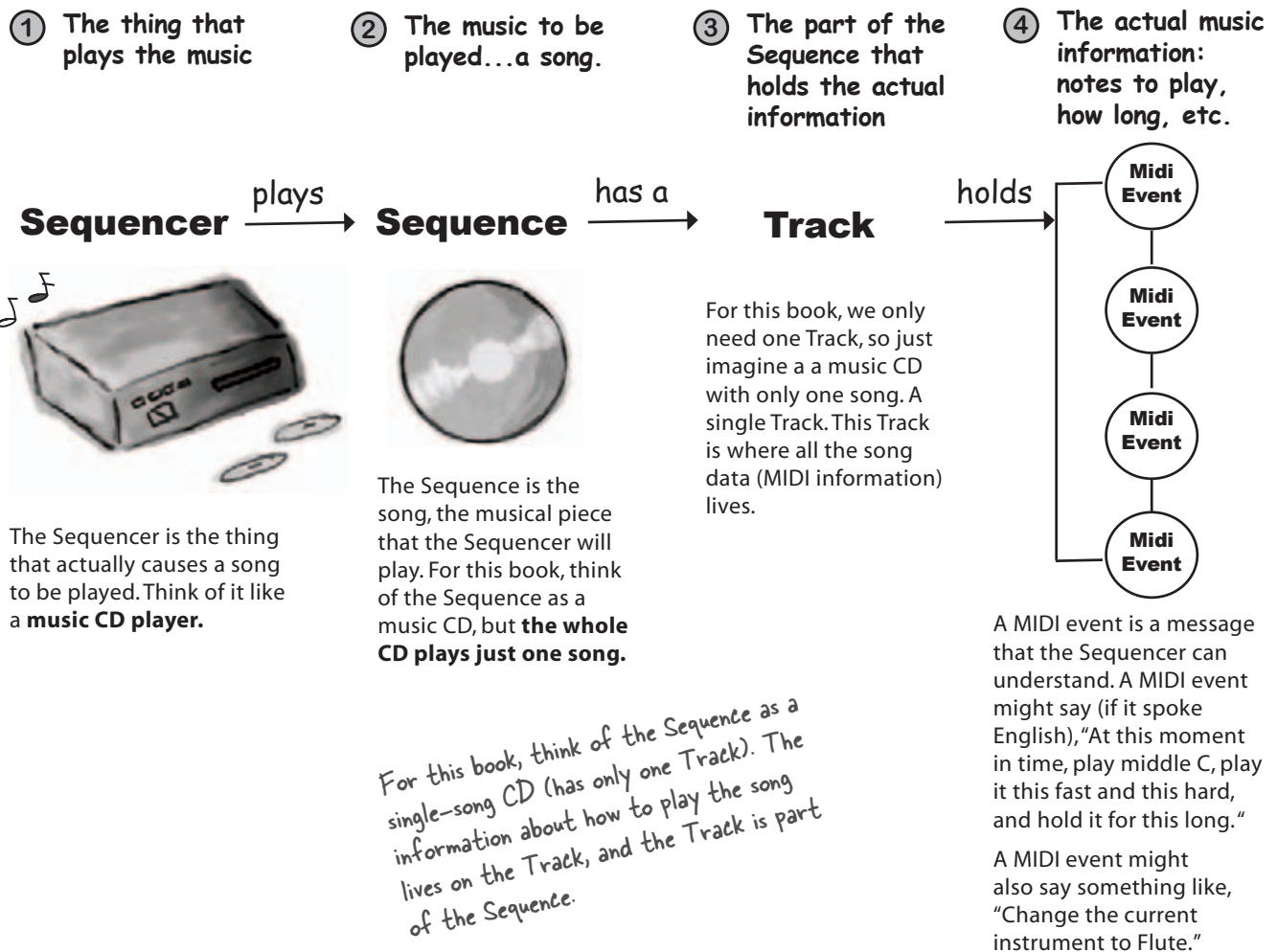
The rest of this chapter is optional; you can use Ready-bake code for all the music apps.

But if you want to learn more about JavaSound, turn the page.

Making actual sound

Remember near the beginning of the chapter, we looked at how MIDI data holds the instructions for *what* should be played (and *how* it should be played) and we also said that MIDI data doesn't actually *create any sound that you hear*. For sound to come out of the speakers, the MIDI data has to be sent through some kind of MIDI device that takes the MIDI instructions and renders them in sound, either by triggering a hardware instrument or a 'virtual' instrument (software synthesizer). In this book, we're using only software devices, so here's how it works in JavaSound:

You need FOUR things:

① **The thing that plays the music**

② **The music to be played...a song.**

③ **The part of the Sequence that holds the actual information**

④ **The actual music information: notes to play, how long, etc.**

Sequencer —plays→ **Sequence** —has a→ **Track** —holds→

The Sequencer is the thing that actually causes a song to be played. Think of it like a **music CD player.**

The Sequence is the song, the musical piece that the Sequencer will play. For this book, think of the Sequence as a music CD, but **the whole CD plays just one song.**

For this book, we only need one Track, so just imagine a a music CD with only one song. A single Track. This Track is where all the song data (MIDI information) lives.

Midi Event
Midi Event
Midi Event
Midi Event

A MIDI event is a message that the Sequencer can understand. A MIDI event might say (if it spoke English), "At this moment in time, play middle C, play it this fast and this hard, and hold it for this long."

A MIDI event might also say something like, "Change the current instrument to Flute."

For this book, think of the Sequence as a single-song CD (has only one Track). The information about how to play the song lives on the Track, and the Track is part of the Sequence.

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);
```

Uh, hate to break it to you, but that's only FOUR steps.

Ahhhh. We forgot to push the PLAY button. You have to start() the Sequencer!

```
player.start();
```

Your very first sound player app

Type it in and run it. You'll hear the sound of someone playing a single note on a piano! (OK, maybe not some*one*, but some*thing*.)

```java
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 {
```
get a Sequencer and open it (so we can use it... a Sequencer doesn't come already open)
```java
①    Sequencer player = MidiSystem.getSequencer();
     player.open();


②    Sequence seq = new Sequence(Sequence.PPQ, 4);
```
Don't worry about the arguments to the Sequence constructor. Just copy these (think of 'em as Ready-bake arguments).
```java
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.
```java
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.
```java
     player.setSequence(seq);
```
Give the Sequence to the Sequencer (like putting the CD in the CD player)
```java
     player.start();
```
Start() the Sequencer (like pushing PLAY)
```java
     } catch (Exception ex) {
         ex.printStackTrace();
     }
  } // close play
} // close class
```

Making a MidiEvent (song data)

A MidiEvent is an instruction for part of a song. A series of MidiEvents is kind of like sheet music, or a player piano roll. Most of the MidiEvents we care about describe *a thing to do* and the *moment in time to do it*. The moment in time part matters, since timing is everything in music. This note follows this note and so on. And because MidiEvents are so detailed, you have to say at what moment to *start* playing the note (a NOTE ON event) and at what moment to *stop* playing the notes (NOTE OFF event). So you can imagine that firing the "stop playing note G" (NOTE OFF message) *before* the "start playing Note G" (NOTE ON) message wouldn't work.

The MIDI instruction actually goes into a Message object; the MidiEvent is a combination of the Message plus the moment in time when that message should 'fire'. In other words, the Message might say, "Start playing Middle C" while the MidiEvent would say, "Trigger this message at beat 4".

So we always need a Message and a MidiEvent.

The Message says *what* to do, and the MidiEvent says *when* to do it.

> A MidiEvent says <u>what</u> to do and <u>when</u> to do it.
>
> Every instruction must include the <u>timing</u> for that instruction.
>
> In other words, at which <u>beat</u> that thing should happen.

(1) Make a **Message**

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

(2) Put the **Instruction** in the Message

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

⟵ This message says, "start playing note 44" (we'll look at the other numbers on the next page)

(3) Make a new **MidiEvent** using the Message

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

⟵ The instructions are in the message, but the Midi-Event adds the moment in time when the instruction should be triggered. This MidiEvent says to trigger message 'a' at the first beat (beat 1).

(4) Add the MidiEvent to the **Track**

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

A Track holds all the MidiEvent objects. The Sequence organizes them according to when each event is supposed to happen, and then the Sequencer plays them back in that order. You can have lots of events happening at the exact same moment in time. For example, you might want two notes played simultaneously, or even different instruments playing different sounds at the same time.

MIDI message: the heart of a MidiEvent

A MIDI message holds the part of the event that says *what* to do. The actual instruction you want the sequencer to execute. The first argument of an instruction is always the type of the message. The values you pass to the other three arguments depend on the type of message. For example, a message of type 144 means "NOTE ON". But in order to carry out a NOTE ON, the sequencer needs to know a few things. Imagine the sequencer saying, "OK, I'll play a note, but *which channel?* In other words, do you want me to play a Drum note or a Piano note? And *which note?* Middle-C? D Sharp? And while we're at it, at *which velocity* should I play the note?

To make a MIDI message, make a ShortMessage instance and invoke setMessage(), passing in the four arguments for the message. But remember, the message says only *what* to do, so you still need to stuff the message into an event that adds *when* that message should 'fire'.

Anatomy of a message

The *first* argument to setMessage() always represents the message 'type', while the *other three* arguments represent different things depending on the message type.

> 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.
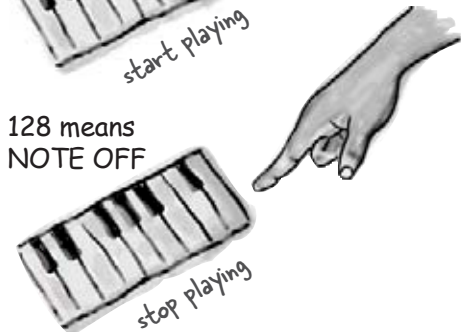
> The *Message* says what to do, the *MidiEvent* says when to do it.

① Message type

144 means
NOTE ON

start playing

128 means
NOTE OFF

stop playing

② Channel

Think of a channel like a musician in a band. Channel 1 is musician 1 (the keyboard player), channel 9 is the drummer, etc.

③ Note to play

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

> 0 1 2 3 4 5 6 7 8 → 127

④ Velocity

How fast and hard did you press the key? 0 is so soft you probably won't hear anything, but 100 is a good default.

Change a message

Now that you know what's in a Midi message, you can start experimenting. You can change the note that's played, how long the note is held, add more notes, and even change the instrument.

① Change the note

Try a number between 0 and 127 in the note on and note off messages.

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

② Change the duration of the note

Change the note off event (not the *message*) so that it happens at an earlier or later beat.

```
b.setMessage(128, 1, 44, 100);
MidiEvent noteOff = new MidiEvent(b, 3 );
```

③ 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);
```

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

Version 2: Using command-line args to experiment with sounds

This version still plays just a single note, but you get to use command-line arguments to change the instrument and note. Experiment by passing in two int values from 0 to 127. The first int sets the instrument, the second int sets the note to play.

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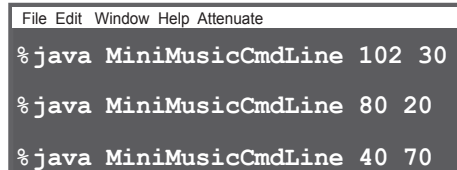
Run it with two int args from 0 to 127. Try these for starters:

File Edit Window Help Attenuate

%java MiniMusicCmdLine 102 30

%java MiniMusicCmdLine 80 20

%java MiniMusicCmdLine 40 70

Where we're headed with the rest of the CodeKitchens

Chapter 15: the goal

When we're done, we'll have a working BeatBox that's also a Drum Chat Client. We'll need to learn about GUIs (including event handling), I/O, networking, and threads. The next three chapters (12, 13, and 14) will get us there.
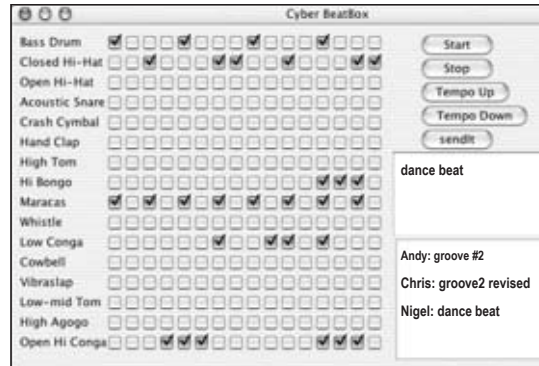
Chapter 12: MIDI events

This CodeKitchen lets us build a little "music video" (bit of a stretch to call it that...) that draws random rectangles to the beat of the MIDI music. We'll learn how to construct and play a lot of MIDI events (instead of just a couple, as we do in the current chapter).

beat one   beat two   beat three   beat four ...

Chapter 13: Stand-alone BeatBox

Now we'll actually build the real BeatBox, GUI and all. But it's limited—as soon as you change a pattern, the previous one is lost. There's no Save and Restore feature, and it doesn't communicate with the network. (But you can still use it to work on your drum pattern skills.)
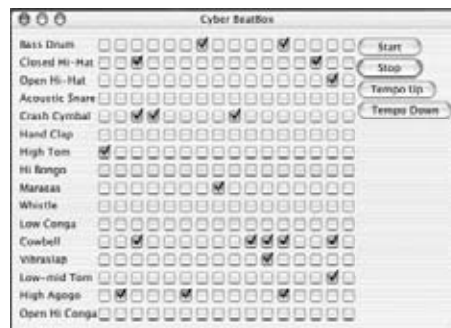
Chapter 14: Save and Restore

You've made the perfect pattern, and now you can save it to a file, and reload it when you want to play it again. This gets us ready for the final version (chapter 15), where instead of writing the pattern to a file, we send it over a network to the chat server.
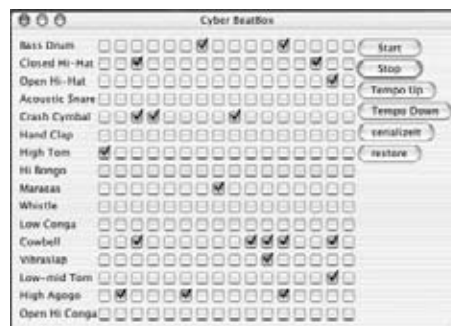
This chapter explored the wonderful world of exceptions. Your job is to decide whether each of the following exception-related statements is true or false.
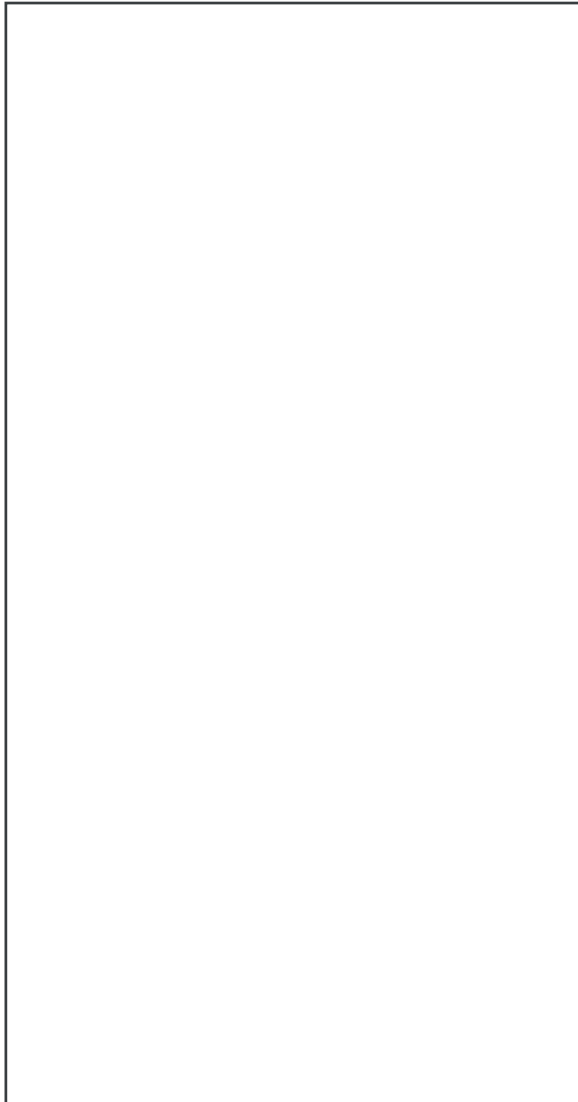
👌 TRUE OR FALSE 👎

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*.

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!

Exercise

```
System.out.print("r");
```

```
try {
```

```
System.out.print("t");
```

```
doRisky(test);
```

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

```
} finally {
```

```
System.out.print("o");
```

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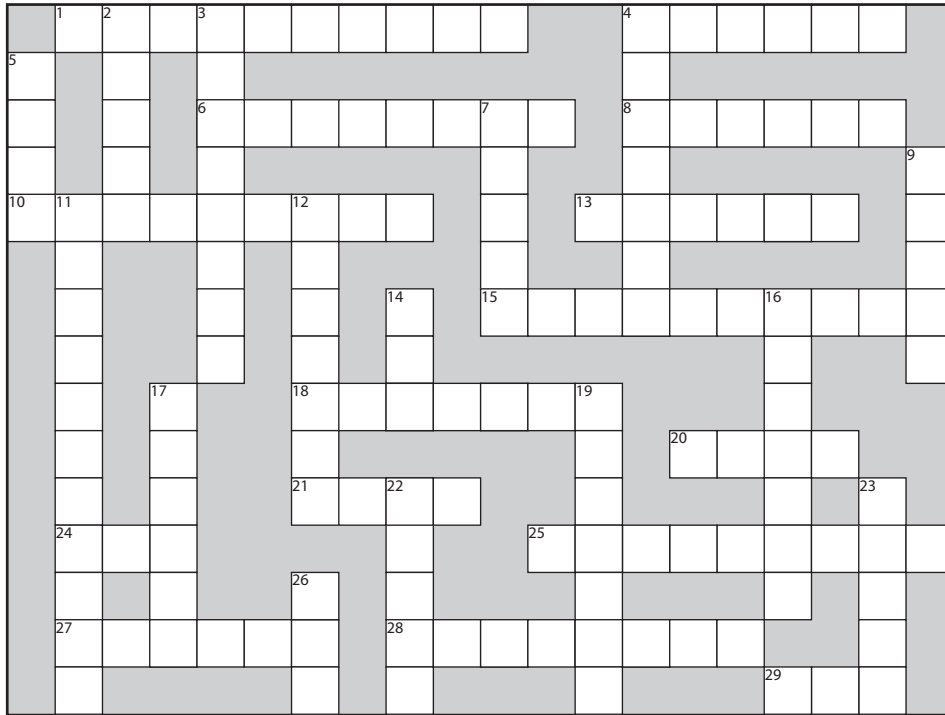
```
public static void main(String [] args) {
    String test = args[0];
```

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

% java ExTestDrive no
throws
```

JavaCross 7.0

You know what to do!

Across

1. To give value
4. Flew off the top
6. All this and more!
8. Start
10. The family tree
13. No ducking
15. Problem objects
18. One of Java's '49'
20. Class hierarchy
21. Too hot to handle
24. Common primitive
25. Code recipe
27. Unruly method action
28. No Picasso here
29. Start a chain of events

Down

2. Currently usable
3. Template's creation
4. Don't show the kids
5. Mostly static API class
7. Not about behavior
9. The template
11. Roll another one off the line
12. Javac saw it coming
14. Attempt risk
16. Automatic acquisition
17. Changing method
19. Announce a duck
22. Deal with it
23. Create bad news
26. One of my roles

More Hints:

Across
6. A Java child
8. Start a method
13. Instead of declare
20. Also a type of collection
21. Quack
27. Starts a problem
28. Not Abstract

Down
2. Or a mouthwash
3. For _____ (not example)
5. Numbers …
9. Only public or default
16. _____ the family fortune
17. Not a 'getter'

Exercise Solutions

Code Magnets

```
class MyEx extends Exception { }

public class ExTestDrive {

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

        System.out.print("t");

        doRisky(test);

        System.out.print("o");

    } catch ( MyEx e) {

        System.out.print("a");

    } finally {

        System.out.print("w");
    }
    System.out.println("s");
  }

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

    if ("yes".equals(t)) {

      throw new MyEx();
    }

    System.out.print("r");

  }
}
```

TRUE OR FALSE

1. False, either or both.

2. False, you can declare the exception.

3. True.

4. False, runtime exception can be caught.

5. True.

6. True, both are acceptable.

7. False, the declaration is sufficient.

8. False, but if it doesn't the JVM may shut down.

9. True.

10. False.

11. True. It's often used to clean-up partially completed tasks.

12. False.

13. False.

14. False, ducking is synonomous with declaring.

15. False, broadest exceptions must be caught by the last catch blocks.

16. False, if you don't have a catch block, you *must* declare.

17. False.

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

% java ExTestDrive no
throws
```

JavaCross Answers

¹A	²S	S	³I	G	N	M	E	N	T		⁴P	O	P	P	E	D	
⁵M		C		N							R						
A		O	⁶S	U	B	C	L	A	⁷S	S	⁸I	N	V	O	K	E	
T		P		T					T		V					⁹C	
¹⁰H	¹¹I	E	R	A	R	¹²C	H	Y		¹³H	A	N	D	L	E	L	
	N			N		H			A		T					A	
	S			C		E		¹⁴T		¹⁵E	X	C	E	P	T	¹⁶I	O	N	S
	T			E		C		R			N					S	
	A		¹⁷S			¹⁸K	E	Y	W	O	R	¹⁹D				S	
	N		E			E				E		²⁰T	R	E	E		
	T		T		²¹D	U	C	²²K		C		R			²³T		
	²⁴I	N	T			A		A		²⁵A	L	G	O	R	I	T	H	M
	A		E		²⁶I			T		A		T			R		
	²⁷T	H	R	O	W	S		²⁸C	O	N	C	R	E	T	E		
	E				A			H						²⁹N	E	W	