CHAPTER – 13

Risky Behavior

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. It's like a device that streams music, but with a few added features. The Sequencer class is in the javax.sound.midi package. So let's start by making sure we can make (or get) a Sequencer object.

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

public class MusicTest1 {
  public void play() {
    try {
      Sequencer sequencer = MidiSystem.getSequencer();
      System.out.println("Successfully got a sequencer");
    }
  }

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

import the javax.sound.midi package

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

Something's wrong!

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

```
File Edit Window Help SayWhat?

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

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

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

See Also:
`getSequencer(boolean)`, `getSynthesizer()`, `getReceiver()`

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

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.

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

public class MusicTest1 {

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

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

Put the risky thing in a "try" block. It's the "risky" getSequencer method that might throw an exception.

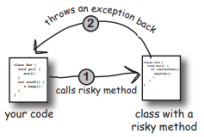
Make a "catch" block for what to do if the exceptional situation happens—in other words, a MidiUnavailableException is thrown by the call to getSequencer().

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.



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.

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:

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

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

BULLET POINTS

- A method can throw an exception when something fails at runtime.
- An exception is always an object of type Exception. (This, as you remember from the polymorphism chapters (7 and 8), 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.
- A method throws an exception with the keyword **`throw`**, followed by a new exception object:

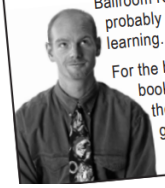
```
throw new NoCaffeineException();
```

- Methods that *might* throw a checked exception ***must*** announce it with a **`throws SomeException`** 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.

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

```
try {
  turnOvenOn();
  x.bake();
} catch (BakingException e) {
  e.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:

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.

EXERCISE

1.TRUE OR FALSE

1. A try block must be followed by a catch and a finally block. - False
2. If you write a method that might cause a compiler-checked exception, you must wrap that risky code in a try/catch block. - False
3. Catch blocks can be polymorphic. - true
4. Only “compiler checked” exceptions can be caught. - False
5. If you define a try/catch block, a matching finally block is optional.- True
6. If you define a try block, you can pair it with a matching catch or finally block, or both.- True
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.- False
8. The main() method in your program must handle all unhandled exceptions thrown to it.- False
9. A single try block can have many different catch blocks.- True
10. A method can throw only one kind of exception.-True
11. A finally block will run regardless of whether an exception is thrown.-True
12. A finally block can exist without a try block.- False
13. A try block can exist by itself, without a catch block or a finally block.- False
14. Handling an exception is sometimes referred to as “ducking.” – False
15. The order of catch blocks never matters.- False
16. A method with a try block and a finally block can optionally declare a checked exception.- False
17. Runtime exceptions must be handled or declared.- False

2.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");
   }
}
```