Software Development (CS2500)

Lecture 35: **Living on the Edge**

M.R.C. van Dongen

January 19, 2011

Contents

1 Outline

This lecture is about *exceptions*. It lays the foundations for a music application. We shall start by looking at the application. Implementing the application requires that we know about *exceptions*. The remainder of the lecture explains exceptions. Next lecture we shall implement a bit of the music app. We shall finish it in our GUI lectures.

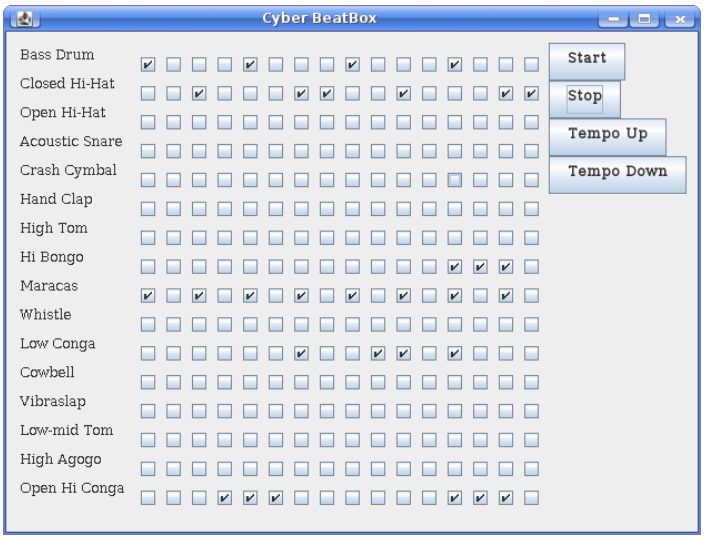

Figure 1: A Java beat box application.

2 The Beat Box

In this lecture we'll lay the foundations for a Java music application. We'll finish the application when we're studying GUIs (Graphical User Interfaces). Figure 1 depicts the application: a beatbox.

You put checkmarks in the boxes for each of the 16 "beats". For example on Beat 1 you click '`Bass Drum`', on Beat 3 you click '`Closed Hi-Hat`', and so on. When you click on the '`Start`' button the different sounds are played at the relevant beats, which are repeated after every cycle of 16 beats. There are also buttons to speed up or slow down the tempo.

3 The Basics

This section studies the basic ingredients for the sound part of our application. At the heart of the application is the JavaSound API (Application Programming Interface).

JavaSound is a collection of classes and interfaces. They are part of the J2SE class library. The library is split into `MIDI` and `Sampled`. As the name suggests it's for sound applications.

We shall only be using `MIDI`. `MIDI` is an acronym for Musical Instrument Digital Interface. It's really a specification on how to play music: play high C, hit it hard, and hold it for 2 beats. The `MIDI` file is read by a `MIDI`-capable instrument: a keyboard, a computer, ... The instrument plays the music by sending it to the speaker. For our beatbox we'll use the Java built-in software instrument which is called *synthesizer*.

4 The Sequencer

Our main object for creating sounds is the Sequencer. The Sequencer is capable of creating and playing back sounds. The following is a first stab at creating the Sequencer.

Don't Try this at Home

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

public class MusicTest {
    public void play( ) {
        Sequencer sythesizer = MidiSystem.getSequencer( );
        System.out.println( "Created Sequencer!" );
    }

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

However, when we compile this simple application we get a compile-time error.

Unix Session

```
$ javac MusicTest
MusicTest.java:5: unreported exception javax.sound.midi.MidiUnavailableException; must be caught or declared to be thrown
        Sequencer sythesizer = MidiSystem.getSequencer( );
                               ^
1 error
$
```

Java refused to turn the Java program into a .class file. The reason for doing this is that the program has an 'unreported exception' which 'must be caught or declared'.

5 Exceptions

In the previous section we got an error stating that our program has an 'unreported exception' which 'must be caught or declared'. The reason for this error is that the MidiSystem class, which returns the Sequencer, does 'risky things', which may go wrong. Before we can resolve the problem in the program that creates our Sequencer we have to learn a bit about risky things.

Let's say you call a method in a class you didn't write.

Java

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

As it turns out the method MidiSystem.getSequencer does something risky, which might not work at runtime.

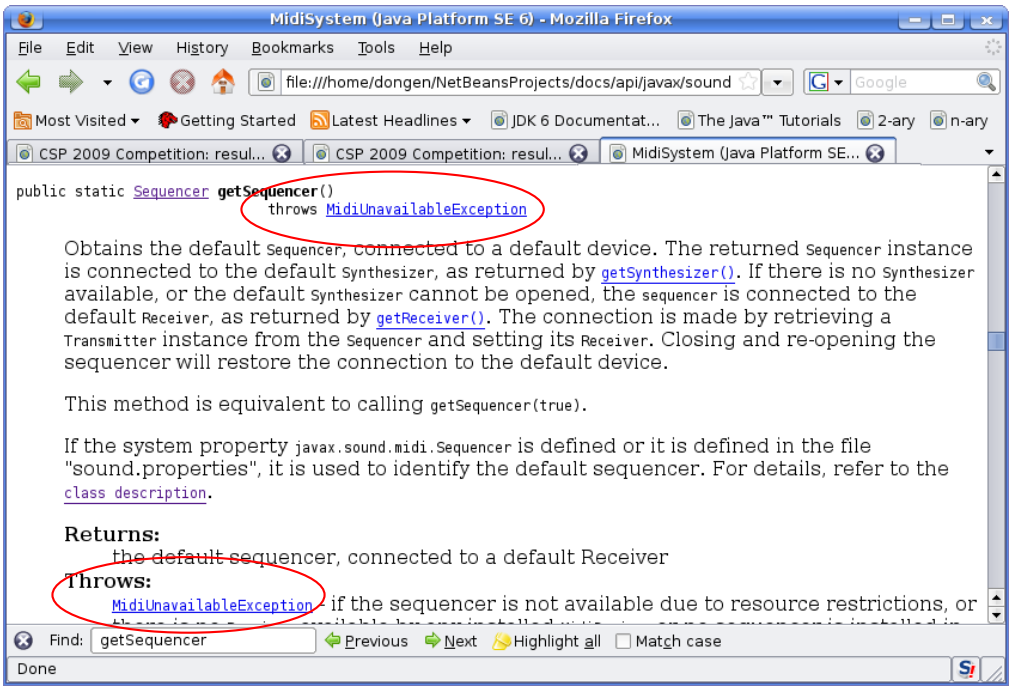

Figure 2: A method `throws` an exception.

```Java
public static Sequencer getSequencer( ) {
   if (serverDown) {
       System.explode( );
   } else {
       return ⟨A Sequencer⟩;
   }
}
```

You need to know the method you're calling is risky. For example, knowing that the method `MidiSystem.getSequencer( )` is risky allows us to prevent things from going wrong when an exceptional event occurs. You do this by writing code that *catches* and *exception*. The result is a safe and robust application.

5.1 Finding Risky Methods

In Java you can recognise risky methods because they *throw exceptions*. (In Java and the Java API you're look for a `throws` clause.) Figure 2 depicts an example.

Methods use *exceptions* to inform the calling code if something bad happens. It is the caller's responsibility to *catch* the exception or *ignore* it. *Catching* the exception means dealing with it. This should be done in a robust way. *Ignoring* the exception means informing the compiler you ignore the exception. This is done by *declaring* the exception. Ignoring an exception doesn't solve it. However, avoiding exceptions doesn't resolve them which is why Java insists that all exceptions should eventually caught.

5.2 Catching Exceptions

In Java you catch an exception by writing a try-catch statement.

- In the try block you call the risky method.
- In the catch block you deal with the exceptions. As we shall see in a moment, you may have several catch blocks.

The following demonstrates how this works.

```Java
public void play( ) {
    try {
        Sequencer sequencer = MidiSystem.getSequencer( );
        ...
    } catch( MidiUnavalableException exception ) {
        System.err.println( "MusicTest: play failed!" );
        ...
    }
}
```

In the try block we write the risky call to MidiSystem.getSequencer( ). We know from the API documentation that the method may throw a MidiUnavalableException exception. The catch block deals with this exception. Since this is the only exception which is thrown, the resulting code is robust.

When the try block is executed at runtime, the catch block will be ignored if no exception occurs. However, if a MidiUnavalableException exception occurs then the catch block will deal with it. The exception is formally assigned to the variable exception and the statements in the body of the catch block are carried out. These statements can use exception but this is not required.

5.3 The Inner Object of Exceptions

Almost everything in Java is an object. So are Exceptions. Figure 3 depicts some of the Exception hierarchy.

5.4 Creating New Exceptions

Creating a new kind of exception is done as usual: by extending the proper Exception (sub)class.

```Java
public class MotherOfAllExceptions extends Exception {
    ...
}
```

5.5 Handling the Error

We've seen how to catch exceptions but we haven't seen how to properly *deal* with the exception. Some clues can be got by studying the exception class hierarchy in Figure 3. The method printStackTrace( ),

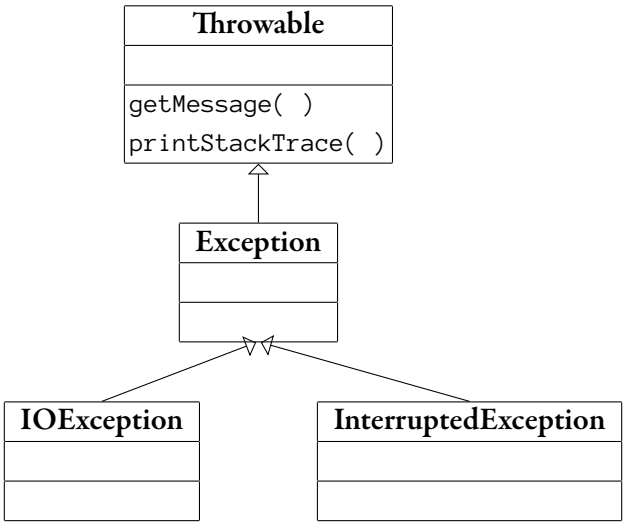

Figure 3: Part of the `Exception` hierarchy.

which is defined in the `Throwable` class, prints a stack trace that leads up to the current method. The method `getMessage( )` returns the details of the exception (this may be `null`). The minimum you arguably should do to handle the error is (1) output the name of the exception, (2) output the reason, and (3) output the stacktrace. When dealing with exceptions, you should print all output `System.err` as this is reserved for error messages.

In the following example the method `risky( )` may throw two exceptions which are called `MotherOfAllExceptions`, and `FatherOfAllExceptions`. These are the only possible exceptions which the method can throw. The method `safe` catches each possible exception and handles it. Notice that it is possible to distinguish between the different possible exceptions: the first `catch` catches the `MotherOfAllExceptions` exception, and the second catches the `FatherOfAllExceptions` exception.

```Java
public void handle( Exception exception ) {
    String cause = exception.getMessage( );
    if (cause != null) {
        System.err.println( cause );
    }
    exception.printStackTrace( );
}

public void safe( ) {
    try {
        risky( );
    } catch (MotherOfAllExceptions exception) {
        handle( exception );
    } catch (FatherOfAllExceptions exception) {
        handle( exception );
    }
}
```

The following gives an idea of the output.

```Unix Session
$ java Risky
MotherOfAllExceptions
        at Risky.risky(Risky.java:10)
        at Risky.safe(Risky.java:26)
        at Risky.main(Risky.java:4)
$
```

The first line of the output is the value which is returned by the instance method `getMessage( )`. In this example this is just the name of the exception. Returning the name of the exception is the default behaviour of the method `getMessage( )` but it may be overridden. The remaining lines are output by the method `printStackTrace( )`.

The last lines are especially useful if you have to debug an application.

Starting at the bottom, and leading to the top, the n-th line shows the n-th method which are on the stack. So for this example, the method `main( )` was called first, then `safe( )`, and finally `risky( )`.

For the bottom $n - 1$ lines, the number in parentheses is the line number of the method call. The remaining number is the line number that threw the exception. So for this example, `safe( )` was called at Line 4 in `Risky.java`, `risky( )` was called at Line 26 in `Risky.java`, and the exception was thrown at Line 10 in `Risky.java`.

5.6 Throwing Exceptions

We've learnt quite a bit about exceptions. We know you can throw them, catch them, or ignore them. We even know how to catch them. It's about time to learn how to throw them.

The following demonstrates how to throw an exception. Methods which throw exceptions report all

thrown exceptions with a `throws` declaration. Inside the method you throw a given exception with a `throw` statement. Of course, you can only throw proper exception object references, which are constructed as usual.

```Java
public void risky( ) throws MotherOfAllExceptions,
                            FatherOfAllExceptions {
    if (allFails( )) {
        throw new MotherOfAllExceptions( );
    } else if (stillDesperate( )) {
        throw new FatherOfAllExceptions( );
    }
}
```

5.7 Ignoring Exceptions

As already mentioned, it is allowed to *ignore* exceptions. This is done by listing all uncaught exceptions with a `throws` declaration. In the following example the method `risky( )` may throw two possible exceptions: `MotherOfAllExceptions` and `FatherOfAllExceptions`. The former exception is caught but the second isn't. Therefore we list `FatherOfAllExceptions` in the `throws` declaration, which is written after the closing parenthesis of the method's argument list.

```Java
public void safeIsh( ) throws FatherOfAllExceptions {
    try {
       risky( );
    } catch (MotherOfAllExceptions exception) {
       // Deal with it.
    }
}
```

In this example there is only one exception in the `throws` declaration. In the event of there being several exceptions, they should all be listed in the `throws` declarations and commas should be used to separate them.

5.8 Finally

The last part of a `try-catch` block may contain an optional `finally` clause. It contains code which should be done regardless of whether an exception occurs or not.

```java
Oven oven = new Oven( );
try {
    oven.on( );
    Dish dish = new Dish( );
    dish.bake( );
} catch (BakingException exception) {
    exception.printStackTrace( );
} finally {
    oven.off( );
}
```

6 For Friday

Study the lecture notes, and study Pages 315–352 (Chapter 10, first half).