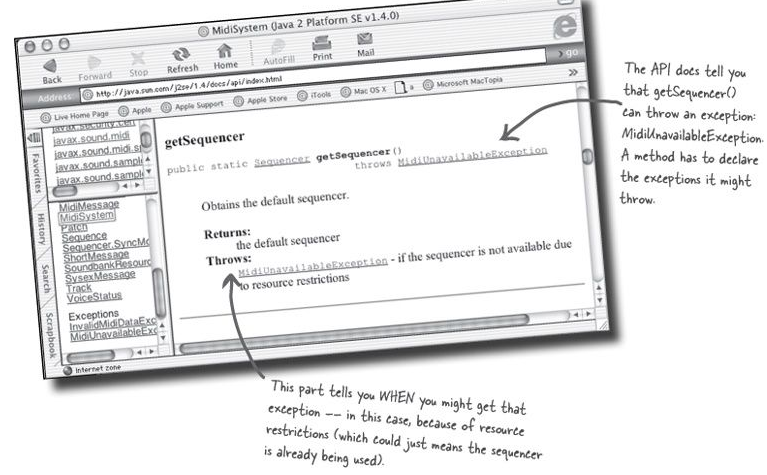
**B”H**

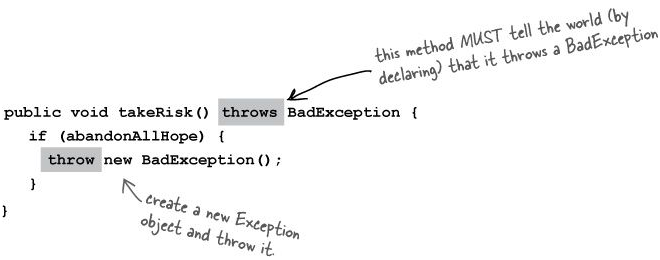
**Chapter 11**

**Exception Handling**

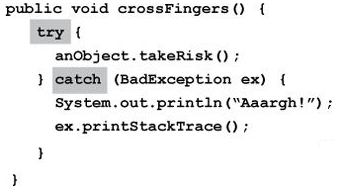
* This and the next chapter explain a lot how MIDI in Java works. It’s not essential so no notes but look there if interested.
* How do you know if a method throws an exception? You find a throws clause in the risky method’s declaration.
* For example the MidiSystem.getSequencer() method takes a risk. It can fail at runtime. So it must ‘declare’ the risk you take when you call it.



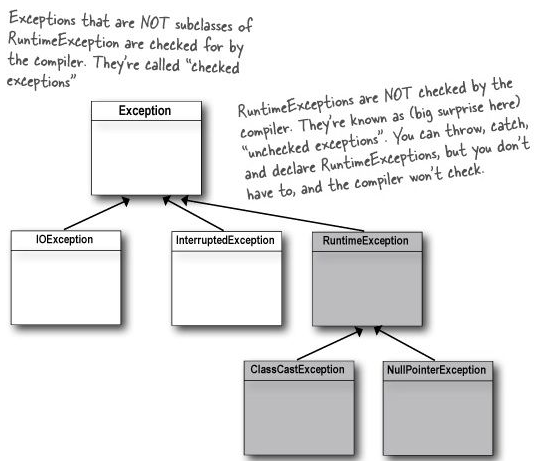
* A **try/catch** block tells the compiler that you know an exceptional thing could happen in the method you’re calling, and that you’re prepared to handle it. That compiler doesn’t care how you handle it; it cares only that you say you’re taking care of it.
* An exception is an object of type Exception (hence can be polymorphic)
* You’ll spend much more of your Java coding time handling exceptions than you’ll spend creating and throwing them yourself.
* One method will catch what another method throws. An exception is always thrown back to the caller.
* Throwing example:



* Calling example:



* The compiler checks for everything except **RuntimeException**

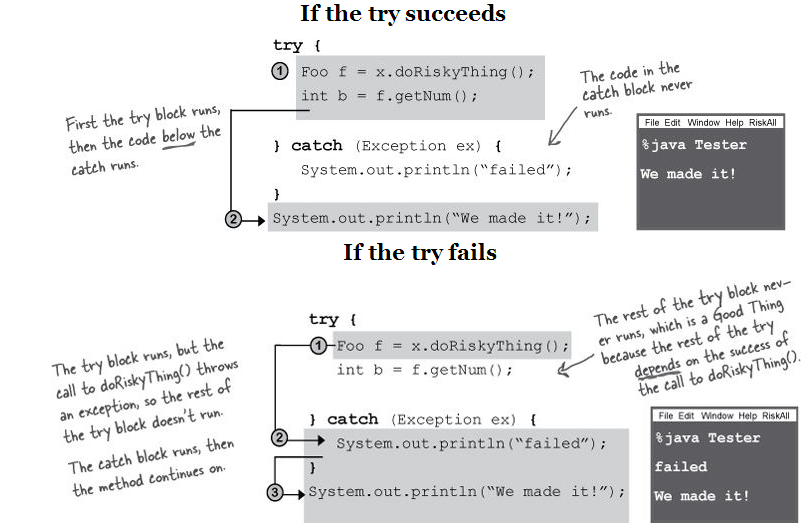


* Some examples of a **RuntimeException**: NullPointerException, DivideByZero, NumberFormatException (from the Integer.parseInt() method) , etc.
* Most RuntimeExceptions come from a problem in your code logic, rather than a condition that fails at runtime in ways that you cannot predict or prevent.

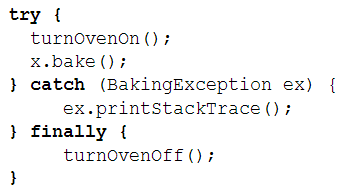
You cannot guarantee the file is there. You cannot guarantee the server is up.

But you can make sure your code doesn’t index off the end of an array (that’s what the .length attribute is for).

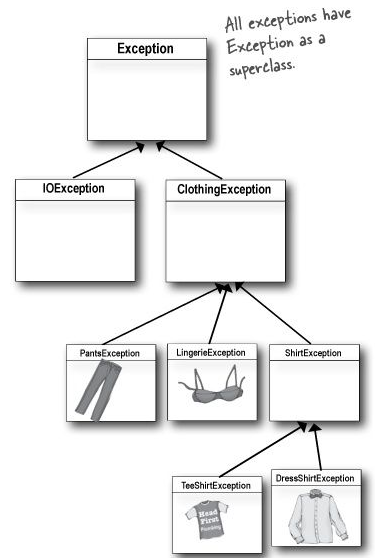
* You WANT RuntimeExceptions to happen at development and testing time. You don’t want to code in a try/catch and have the overhead that goes with it, to catch something that shouldn’t happen in the first place.



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



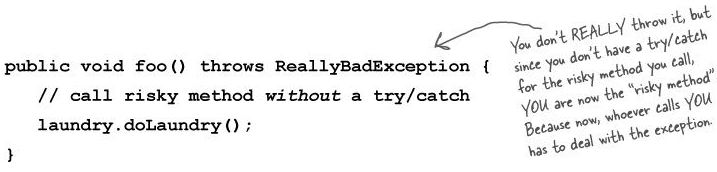
* If the try or catch block has a return statement, finally will still run! Flow jumps to the finally, then back to the return.
* A method can throw multiple exceptions. A method’s declaration must declare all the checked exceptions it can throw
* You can DECLARE and CATCH exceptions using a supertype of the exception
* Write a different catch block for each exception that you need to handle uniquely.



* Multiple catch blocks must be ordered from smallest to biggest. You can’t put bigger baskets above smaller baskets (it won’t compile).



* 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.
* 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.
* If you don’t want to handle an exception, you can duck it by declaring it.
* 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...



* Ducking only delays the inevitable. Sooner or later, somebody has to deal with it.
* If **main()** ducks the exception as well then the exception is thrown to the JVM which will then cause it to shut down
* Some exception rules:
  + You cannot have a **catch** or **finally** without a **try**
  + You cannot put code between the **try** and the **catch**
  + A **try** MUST be followed by either a **catch** or a **finally**
  + A **try** with only a **finally** (no **catch**) must still declare the exception (because it still doesn’t satisfy the “**must handle or declare”** law):

