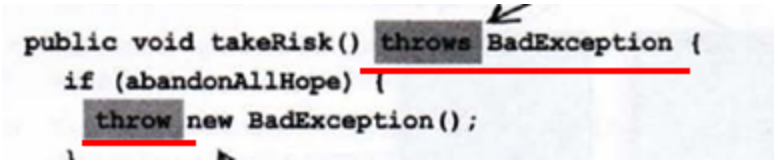
程序运行过程中不能正常秩序的情况称为异常
程序语言提供相应的特殊处理机制称为“异常处理”
For now, just know that when your code calls a risky method—a method that declares an exception, it’s the risky method that throws the exception back to you, the caller.



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

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.)

- 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.

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.

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.

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.

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!

The catch parameter has to be the 'right' exception. If we said 'catch(FileNotFoundException f), the code would not compile, because polymorphically a MidiUnavailableException 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