Exception

Write Your Try-Catch-Finally Statement First:

One of the most interesting things about exceptions is that they define a scope within your program. When you execute code in the try portion of a try-catch-finally statement, you are stating that execution can abort at any point and then resume at the catch. In a way, try blocks are like transactions. Your catch has to leave your program in a consistent state, no matter what happens in the try.

For this reason it is good practice to start with a try-catch-finally statement when you are writing code that could throw exceptions.

Each exception that you throw should provide enough context to determine the source and location of an error.

Use Unchecked Exceptions:

The signature of every method would list all of the exceptions that it could pass to its caller.

The price of checked exceptions is an Open/Closed Principle violation. If you throw a checked exception from a method in your code and the catch is three levels above, you must declare that exception in the signature of each method between you and the catch. This means that a change at a low level of the software can force signature changes on many higher levels.

Encapsulation is broken because all functions in the path of a throw must know about details of that low-level exception. Given that the purpose of exceptions is to allow you to handle errors at a distance, it is a shame that checked exceptions break encapsulation in this way.

Wrapping third-party APIs:

When we define exception classes in an application, our most important concern should be how they are caught.

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  …
}
```

Exception

Sol:

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  …
}
```

Our `LocalPort` class is just a simple wrapper that catches and translates exceptions thrown by the `ACMEPort` class:

```
public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {


      throw new PortDeviceFailure(e);
    }
  }
  …
}
```

Wrappers like the one we defined for ACMEPort can be very useful. In fact, wrapping third-party APIs is a best practice. When you wrap a third-party API, you minimize your dependencies upon it: You can choose to move to a different library in the future without much penalty. Wrapping also makes it easier to mock out third-party calls when you are testing your own code.

Use different classes only if there are times when you want to catch one exception and allow the other one to pass through.

Don’t Return Null:

Exception

```
public void registerItem(Item item)
 {
        if (item != null) {
                ItemRegistry registry = peristentStore.getItemRegistry();
        if (registry != null) {
                Item existing = registry.getItem(item.getID());
        if (existing.getBillingPeriod().hasRetailOwner()) {
                existing.register(item);
                        }
                }
        }
}
```

If you work in a code base with code like this, it might not look all that bad to you, but it is bad! When we return null, we are essentially creating work for ourselves and foisting problems upon our callers. All it takes is one missing null check to send an application spinning out of control. Did you notice the fact that there wasn't a null check in the second line of that nested if statement? What would have happened at run time if persistentStore were null? We would have had a NullPointerException at run time, and either someone is catching NullPointerExceptionat the top level or they are not. Either way it's bad. What exactly should you do in response to a NullPointerExceptionthrown from the depths of your application?

If you are tempted to return null from a method, consider throwing an exception or returning a SPECIALCASE object instead. If you are calling a null-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.

```
public class PerDiemMealExpenses implements MealExpenses {
public int getTotal() {
        // return the per diem default
        }
}
```

This is called the SPECIALCASEPATTERN[Fowler]. You create a class or configure an object so that it handles a special case for you. When you do, the client code doesn't have to deal with exceptional behavior. That behavior is encapsulated in the special case object.

Don't Pass Null:

Returning null from methods is bad, but passing null into methods is worse.
We could use a set of assertions.