1. **Use Exceptions Rather Than Return Codes (Sử dụng ngoại lệ hơn là trả về Codes)**

Bạn có thể thiết lập một error flag hoặc trả về một mã lỗi mà người gọi có thể kiểm tra. Vấn đề ở đây là nó lộn xộn với người gọi.

public class DeviceController {

...

public void sendShutDown() {

DeviceHandle handle = getHandle(DEV1);

// Check the state of the device

if (handle != DeviceHandle.INVALID) {

// Save the device status to the record field

retrieveDeviceRecord(handle);

// If not suspended, shut down

if (record.getStatus() != DEVICE\_SUSPENDED) {

pauseDevice(handle);

clearDeviceWorkQueue(handle);

closeDevice(handle);

} else {

logger.log("Device suspended. Unable to shut down");

}

} else {

logger.log("Invalid handle for: " + DEV1.toString());

}

}

...

}

Code tốt hơn bởi vì có hai quan hệ lộn xộn ở đây, thuật toán cho thiết bị tắt máy và xử lý lỗi, và bây giờ nó đã được tách ra.

public class DeviceController {

...

public void sendShutDown() {

try {

tryToShutDown();

} catch (DeviceShutDownError e) {

logger.log(e);

}

}

private void tryToShutDown() throws DeviceShutDownError {

DeviceHandle handle = getHandle(DEV1);

DeviceRecord record = retrieveDeviceRecord(handle);

pauseDevice(handle);

clearDeviceWorkQueue(handle);

closeDevice(handle);

}

private DeviceHandle getHandle(DeviceID id) {

...

throw new DeviceShutDownError("Invalid handle for: " + id.toString());

...

}

...

}

# 2. Write Your Try-Catch-Finally Statement First

Test này thất bại nó chưa ném ra ngoại lệ. Tiếp theo chúng ta thay đổi hiện thực, do đó mà nó cố gắng truy cập vào một tập tin không hợp lệ. Thao tác này thows một exception:

public List<RecordedGrip> retrieveSection(String sectionName) {

try {

FileInputStream stream = new FileInputStream(sectionName)

} catch (Exception e) {

throw new StorageException("retrieval error", e);

}

return new ArrayList<RecordedGrip>();

}

Test passes bởi vì nó đã bắt lấy một exception. Tái cấu trúc lại:

public List<RecordedGrip> retrieveSection(String sectionName) {

try {

FileInputStream stream = new FileInputStream(sectionName);

stream.close();

} catch (FileNotFoundException e)

throw new StorageException("retrieval error”, e);

}

return new ArrayList<RecordedGrip>();

}

### **Use Unchecked Exceptions (Sử dụng ngoại lệ không được kiểm tra)**

**Checked**: là các ngoại lệ được kiểm tra tại thời điểm biên dịch.

**Unchecked**: là những ngoại lệ không được kiểm tra tại thời điểm biên dịch.

Các ngoại lệ được kiểm tra đôi khi có thể hữu ích nếu bạn đang viết một thư viện quan trọng: Bạn phải nắm bắt chúng. Nhưng trong phát triển ứng dụng nói chung, chi phí phụ thuộc lớn hơn lợi ích.

# 3. Provide Context with Exceptions (Cung cấp bối cảnh cho ngoại lệ)

Mỗi trường hợp ngoại lệ bạn nên cung cấp ngữ cảnh để xác định nguồn gốc và vị trí của lỗi. Trong Java, bạn sẽ nhận được một stack truy tìm (trace) từ bất kỳ ngoại lệ nào. Tuy nhiên stack đó không cho bạn biết ý nghĩ của các ngoại lệ không thành công.

Tạo một thông điệp báo lỗi và truyền cho chúng với ngoại lệ của bạn. Đề cập đến các hành động gây ra lỗi và loại lỗi.

# 4. Define Exception Classes in Terms of a Caller's Needs the Normal Flow

Có nhiều cách để phân loại lỗi.

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 {

…

}

Đơn giản code bằng wrapping API mà chúng ta gọi và đảm bảo rằng nó sẽ trả về một ngoại lệ chung.

LocalPort port = new LocalPort(12);

try {

port.open();

} catch (PortDeviceFailure e) {

reportError(e);

logger.log(e.getMessage(), e);

} finally {

…

}

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);

}

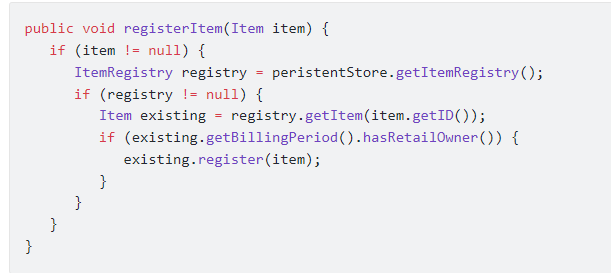
}

…

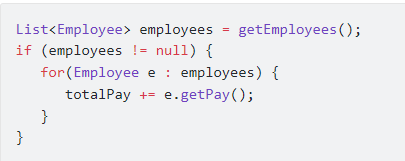
}

# 6. Don't Return Null

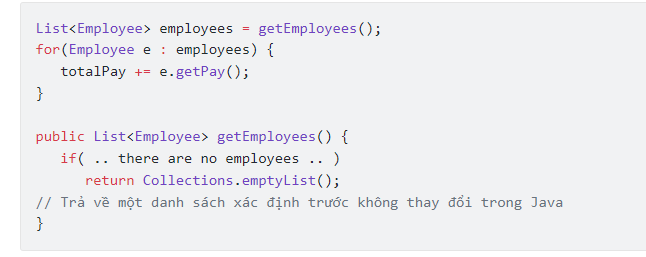
Khi chúng ta trả về null, chúng ta tự tạo công việc cho bản thân và thêm vấn đề cho người gọi nó. Dễ thown exception NullPointerException.



Ví dụ bạn có đoạn code sau:

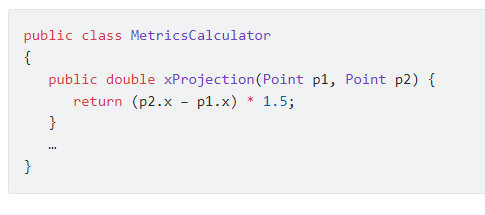


Nếu trả về một danh sách rỗng, chúng ta có thể dọn dẹp code như sau:



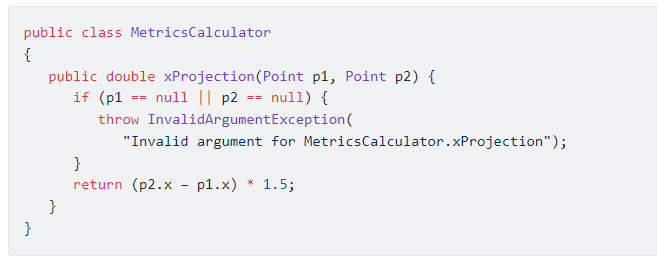
# 7. Don't Pass Null

Trả về null từ phương thức đã xấu, nhưng vượt qua null còn tồi tệ hơn.

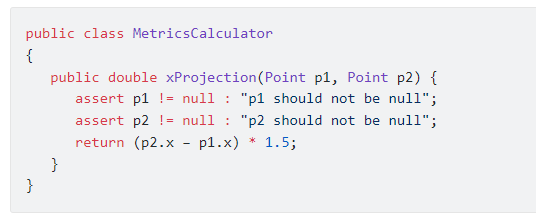


Điều gì xảy ra nếu đi qua một tham số gán null? calculator.xProjection(null, new Point(12, 13));

Chúng ta sẽ có một NullPointerException. Sửa chữa lại:



Có một thay thế khác tốt hơn:



# 8. Conclusion

Code sạch là code có thể đọc được, nhưng nó cũng cần phải mạnh mẽ. Đây không phải là điều mâu thuẫn. Chúng ta có thể viết code sạch và mạnh mẽ nếu chúng ta thấy được xử lý lỗi là một mối quan tâm riêng, đôi khi có thể xem nó như không phụ thuộc với logic cơ bản của chúng ta. Để đến mức độ chúng ta làm được điều đó, chúng ta cần lý giải về nó một cách độc lập, và chúng ta có thể tiến tới bước tiến lớn trong việc bảo trì code của chúng ta.