Clean Code

Chapter 7: Error Handling

Author: Hamed Damirchi
hameddamirchi32@gmail.com
github.com/hamed98
linkedin.com/in/hamed-damirchi-ba4085178/

Author: Hamed Damirchi

مقدمه

خیلی وقتا خوندن کد و نگهداریش به خاطر error handling های پراکنده که استفاده شده، سخت میشه.

Error handling مهمه، اما اگر باعث مبهم شدن و ناخوانا تر شدن کد بشه، اشتباهه!

تو این بخش قراره یاد بگیریم چطور error handling رو بنویسیم، به طوری که به تمیز بودن و robust بودن کد لطمه ای وارد نشه.

Author: Hamed Damirchi

به جای کد خطا، از اکسپشن ها استفاده کنید

قبلا، خیلی از زبان ها Exception نداشتن و واسه این که بگیم این جا خطا اتفاق افتاده، اینو با کد یا یه فلگ مشخص میکردیم و کلا مدیریت اکسپشن ها سخت بود. مثلا:

```
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());
    }
  }
  ...
}
```

یه مشکل اساسی این نوع هندلینگ خطا، اینه که فراخوانی کننده تابع رو آشفته میکنه. چرا؟ چون باید بعد از کال کردن هر تابع، همه اکسپشن های ممکن رو هم چک کنه و خب ممکنه بعضیاش فراموش بشه

همچنین مشکل دیگه اینه که ارور هندلینگ با لاجیک قاطی میشه که اشتباهه.

اما اگر این اکسپشن throw بشه، کال کننده تابع از شر این مشکل ها خلاص میشه!

```
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());
      ...
    }

    ...
}
```

این خیلی بهتر شد، چون دو بخش الگوریتم برنامه واسه shut down و همچنین بخش کنترل خطا، از هم مجزا شد و الان میتونی هر کدوم رو مجزا کدش رو بخونی.

Author: Hamed Damirchi

همیشه Try/Catch/Finally رو اول بنویس

اگر میخوای کدی بنویسی که امکان داره اکسپشن پرتاب کنه، بخش try/catch/finally رو اول بنویس.

فرض کنید از روش Test Driven Development استفاده میکنیم و ابتدا این تست رو مینویسیم:

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
  sectionStore.retrieveSection("invalid - file");
}
```

این تست، سعی میکنه یه ورودی اشتباه بده و انتظار داره که یه اکسپشن پرتاب بشه.

اما اگه ابتدا از try/catch استفاده نکنیم و تابع ما این شکلی باشه:

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  // dummy return until we have a real implementation
  return new ArrayList<RecordedGrip>();
}
```

هیچ اکسپشنی پرتاب نمیشه و تست ما fail میشه.

بعد کد تابع رو عوض میکنیم و این شکلی مینویسیم:

```
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>();
}
```

حالا تست هم درست کار میکنه!

اما بهتره باز هم ریفکتور کنیم و اکسپشن رو باریک تر کنیم:

```
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>();
}
```

همیشه سعی کنید تست هایی بنویسید که حالت های استثنا رو در نظر میگیرن. و بعد به تابع (handler) ، هندل کردن این اکسپشن ها رو هم اضافه کنید. واسه همین بهتره اول بخش های try/catch/finally نوشته بشن.

Author: Hamed Damirchi

از Unchecked Exception ها استفاده کنید(مخصوص جاوا)

در جاوا امکان استفاده از checked exception ها هست، به این معنی که موقع تعریف تابع، همه ی اکسپشن هایی که این تابع ممکنه پرتاب کنه هم باید در signature آن تابع اضافه بشه.

Author: Hamed Damirchi

کلاس درست کنید و Exception را آن جا هندل کنید

```
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 {
  …
}
```

به جای این کد، بهتره یه wrapper درست کنیم و اکسپشن رو اون جا هندل کنیم، طبق چیزی که نیاز داریم(اسلاید بعدی)

Author: Hamed Damirchi

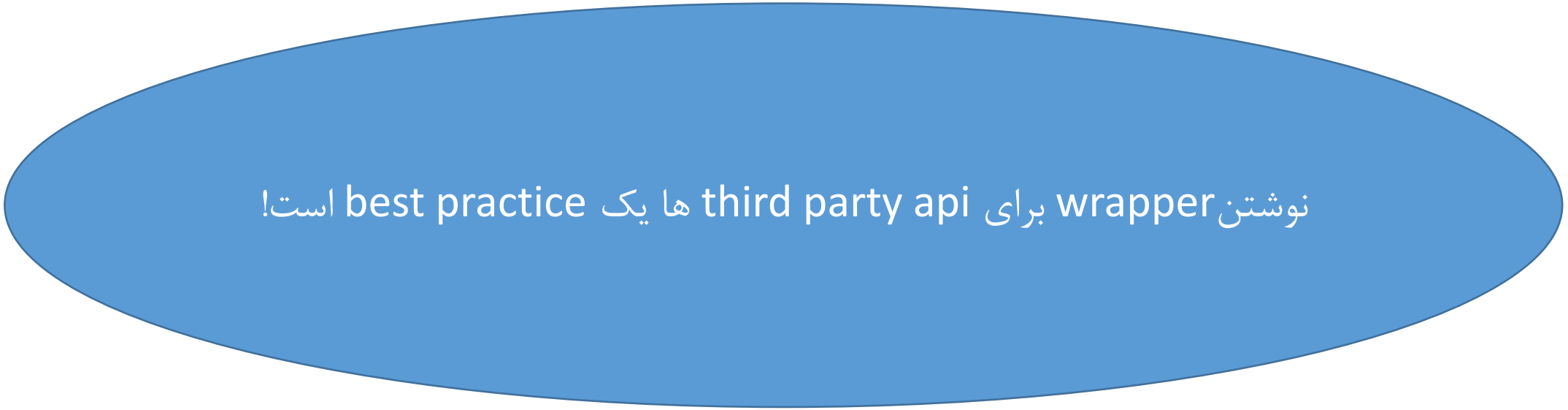
```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  …
}
```

```
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);
    }
  }
  …
}
```

استفاده از چنین wrapper هایی میتونه خیلی مفید باشه. در واقع:



علتش اینه که با این کار وابستگی کد تو به اون api به حداقل ممکن میرسه و میتونی هر زمان که لازم بود، اون api رو با یه api دیگه عوض کنی (فقط wrapper ها رو عوض میکنی) بدون این که به بقیه کد دست بزنی.
همچنین میتونی میزان کال کردن api ها رو بررسی کنی و ...

Author: Hamed Damirchi

جریان نرمال را تعریف کنید

اگر توصیه هایی که در ادامه گفته خواهند شد را انجام دهید، business logic و error handling از هم جدا میشوند.

یک مثال: فرض کنید قطعه کد زیر، از یک نرم افزار محاسبه صورت حساب برداشته شده است:

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

ملاحظه میکنیم که لاجیک بیزنس ما با بخش کنترل خطا در هم ادغام شده است که این خوب نیست.

در اسلاید بعد قطعه کد اصلاح شده آورده شده است.

Author: Hamed Damirchi

جریان نرمال را تعریف کنید

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

تابع getMeals برای حالت استثنا یک شیء مجزا برمیگرداند.

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

یعنی شیء expense در حالت استثنا، یک آبجکت متفاوت، اما با همان ساختار قبلی است و فراخوانی کننده تابع getMeals اصلا متوجه این موضوع نمیشود.

Author: Hamed Damirchi

جریان نرمال را تعریف کنید (الگوی حالت خاص)

در مثالی که استفاده شد، از الگوی Special Case Pattern استفاده شده است.

در این الگو، برای حالت های خاص یک کلاس مجزا تعریف میشود و هنگام اتفاق افتادن این حالت خاص، یک شیء از آن کلاس به فراخوانی کننده تابع برگردانده میشود. در این صورت فراخوانی کننده تابع (کلاینت) درگیر ین حالت های خاص نمیشود.

Author: Hamed Damirchi

NULL ریترن نکنید

ریترن مقدار نال، باعث میشه کد شما این شکلی بشه:

```
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);
      }
    }
  }
}
```

و این کد کثیفه!.

NULL ریترن نکنید. پس چی کار کنیم؟!

اگه نال ریترن کنید مجبورید همیشه چک کنید مقدار نال هست یا نه، ممکنه یادتون بره این چک کردن و ... . مثلا این جا اگه persistantStore نال باشه چی؟ تو ران تایم nullPointerException پرتاب میشه و اصلا خوب نیست. **بهتره به جای ریترن نال، exception پرتاب کنید یا مثل بحث قبلی از Special Case object استفاده کنید.**

اگر از یک api استفاده میکنید که ممکنه نال برگردونه، خودتون یه wrapper درست کنید واسش و داخل اون این حالت رو چک کنید و exception پرتاب کنید یا یک special case object ریترن کنید.

Author: Hamed Damirchi

NULL ریترن نکنید – مثال جایگزین

به جای این کد:

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

بهتره از این کد استفاده کنید:

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

و تعریف تابع getEmployees() رو عوض کنید:

```
public List<Employee> getEmployees() {
  if( .. there are no employees .. )
    return Collections.emptyList();
}
```

Author: Hamed Damirchi

NULL پاس ندید

Return کردن مقدار نال بده، اما پاس دادن نال به یه تابع بدتره!

هیچ موقع نال پاس ندید، مگر این که دارید از یه api خارجی استفاده میکنید و مجبورید.

فرض کنید این تابع رو داریم:

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x - p1.x) * 1.5;
  }
  …
}
```

و این جوری کال میکنیم:

```
calculator.xProjection(null, new Point(12, 13));
```

بله! Null pointer exception پرتاب میشه!

شاید با خودتون بگید خب این طوری هندل میکنیم:

```
public double xProjection(Point p1, Point p2) {
  if (p1 == null || p2 == null) {
    throw InvalidArgumentException(
      "Invalid argument for MetricsCalculator.xProjection");
  }
  return (p2.x - p1.x) * 1.5;
}
```

درسته این بهتر از پرتاب کردن null pointer exception هست، اما یادتون نره، به هر حال باید یه هندلر واسه InvalidArgumentException تعریف کنیم. این هندلر قراره چی کار کنه دقیقا؟

استفاده از assert هم همینطو، به هر حال در صورت مسئله تفاوتی ایجاد نمیشه

پس بهتره کلا نال پاس ندید که درگیر این داستانا نشید!

نتیجه گیری

کد تمیزدر عین خوانا بودن باید قدرتمند و قابل نگهداری باشه (ترجمه robust) و این دو هدف، متضاد هم نیستن. ما میتونیم با جدا کردن بخش منطق برنامه و بخش کنترل استثنا، کد های robust بنویسیم. با این کار قدم بزرگی در قابل نگهداری بودن برنامه خود برمیداریم.

Author: Hamed Damirchi
hameddamirchi32@gmail.com
github.com/hamed98
linkedin.com/in/hamed-damirchi-ba4085178/