Welcome to Code 4 Greater Good. For this course, I will assume all of you already know how to code and today and tomorrow I will teach you principles to write better code. I also assume you know Java. You may or may not have already heard of some of the concepts I will cover, and I will not be talking about certain topics. But my intention is not to completely cover Clean Code in its entirety as a theory and practice, but to give you the real hard skills to practise the principles in your day-to-day work, with examples to help you along. I am doing this course because I realised that there are a lot of ISTD graduates who don't know what it means to code well, myself included. Disclaimer is that I have never been a pro software developer for a company so if you don't trust me, you are free to leave at any time. The book I have adapted (stolen) from is called Clean Code by Robert Martin, I highly suggest you get it and read it. But if you do not have the time, I guess this will serve as a TLDR.

Then I will go through some case studies. If you guys have a project you'd like to be cleaned up and don't mind the rest of the class seeing it, I'll welcome you to present it to the class and let us see how we can apply the concepts to clean it up.

But first, I'd like to begin by saying that there is a difference between cleanliness for your eyes and cleanliness for your brain. PEP8, the formatting standard for Python code ensures that your code conforms to some standard of readability on a standard screen, or clean for the reader's eyes. It does not ensure that your code is clean for your brain (i.e. easy to understand). Imagine organic food, which is good for your body but possibly not eye-pleasing, and fast food which is visually appealing but bad for your body. We want to make organic food that looks/tastes good, so that we eat it more often and thus become healthier. Similarly, the code you write will need to be both pleasing for the eyes to ingest and easy for the brain to digest. We will focus on making it easy for brain in this lesson. To help with PEP8, there are linters like Flake8.

Naming

Give meaningful names - these should be nouns for variables that succinctly describe what the object or number **is** or **represents**.
Functions should be verbs that similarly succinctly describes what the function **does**.
The key to finding meaningful name is to understand the **intention** behind the variable or function. Explicitly revealing the intention in the name results in meaningful names.

Meaningful names answer questions like:

1. What kinds of things are in `the_list`?
2. What is the significance of the zero-th subscript of `the_list`?
3. What is the significance of the value 4 being assigned to `this_variable`?
4. How is this list that is being returned used?

Disinformation

Avoid disinformation by not including words that are used commonly elsewhere i.e. `ls`. Also avoid using names that are too similar-looking i.e. `XYZControllerForEfficientHandlingOfStrings` and `XYZControllerForEfficientStorageOfStrings`.

Similar spelling for similar concepts is *information*, inconsistent spelling is *disinformation*.

```
int a = l;
if ( O == l )
    a = O1;
else
    l = 01;
```

Using 0 and capital o, small l and 1 in names together is a big no-no.

Meaningful distinctions

Don't just create names to satisfy the compiler i.e. using `klass` for a new object because `class` was already used.

Numbered naming i.e. `a1`, `a2`,... are non-informative. Consider below:

```
public static void copyChars(char a1[], char a2[]) {
    for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
    }
}
```

Just using `source` and `destination` would be better.

Noise words are redundant. Words like `variable` or `object` should never be in the name. `table` should never be in the name of a table.
i.e. `NameString` should just be `Name`, because names are always strings. If it can be a number, using the word `Name` in the variable name is disinformation. Or perhaps having `Customer` and `CustomerObject`, which would best represent the customer's payment history (or whatever other purpose this object is being used for)?

Another negative example:

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Which to use? Distinguish names so the reader knows the value of the difference.

Pronouncable names

If you can't pronounce the names, you can't have intelligible conversations about it. Compare:

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
    /* ... */
};
```

With:

```
class Customer {
    private Date generationTimestamp;
    private Date modificationTimestamp;
    private final String recordId = "102";
    /* ... */
};
```

Searchable names

Single letter names are hard to find in a large body of text. Length of a name should correspond to the size of its scope (single letter names belong in short methods as local variables). i.e. `MAX_CLASSES_PER_STUDENT` is better than hard-coding a constant value everywhere it is used (like 7, if that is the maximum number of classes a student can take). Searchable names means you can find where it is being used and what it affects.

Avoid mental mapping

Reader should not have to translate variable names to terms they know. Problem arises from using neither problem domain terms nor solution domain terms. Mental mapping shows you are a smart programmer, but professional programmers know that clarity is king.

Don't need to put in types in the name. Also in general, there is no need for prefixes in names.

Class names

Should be nouns like `Customer`, `WikiPage`, `Account`, and `AddressParser`. Avoid names like `Manager`, `Processor`, `Data`, or `Info` in a class name.

Method names

Should be verbs like `postPayment`, `deletePage`, or `save`.Accessors, mutators and predicates should be prefixed with `get`, `set` and `is`, respectively.

```
String name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

Don't be cute or pun

Don't use colloquial language or slang, like using `whack()` for `kill()`. Puns are terms that look like something but do something else i.e. naming a function `add` because it is used everywhere when in fact nothing is being added.

Use solution or problem domain names

People who will read your code will be programmers. Use CS, algorithmic, pattern, math, etc. names. `AccountVisitor` or `JobQueue` are standard enough names for another programmer to understand what it means. Choose technical names where possible so reader will not have to ask domain expert about what each term means (if written with problem domain names). But sometimes code will have more to do with problem domain, so names should be drawn from there. Separate solution and problem domain names.

Add meaningful context

Add context by placing names in well-named classes, functions, or namespaces. Last resort is to use a prefix. i.e. `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state`, and `zipcode` occurring together look like they should be put into a class called `Address`.

If you have an application called “Statistical Genome Decoder”, don’t prefix every function, class and variable with `SGD`. Doing this makes it hard for the IDE to suggest (it will suggest every function, class and variable when you press `S`).

Functions

First line of organisation in any program. Consider the function below and see how much of it you can understand in 3 minutes. Or skip past the highlighted part and just know it's long and convoluted.

```
public static String testableHtml(
                     PageData pageData, boolean includeSuiteSetup)
              throws Exception {
    WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup = PageCrawlerImpl.getInheritedPage(

SuiteResponder.SUITE_SETUP_NAME, wikiPage );
            if (suiteSetup != null) {
                WikiPagePath pagePath =
suiteSetup.getPageCrawler().getFullPath(suiteSetup);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -setup .")
.append(pagePathName) .append("\n");
            }
        }
        WikiPage setup = PageCrawlerImpl.getInheritedPage("SetUp",
wikiPage);
        if (setup != null) {
            WikiPagePath setupPath =
wikiPage.getPageCrawler().getFullPath(setup);
            String setupPathName = PathParser.render(setupPath);
            buffer.append("!include -setup .").append(setupPathName)
.append("\n");
        }
    }
    buffer.append(pageData.getContent());
    if (pageData.hasAttribute("Test")) {
        WikiPage teardown =
PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
        if (teardown != null) {
            WikiPagePath tearDownPath =
wikiPage.getPageCrawler().getFullPath(teardown);
```

```
            String tearDownPathName =
PathParser.render(tearDownPath);
            buffer.append("\n") .append("!include -teardown .")
                                            .append(tearDownPathName)
.append("\n");
        }
        if (includeSuiteSetup) {
            WikiPage suiteTeardown =
PageCrawlerImpl.getInheritedPage(

SuiteResponder.SUITE_TEARDOWN_NAME, wikiPage);
            if (suiteTeardown != null) {
                WikiPagePath pagePath =
suiteTeardown.getPageCrawler()

.getFullPath (suiteTeardown);
                String pagePathName = PathParser.render(pagePath);
                buffer.append("!include -teardown .")
.append(pagePathName) .append("\n");
            }
        }
    }
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
}
```

Phew. Now try this.

```
public static String renderPageWithSetupsAndTeardowns(
                     PageData pageData, boolean isSuite)
              throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
        WikiPage testPage = pageData.getWikiPage();
        StringBuffer newPageContent = new StringBuffer();
        includeSetupPages(testPage, newPageContent, isSuite);
        newPageContent.append(pageData.getContent());
        includeTeardownPages(testPage, newPageContent, isSuite);
        pageData.setContent(newPageContent.toString());
    }
    return pageData.getHtml();
}
```

Better? Even if you don’t understand the details, you understand what’s going on.

Small!

Two to four lines. Blocks within if statements, else statements, while statements, and so on should be one line long. Probably that line should be a function call. Adds documentary value by giving the function inside the block a nice descriptive name.

The function above can actually be condensed further to:

```
public static String renderPageWithSetupsAndTeardowns(
                     PageData pageData, boolean isSuite)
           throws Exception {
   if (isTestPage(pageData))
      includeSetupAndTeardownPages(pageData, isSuite);
   return pageData.getHtml();
}
```

Do one thing

Functions should do one thing. They should do it well. They should do it only.

So what is “one thing”? That function looks like it’s actually doing 3 things. You can usually describe what the function is doing in a TO string “TO renderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.”

The function performs the steps one level below the stated name. This allows us to decompose it into stepwise abstractions of the big concept which is the program. The other two functions do things that are more than one level below the stated level of abstraction. If your function has sections, it is obviously not doing one thing.

One Level of Abstraction per Function

Statements inside a function should be on the same level of abstraction.
High level - *`getHTML()`*
Mid level - `String pagePathName = PathParser.render(pagePath);`
Low level - `.append("\n")`
Don’t mix the levels in functions, or reader will not be able to differentiate **detail** from **concept**. Worse, they will start inserting more details into functions.

Stepdown rule

Code should read like a story from top to bottom. For example, the below TO statements describe the stepwise abstractions of nested functions.

To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.
To include the setups, we include the suite setup if this is a suite, then we include the regular setup.
To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.
To search the parent. . .

Learning to do this is key to keep your functions doing "one thing".

Switch statements

Does *N* things. Sometimes it can't be helped, and when this happens, the statement should only occur once for a particular class (in the constructor, rather than switching between different operations inside methods of subclasses). Bury this switch statement in a polymorphic constructor which creates subclasses that implement the same functions differently.

```
public abstract class Employee {
    public abstract boolean isPayday();
    public abstract Money calculatePay();
    public abstract void deliverPay(Money pay);
}

public interface EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws
InvalidEmployeeType;
}

public class EmployeeFactoryImpl implements EmployeeFactory {
    public Employee makeEmployee(EmployeeRecord r) throws
InvalidEmployeeType {
        switch (r.type) {
            case COMMISSIONED: return new CommissionedEmployee(r) ;
            case HOURLY: return new HourlyEmployee(r);
            case SALARIED: return new SalariedEmploye(r);
            default: throw new InvalidEmployeeType(r.type);
        }
    }
}
```

Use descriptive names

Each function/routine should turn out to be pretty much what you expected.
Don't be afraid to use long names. Long, descriptive name is better than both a short, ambiguous one as well as a long, descriptive comment.
Spend time on naming. Choose a convention that allows multiple words to describe what a function does and stick to it - be consistent.

Function arguments

Arguments exist at different levels of abstraction than where they are passed, so it creates complication when you have to understand the context of where each argument came from inside a lower-level function. For testing, having more arguments means you have to come up with test cases that include all the different combinations of arguments. Reduce number of arguments if possible. Try not to have more than 3 arguments.

Common monadic forms (single argument)

Performing a transformation on the argument and returning it or asking a question about it i.e. `fileExists("MyFile");`.
Passing an event to alter state of the system - must be clear that it is an event.
No flag (boolean) arguments - indicates that function does more than one thing depending on the flag.
No output arguments - causes confusion because arguments are usually "inputs" and someone unfamiliar with code will have to search for the definition of the function to understand that it is an output argument.

Dyadic Functions

Some functions lend themselves well to dyadic functions i.e. Point(0.0, 0.0); because there is a natural ordering. Others do not and require practise to get accustomed to i.e. assertTrue(expected, actual);. Know the cost and convert to monadic functions where possible i.e. make an argument a member attribute.

Triads

Try not to do this. Just no. Ok it's up to you.

Argument objects

If an argument can be abstracted into a higher level concept that deserves a name, do it. i.e.

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Argument lists

Ok if all elements in list are treated in the same way i.e. the String.format method:
String.format("%s worked %.2f hours.", name, hours);

Verbs and keywords

Name well, ‘nuff said. Ok like if you have this function `writeField(name)`, it’s clear that `name` is a field and it is being written. Nice verb/noun pair for the function name.

Have no side effects

Side effects are lies i.e. `public boolean checkPassword(String userName, String password)` should only check the password and return `true` or `false`, it should not perform a login.

Output arguments

Already mentioned, don’t do it i.e. it’s not so clear what `appendFooter(s)` does. Is `s` the footer or is a footer being appended to it?
Instead, have an object that has a `footer` attribute so you can just do `s.appendFooter()`.

Command-query separation

Functions should either do something or answer something, not both i.e. for `public boolean set(String attribute, String value)` it’s not so clear what it is doing. Will it return whether setting `attribute` was successfully set to `value` or whether `attribute` is currently set to `value`?

Better to change it to `public boolean attributeExists(String attribute)` to check if the attribute exists and then `public void setAttribute(String attribute, String, value)` to change the value of the attribute

Prefer Exceptions to returning error codes

Returning error codes violates command-query separation. Promotes commands being used as predicates of `if` statements i.e. `if (deletePage(page) == E_OK)`. Caller must deal with the error immediately. Throwing exceptions helps separate error handling from business logic code:

```
try {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
} catch (Exception e) {
    logger.log(e.getMessage());
}
```

Extract try/catch blocks

Extract the body inside try blocks to separate normal execution from error handling.

```
public void delete(Page page) {
    try { deletePageAndAllReferences(page);
    } catch (Exception e) {
        logError(e);
        }
    }

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

The delete function dispatches the code execution to `deletePageAndAllReferences` and error handling to `logError`. This makes it easy to modify behaviour for normal execution and error handling.

Error handling is one thing

Function doing error handling should have `try` as first statement and nothing after `catch`/`finally` blocks.

Error.java dependency magnet

If you have a class that contains all the error codes, then a lot of modules will have to import it to use the error codes. It also deters adding new error codes because everything that uses error codes will have to be recompiled. Exceptions are just derivatives of the `Exception` class, so it can just be caught by catch blocks without recompilation.

```
public enum Error {
    OK,
    INVALID,
    NO_SUCH,
    LOCKED,
    OUT_OF_RESOURCES,
    WAITING_FOR_EVENT;
}
```

Don’t Repeat Yourself

DRY, one of the most-used acronyms in the industry. Author claims it may be the root of all evil in software development, because it bloats the code and causes modifications in multiple places when an algorithm needs to change, causing errors everywhere it doesn’t change.

Structured programming

Edsger Dijkstra’s rules of programming - only one entry and one exit into functions. Only one `return` statement, no `break` or `continue` in loops and never any `goto` statements.
These are probably only useful in big functions. Keep functions small and the occasional multiple `return`, `break` or `continue` might be more expressive. But still don’t *ever* use `goto` statements.

How to write functions like this

Practise. Lots and lots of practise. First get your algorithm to work, perhaps in a huge function then abstract out common functionality between different parts into functions. Change names to make them more descriptive, shrink methods, break out classes where it makes sense to do so, reorder functions, eliminate duplication.

Comments

Comments compensate for our **failure** to sufficiently express ourselves in code. It is a failure - grimace and feel the failure when you have to use a comment and pat yourself on the back when you can express yourself in code. Comments cannot realistically be maintained. Clean coding is coding to prepare for change. Comments that are old may describe what was a previous intended behaviour but as code is refactored and functions moved around, the comment may not move with the intended functionality.

Inaccurate comments mislead and are worse than no comments. Truth can only be found in the code. Only it can tell you what it really does - comments that describe it wrongly (whether by accident or not) are bad.

Comments cannot make up for bad code. One of the most common excuses for writing comments is writing bad code. Clean up messy code rather than spend time explaining the mess you've made.

Explain yourself in code. Which is better, this:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) && (employee.age > 65))
```

Or this:

```
if (employee.isEligibleForFullBenefits())
```

Some places where comments are necessary or beneficial

- Legal comments i.e. licensing, authorship. Keep it short - refer to standard license or external document.
- Explain what a given regex should be matching
- Explain intent (i.e. trying to cause race condition in a test)
- Warning of consequences (test takes long time to run)
- TODO (idk kinda useless because you have things to keep track of tasks that need to be done now)
- Amplification of the importance of something that otherwise seems inconsequential
- Documentation - some documentation tools use specially-formatted comments to produce documentation files

Bad comments

There are lots of comments that are bad, I will give just a few examples of the most common I have encountered.

- Commented out code -> makes us question whether it was an important part of the program and whether leaving it commented out is suppressing any needed behaviour. Or was it just to try something out?
- Misleading comments -> comments that despite the best intentions of the author, are inaccurate and thus cause a user to misinterpret what a particular function actually did.
- Mumbling -> like misleading comments but are just plain ambiguous. Something like “no file found” inside an empty catch block, makes you question what the caller is supposed to do if the file isn’t found, or is there missing functionality since the catch block is empty? These comments do not lead you to understand anything about the code. They make you look around in different modules to understand what the comment means, raises more questions than provides answers.
- Mandated comments -> like requiring a javadoc for every single function even when it’s quite clear what the function does. Also a form of redundant comments, adds no documentary value. You’ll see a lot of this in your Elements of Software Construction (ESC)homework and exams.

Testing

This is arguably one of the most important parts of this lesson that I believe is not instilled in students (at SUTD at least), especially since many would wonder what the use of tests be if their school projects would no longer be used after the school term ended. But this is a huge handicap because many software dev jobs require you to know about test-driven development (TDD). Your ESC classes introduced you to testing but don't really enforce it in the project. The book also says that even though TDD is catching on, a lot of programmers miss certain subtleties when using it.

To start, there are 3 laws of TDD:

1. You may not write production code until you have written a failing unit test.
2. You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
3. You may not write more production code than is sufficient to pass the currently failing test.

So basically you cycle from writing a test, checking that it fails, writing code to make the test pass and then back to writing another test. Ensures that tests and production code are written together.

Dirty tests are worse than no tests. They cost more developer time debugging test code than actual production code. The more you edit production code without maintaining test code, the more likely you will be to drop the tests because they start to take too long to debug. This starts a vicious cycle because tests help you ensure your production code works as it should. If you drop them, you start to fear changing/maintaining production code (making it better) and thus your code just rots.

So before making any changes to production code, you should always modify the tests to test for the new behaviour, see that the tests fail, then modify the production code until it passes.

Clean tests

Clarity, simplicity, and density of expression results in readable (and thus clean) code. In a test you want to say a lot with as few expressions as possible. To get these, use the Build-Operate-Check pattern. One function should build the test data (which may be shared between different tests), one should operate on the test data and one should check that the output is correct. Again doing only one thing. Don't be afraid to refactor tests - they're like any other code.

Dual standard

There are dual standards for test and production environments. Test environment need not be memory/CPU-constrained (at least in the build and check parts).

One concept per test

One function does one thing => one test tests one thing => one concept per test. Also, minimize number of asserts in each test.

This allows reader to understand which test is for which functionality and a tester to easily find where the problem is when a test fails.

F.I.R.S.T

Clean tests follow five other rules:
FAST - They should run quickly. If they don't you won't want to run them often, leading to finding bugs later -> harder to fix. You won't feel free to clean up the code and the code rots.
INDEPENDENT - Tests should be independent of each other (no test should do the setup for another test, that's what the test data build functions are for), because this can cause errors downstream and make it hard to find where the error is.
REPEATABLE - They should run in any environment. Should be able to run the tests in the production environment. Otherwise, you'll always have a reason as to why they fail and won't be able to run the tests when the environment isn't available. Better yet, use Docker.
SELF-VALIDATNG - Tests should have a boolean output, either pass or fail. Should not have to manually inspect output or log files to verify that a test passed or failed. If they aren't self-validating, they can become subjective and running tests can require long manual inspection.
TIMELY - Written in a timely fashion. Unit tests should be written just before the production code which makes them pass. If you write it after the production code, you may find it hard to test or may not design the production code to be testable.

If you let your tests rot, your code will rot too. Keep your tests clean.

Error Handling

Error handling needed to ensure program does what it need to when things go wrong. But if it obscures logic, it's wrong.

Use Exceptions rather than return codes

Consider return codes:

```
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());
        }
    }
    …
}
```

Now compare with simply throwing an error wherever it occurs:

```
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());
        ... }
    …
}
```

You can see how the concerns between invoking a shutdown (logic) and error handling are separated.

Write the try-catch-finally statement first

Exceptions define the scope within the program. When executing in a try-catch-finally statement, code should be able to abort at any point inside the `try` block and resume in the `catch` block. The `try` block is like a transaction. The `catch` block has to leave the system in a consistent state no matter what goes wrong in the `try` block. Let's have a look at an example (also an example of how to do TDD).

Start by writing a test that tries to get a file that doesn't exist:

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

It drives us to create this stub:

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

Test fails, so now we change the implementation to try to get a file that doesn’t exist. This throws an exception:

```
public List retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName)
    } catch (Exception e) {
        throw new StorageException("retrieval error", e);
    }
    return new ArrayList();
}
```

Test passes because Exception is caught. Refactor to narrow the exception caught in the code to FileNotFoundException:

```
public List retrieveSection(String sectionName) {
    try {
        FileInputStream stream = new FileInputStream(sectionName);
        stream.close();
    } catch (FileNotFoundException e) {
        throw new StorageException("retrieval error”, e);
    }
    return new ArrayList();
}
```

Then continue using TDD to fill up the rest of the logic. Try to write tests that force Exceptions, then add behaviour to handler to satisfy the tests. This builds the transactional nature of the `try` block first and maintain it.

Use unchecked Exceptions

Catch plain old `Exceptions`. The book says the price of checked Exceptions is the violation of the Open-Closed Principle. If the checked exception (like `FileNotFoundException`) is thrown 3 levels below the `catch`, every method in between must declare the checked exception. This means that should the exception type changes, every method that throws it must change too. A small change in a low level of the code can force changes in many higher levels.

Checked exceptions can be useful in critical libraries but usually benefits outweigh costs in general development.

Provide context with Exceptions

Need to know source and location of error. Create informative error messages and pass them along with your exceptions. Mention the operation that failed and the type of failure. If logging, pass enough information to be able to log the error in the `catch` block.

Define exception classes in terms of a caller's needs

Classify errors based on what might've caused it. So instead of:

```
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 {
    …
}
```

Should instead throw a more generic error related to opening a port:

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

`LocalPort` is just a wrapper class that catches and translates the 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);
        }
    }
    …
}
```

These kinds of wrapper classes for external libraries are useful because you can switch the external library without much cost. Using only one Exception class is useful because the calling code becomes much cleaner. Only use different classes if you want to catch one and let another one through.

Define the normal flow of execution

Sometimes, instead of aborting when an Exception is caught, you want to do some other default computation. This results in code that has logic inside the error-handling `catch` block, such as the below:

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

You probably want to avoid this by using the Special Case pattern. This creates a class that handles the special case so the client code doesn't have to handle it:

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

Then `expenseReportDAO` can be changed to always return a `MealExpenses` object, which is an object of the special case class above if there are no meals found, instead of throwing an Exception. The code will then look like this:

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

Don't return nor pass null

Consider the code below:

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

Notice that every other line is a check for `null`. If you return `null`, you are inviting errors and causing more work in every other place in the code. One missing check for null will cause the entire application to crash. Notice there was no check for `null` for `persistentStore`? Either someone is catching the `NullPointerException` above, or they're not. What do you do when you get a `NullPointerException` from the depths of your application, considering it isn't very informative? You may think there's just one missing `null` check but in fact there are too many already. In a language like Python, you get what's known as a code tornado if you use too many nested `if`s (look at the shape of the code).

If you are tempted to return `null`, consider using the Special Case pattern as above, or throw an Exception. If you're calling a third-party API that returns `null`, wrap it in a class that throws an Exception or returns a special case object.

Equally worse is passing `null` into a function. Unless working with an API that is expecting a `null`, don't do it. You will either have to handle it inside the function or outside it. Either way you create WAY more work than is worth it and cause more errors overall.

Classes

Okay so we've seen a lot of low-level code, now it's time to look at a higher level view of how to organise your code so that it's clean.

Encapsulation

Encapsulate all data as `private` and only expose a few methods. A test may need to access a method so it can be `protected` instead, but always look for ways to maintain privacy - loosening encapsulation is a last resort.

Small

Just like functions, classes should be small. The size is defined by the number of responsibilities (see next part). You should be able to name the class something succinct without the words Manage, Processor or Super. These indicate that the class is doing too many things. You should also be able to describe what the class does in a sentence without the words "and", "but", "if", "or" or other such qualifiers.

Single Responsibility Principle

Classes or modules should only have one responsibility and thus only one reason to change. Thus does a few things for us:

- Limits the scope of our attention
- Organises and labels the purpose of each part of the program
- Reduces the amount of work we need to do when making changes or adding new functionality

A system with a few huge classes and does the same thing as a system with many small ones has no less complexity. You would rather a toolbox with many labeled drawers than a toolbox that you just dump everything into. Having huge classes insists that you wade through things that you don't need to know to understand how a small part of the system works.

Cohesion

Classes should have a small number of instance variables. Each method of the class should manipulate one or more of those variables. The more variables a method uses, the more cohesive it is to its class.

We want more cohesion because it means that the instance variables and the methods are co-dependent and hang together as a logical whole. Essentially by refactoring functions (**small functions and short parameter lists**), it gives us a chance to see where a new class should exist because only a subset of methods use some of the variables.

Organising for change

Follow Open-Closed Principle - classes and modules should be open for extension but closed to modification. Should be able to extend functionality by extending (subclassing) instead of modifying existing classes to get the new behaviour.

Isolating from change

follow Dependency Inversion Principle (DIP) - classes should depend on abstractions, not concrete details.
Wrap external libraries in a class, so that when you change the library, you only need to change that class to deal with the new library and keep the interface to the rest of your code consistent.

Credits

- Robert C. Martin for basically all the content here. This is just a condensation of some of the content from his book Clean Code
- Kwok Shun Git for advice about analogy about organic food vs fast food and PEP8 introduction

These notes were compiled by Tenzin Chan.