UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Module 01: Coding Standards

Advanced Programming

Maya Retno Ayu Setyautami
Editor: Teaching Team (Lecturer and Teaching Assistant)
Advanced Programming
Semester Genap 2024
Fasilkom UI

Author: *Maya Retno*
Email: mayaretno@cs.ui.ac.id

Table of Contents

Learning Objectives

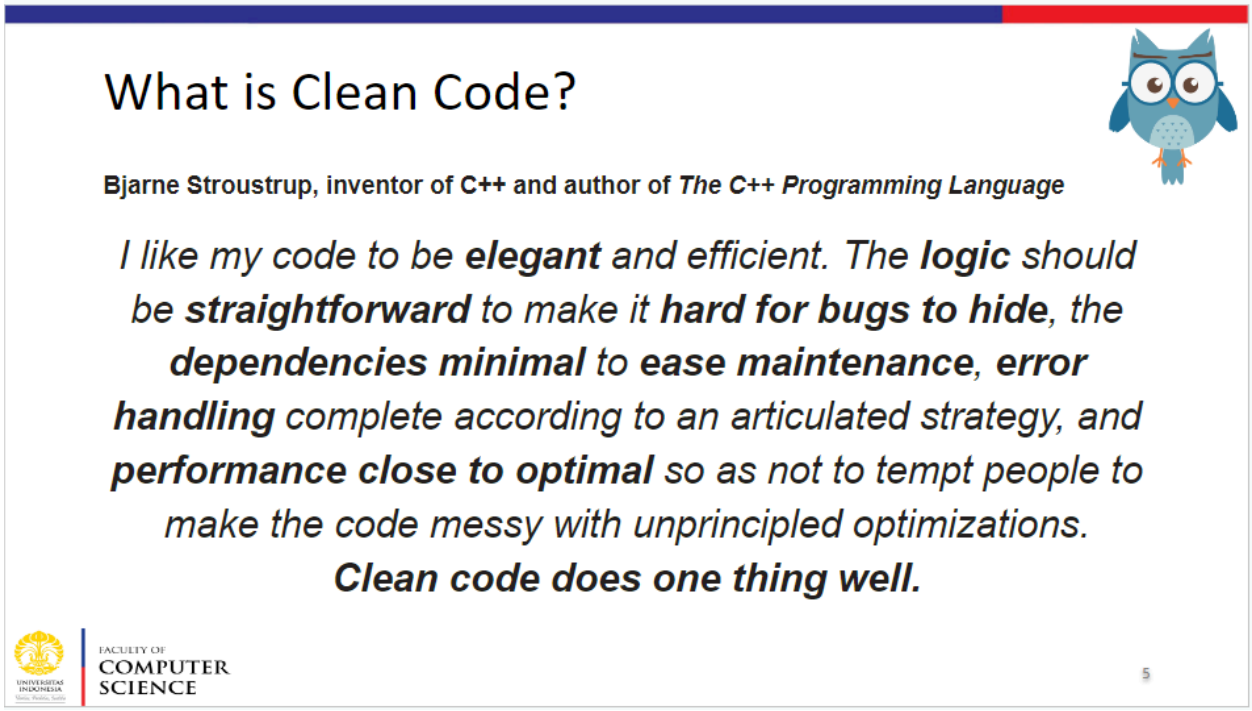
1. Students should be able to apply Clean Code principles in software development.
2. Students should be aware of security issues and able apply Secure Coding practices.
3. Students should be able to practice git flow (best practice) for teamwork development.
4. Students should be able to test the software using unit test and functional test.

References

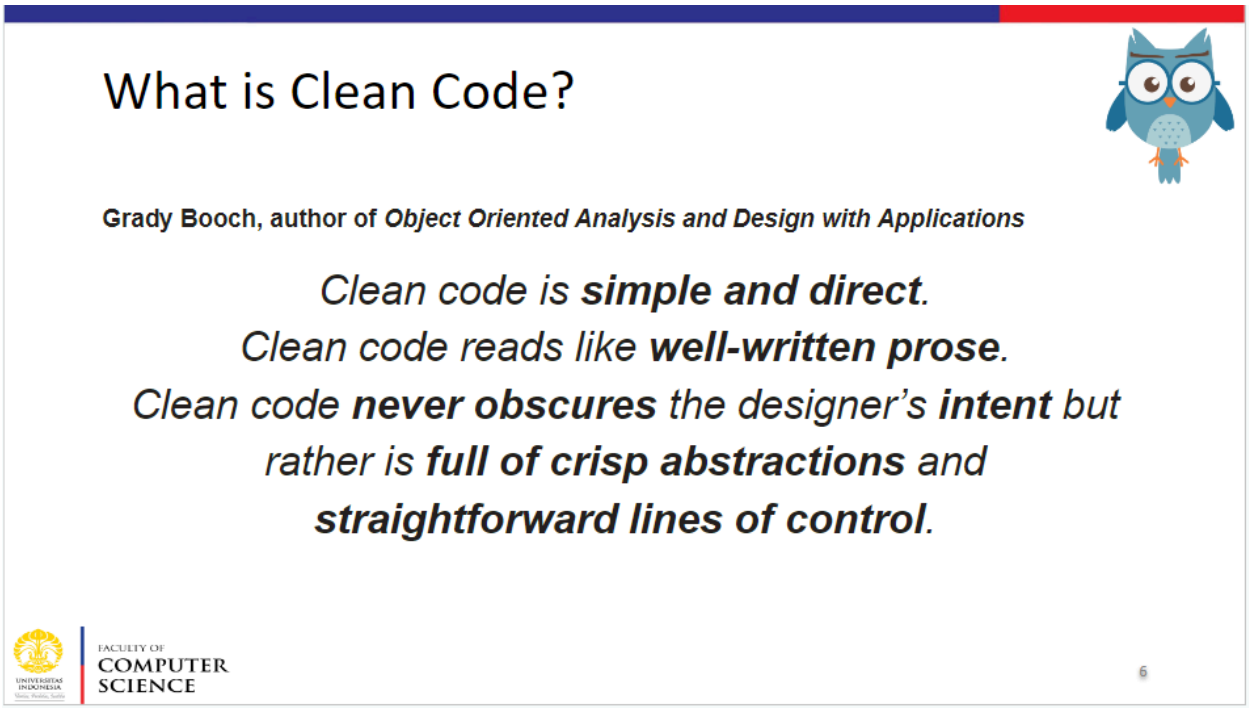
1. Martin, R. C (2007). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson.

1. Clean Code

Before applying clean code principles in the development process, let's see several perspectives about clean code from experts and practitioners.

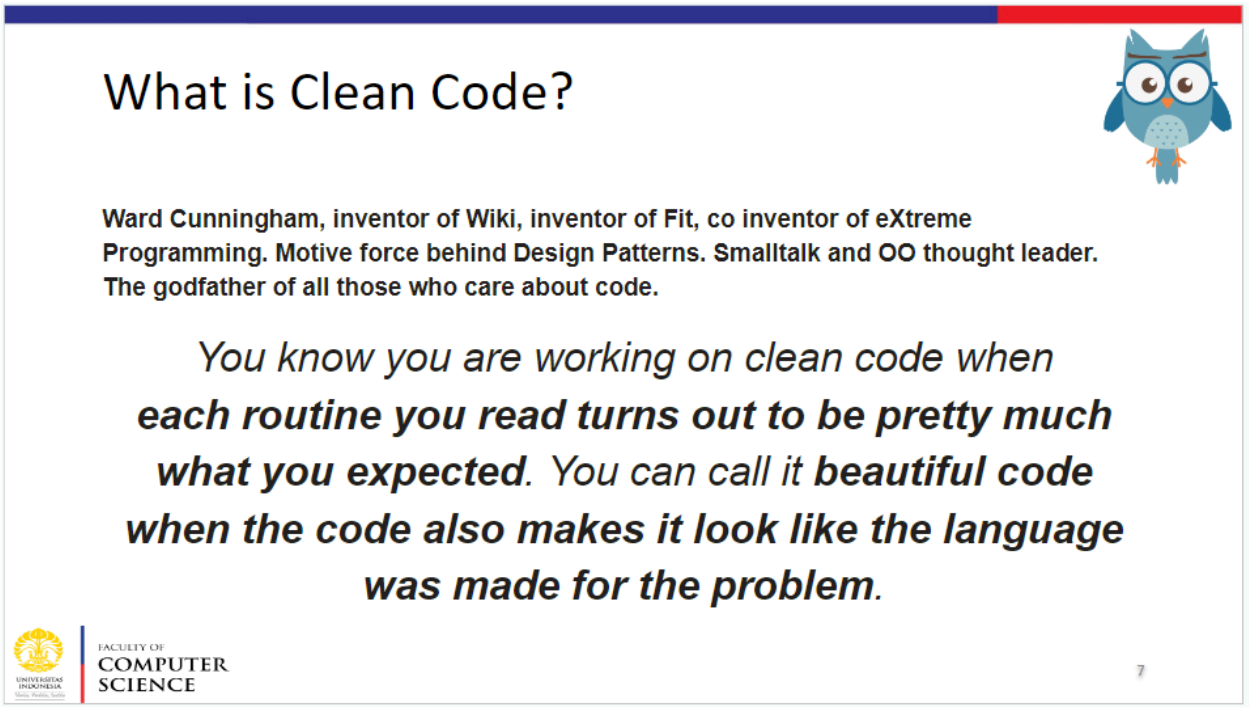


Bjarne uses the word "*elegant*". In Oxford dictionary, "elegant" (in terms of idea or plans) means clever but simple. So, the logic of code is sound and easy to follow, and the maintenance is not complicated with minimal dependencies. Clean code principles also prevent the developer from creating messy source code that is difficult to understand. One principle that is highlighted by Bjarne is the idea that a part in clean code should do exactly one thing well. Clean code is *focused*.



Grady is one of the founders of Unified Modeling Language, a standard language for modeling system's requirements. He said that clean code is simple and direct so we can read the code like we read the sentences in good prose. Yeah, reading clean code will never be quite like reading Harry Potter. Uncle Bob* said, "*Like a good novel, clean code should clearly expose the tensions in the problem to be solved*." A developer may use abstractions in the source code to represent the problem, and clean code helps the readers to understand the abstractions.

**PS: Robert C. Martin, the author of the Clean Code book, is known as Uncle Bob.*



Ward said that a clean code is a beautiful code that is easy to understand. It means that we can interpret the source code as expected, without any misunderstanding. When you read clean code, you won't be surprised at all. Programming is a part of problem solving, the source code should be easy to follow to help the reader understand the solution. So, the reader will not expend much effort. Uncle Bob said, "*It is not the language that makes programs appear simple. It is the programmer that make the language appear simple!*"

What is Clean Code?

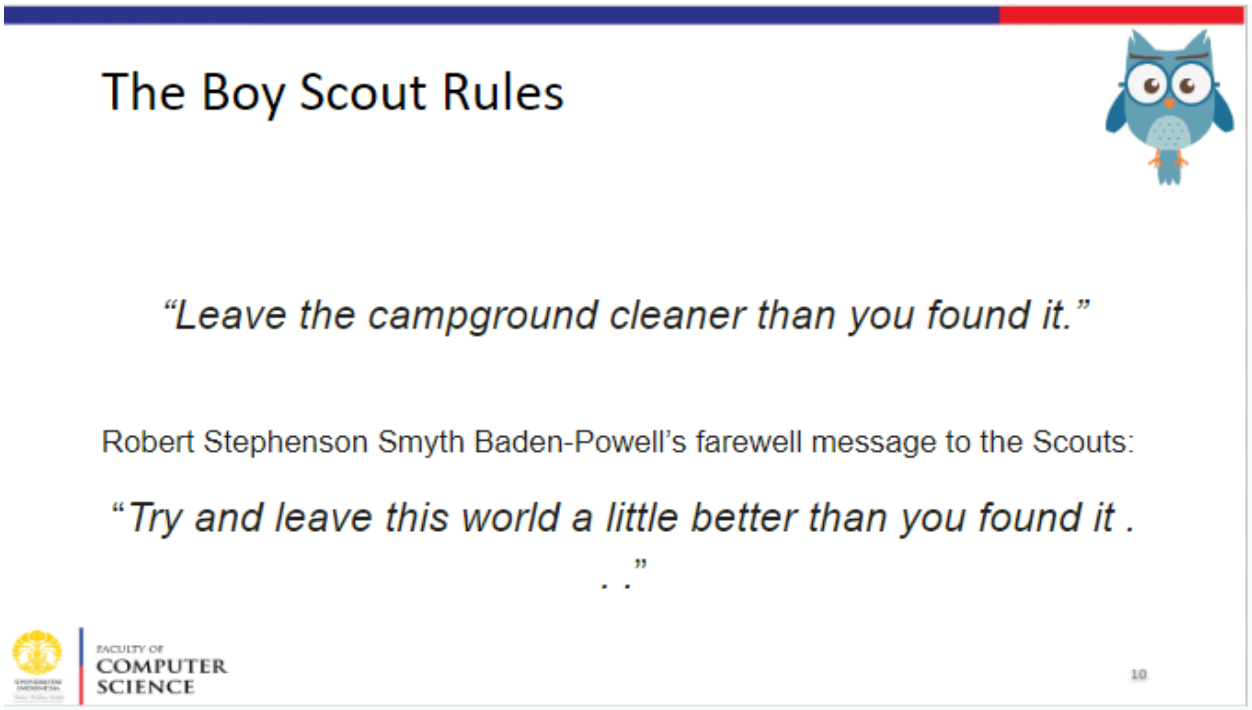
- There is no determined definition.
- In short, There are some efforts that can be maintained and pursued to conduct ***software craftsmanship.***
- Those efforts are related to several aspects in programming such as function naming, error handler, class organization, etc.

FACULTY OF COMPUTER SCIENCE

8

We have read several perspectives about clean code and there is no established definition. In short, clean code is a term used to describe computer code that is easy to read, understand, and maintain. Clean code is related to several aspects in programming, such as naming, functions, error handler, and class organizations. The written source code is not only read and executed by the machine. The source code is also read by programmers, so the clean code standard should be applied to help the programmers work effectively. Writing lines of code is not only an engineering process to build a block, but also an art to create a craft. Let's see more about Software Craftmanship.

Craftmanship is not about new stuff but is about old stuff. Software Craftsmanship is about professionalism in software development. Software Craftsmanship is an ideology, which many developers decided to adopt to get better at what they do. Programmer is not only responsible to write the code, but also think how to improve interaction, communication, and collaboration using the code. The essence of Software Craftsmanship is captured in its subtitle “Raising the Bar.”



Software Craftsmanship is not a fixed target. it is a vision whom all programmers should have. By applying the boy scout rules, we hope to make a better world of source code. The cleanup doesn't have to be something big. We can try to change one variable name to make it clearer, break up one function to separate functionalities, eliminate duplication. As said in the Boy Scout Rules, *leave the campground cleaner than you found it*. Make our code a little cleaner than we check it out first.

What should we do to clean our code?

When we clean up our room, or clean up the campground, *what do you do? What is in your mind? How you organize your cleaning?*

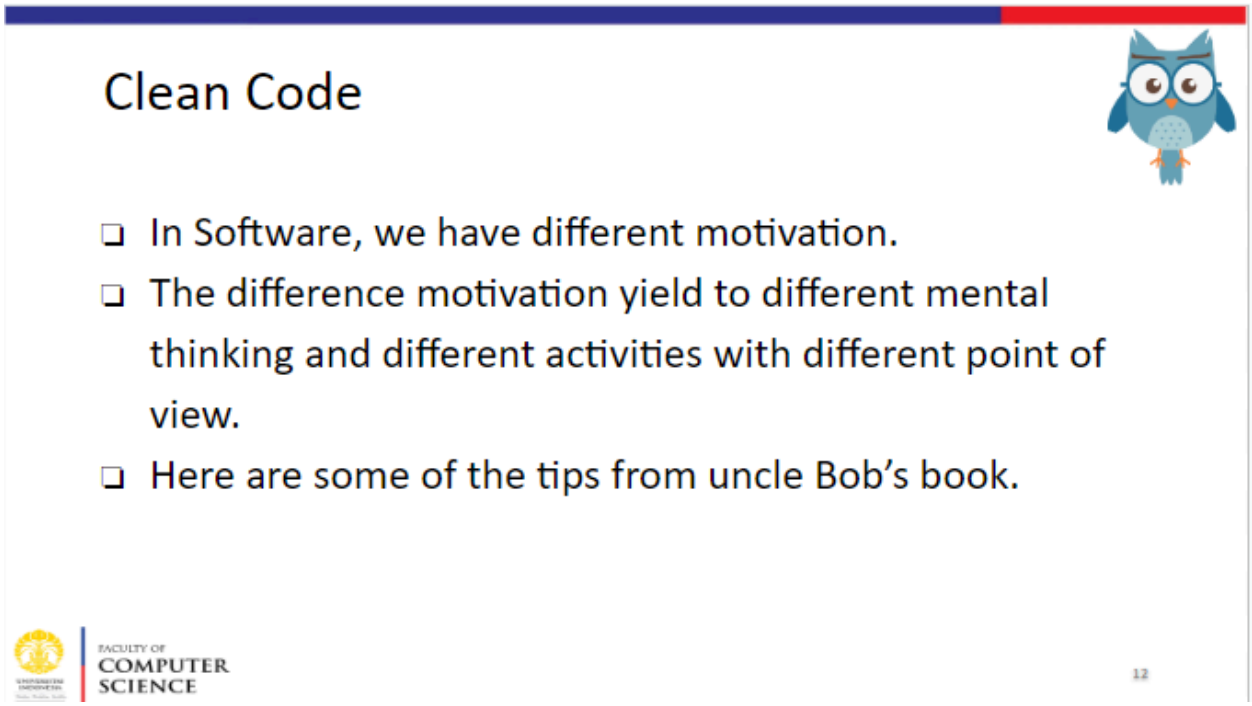
What is your **motivation**?

https://robyndykstra.com/wp-content/uploads/2017/02/Messy-Room-Cartoon.jpg

FACULTY OF COMPUTER SCIENCE

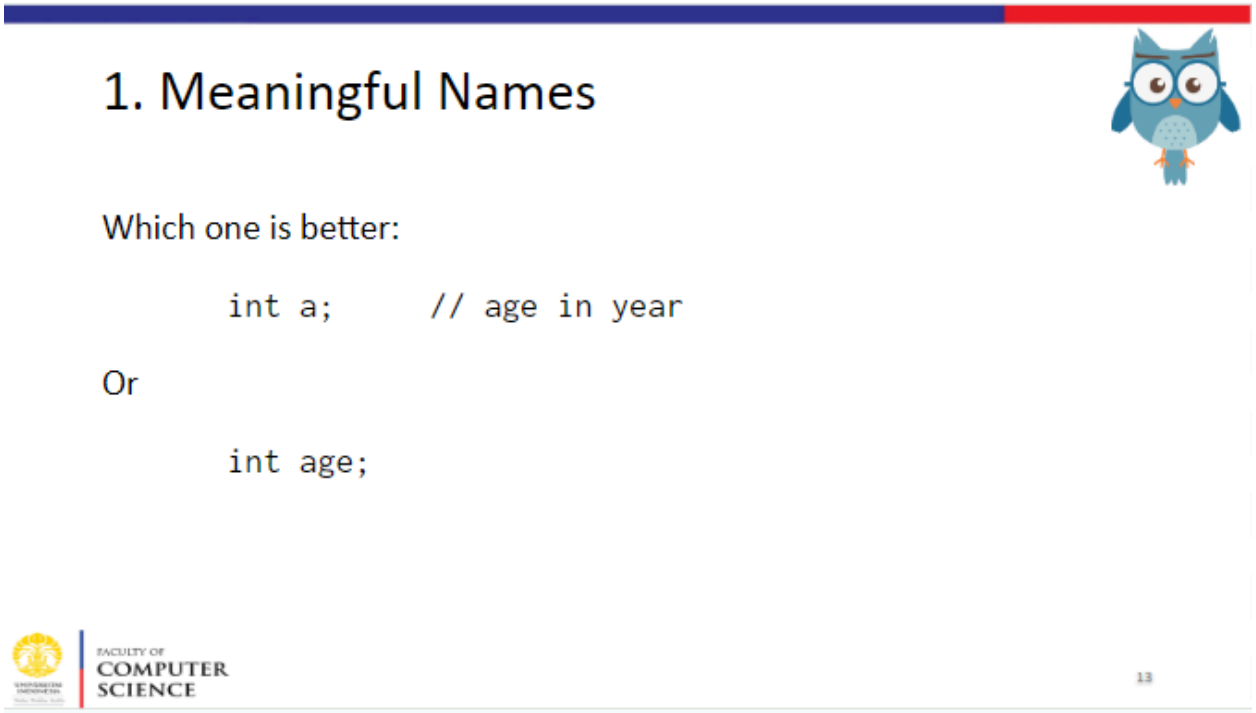
11

So, what should we do to clean our code. Everyone has different standards in cleaning up their room. Put the stuff on the right place, clean the floor twice a day, no trash in the room, etc. The most important thing is the motivation, what is your goal? So, before you apply clean code principles in your code, you have to keep in mind that raising the bar is important.



We have different motivation to do clean code, but first you must realize the importance of making clean code in software development. The difference motivation yield to different mental thinking and different activities with different point of view. We will see tips and example from Uncle Bob's book. Uncle Bob mentioned that *Books on art* don't promise to make you an artist. All they can do is give you some of the tools, techniques, and thought processes that other artists have used. Clean code principles also cannot promise to make you a good programmer. Let's see the example*, how to do clean code in software development*.

Meaningful Names



In the source code, names are everywhere. Variables, functions, classes, arguments, and many others must have name. Names play important role to guide the reader's perception and understanding. See, the first example, without having any comments, we don't know that variable a represent age in year. The name "a" reveals noting. Uncle Bob said ***if a name required comments, then the name does not reveal its intent.*** See the example above, we change "int a" to "int age". What do you think?

Exercise:

Hint: **Classic Mine Sweeper Game**

What is the purpose of the following code?

Prepare your pen/pencil and paper

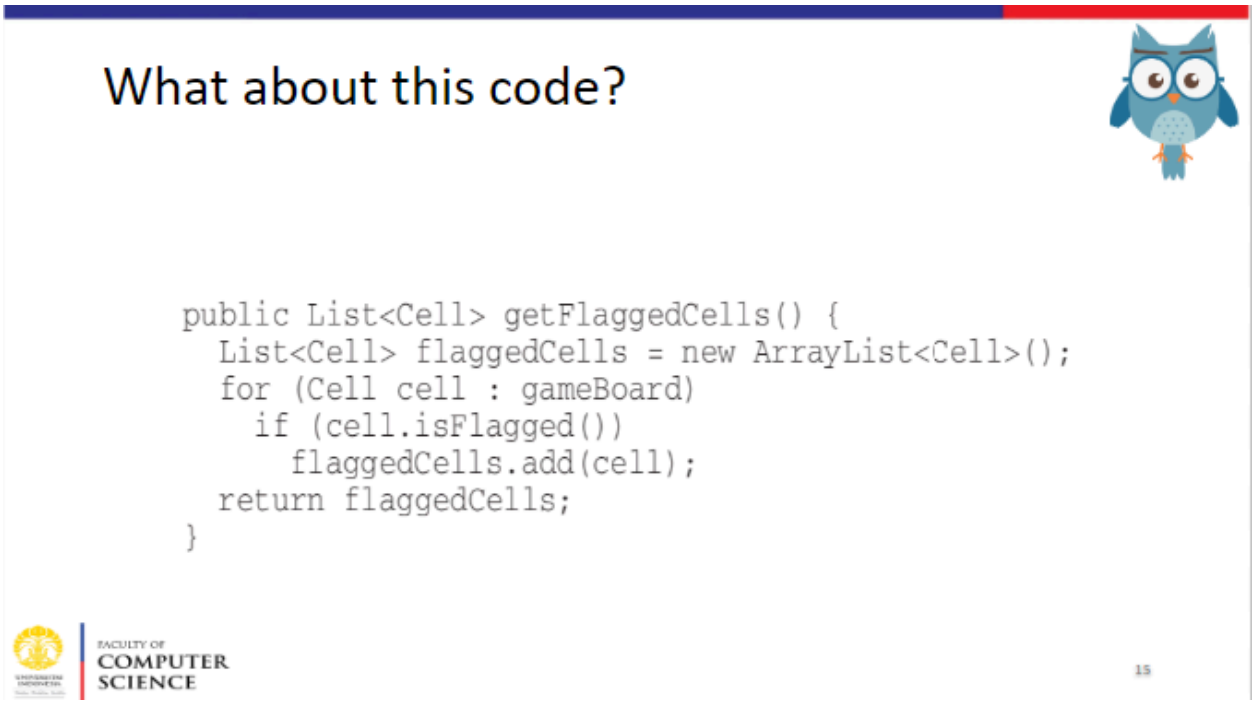
```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();
  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

FACULTY OF COMPUTER SCIENCE

14

Let's move to another example. What is the purpose of the code above? There are only three variables and two constants, no complex expressions, just a list of arrays. What is the problem? The context is not explicitly mentioned in the code. W*hat kinds of things are in* `theList`*? what is the significance of the value 4?* We could trace the program, but it is difficult to understand the context because the meaning of variables and constant is not clear.

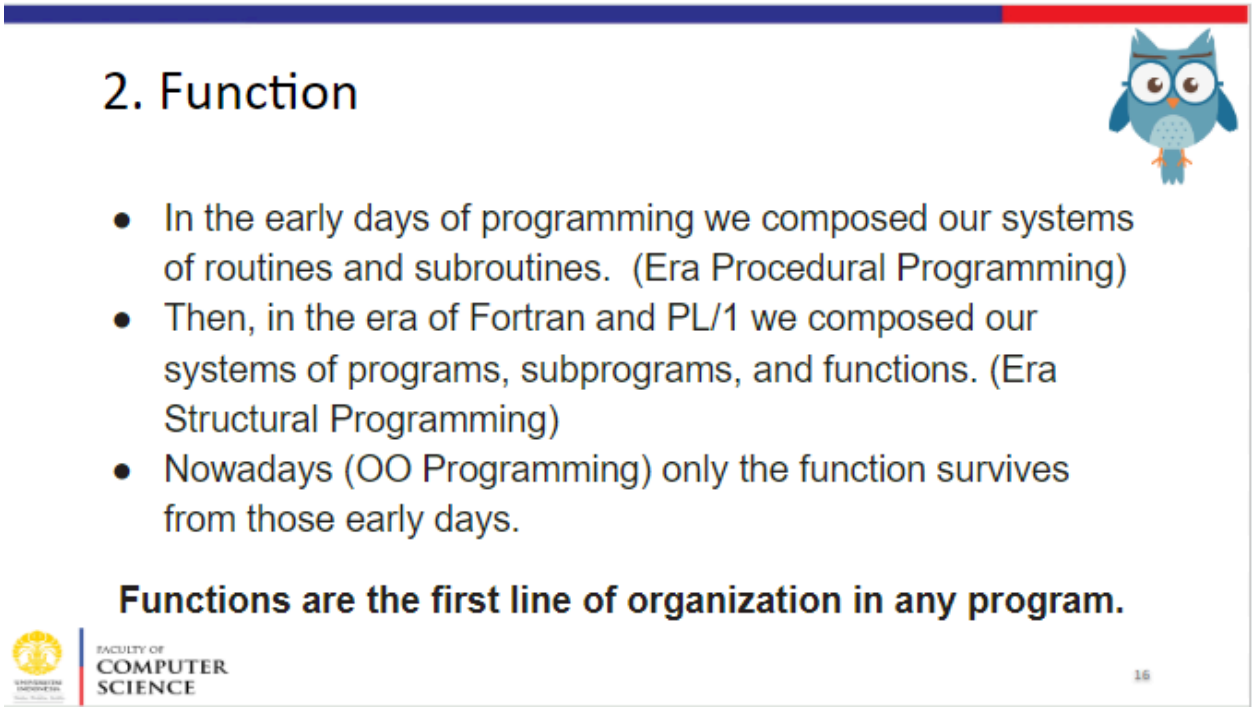
Good variable naming is important.



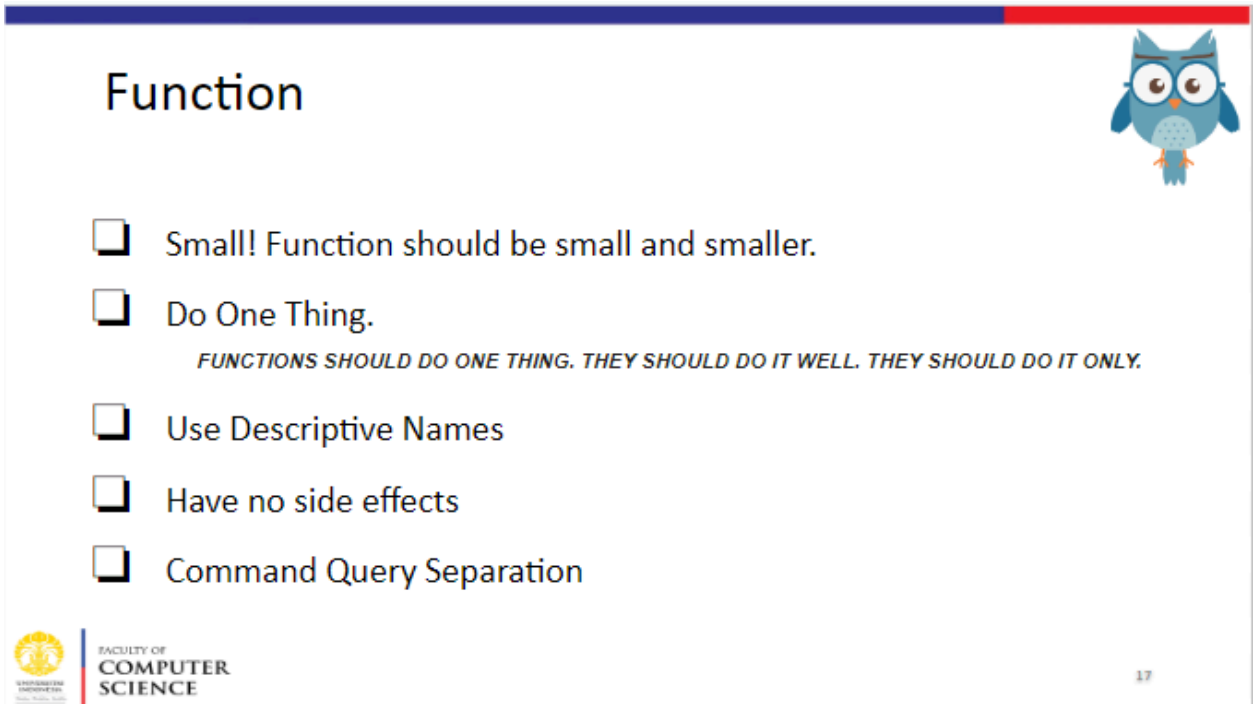
Say that we're working in a mine sweeper game. We find that the board is a list of cells called `theList`. Let's rename that to `gameboard`. By renaming `list1` to `flaggedCells` and `getThem()` into `getFlaggedCelss()`, the meaning of method is more straightforward. With these simple name changes, it's not difficult to understand what's going on. **This is the power of choosing good names.** Notice that the complexity of the code has not changed. It still has exactly the same number of variables and constants, but the code has become much more explicit.

Some tips: use pronounceable names, avoid number-series naming (such as a1, a2, .., an), use searchable names, avoid encodings, use noun or noun phrase for classes and objects, use verb or verb phrase for methods.

Function



There are a lot of programming paradigm nowadays. In the early days of programming, using procedural programming, systems are composed by routines and subroutines. Then, in era structural programming, systems are composed by programs, subprograms, and functions. In object-oriented programming, a new paradigm is proposed using classes, objects, and functions. Only the function still survives from the early days. So, functions are an important part in the program, the first line of organizations. Writing them well is one of main principle in clean code.

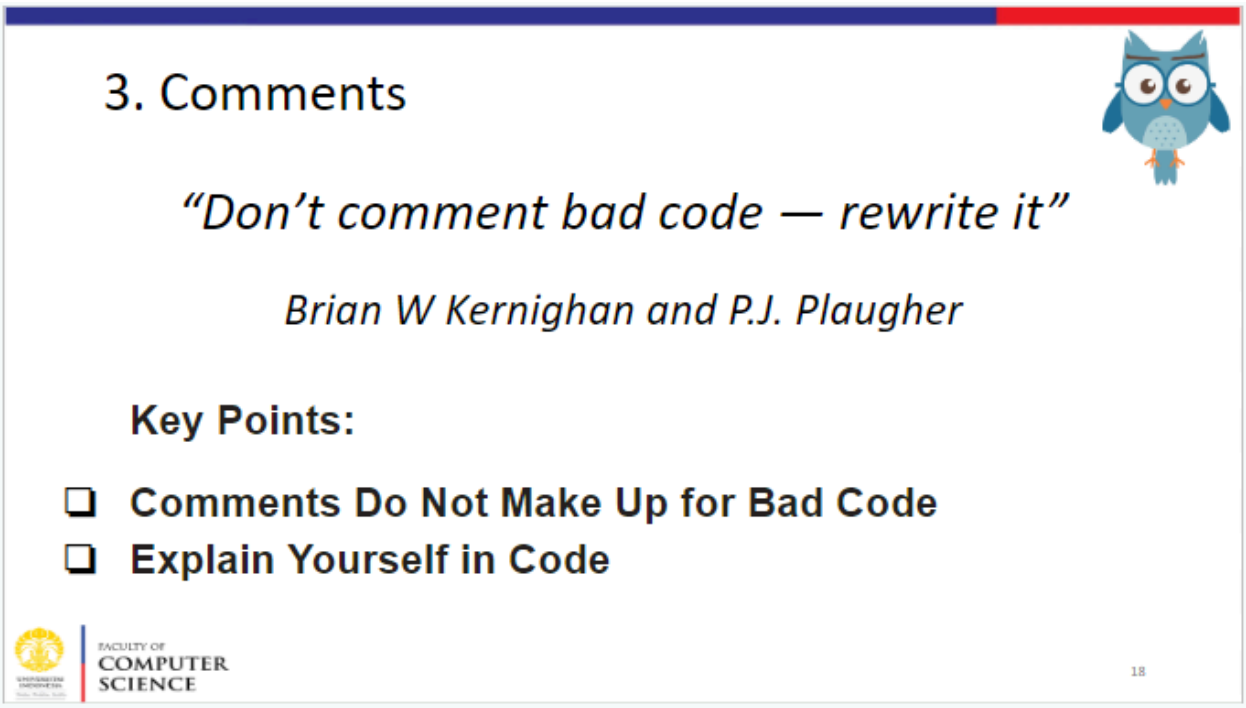


If you follow the rules in Clean Code, your functions will be short, well named, and nicely organized. Master programmers think of systems as stories to be told rather than programs to be written, and that the functions you write need to fit cleanly together into a clear and precise language to help you with that telling.

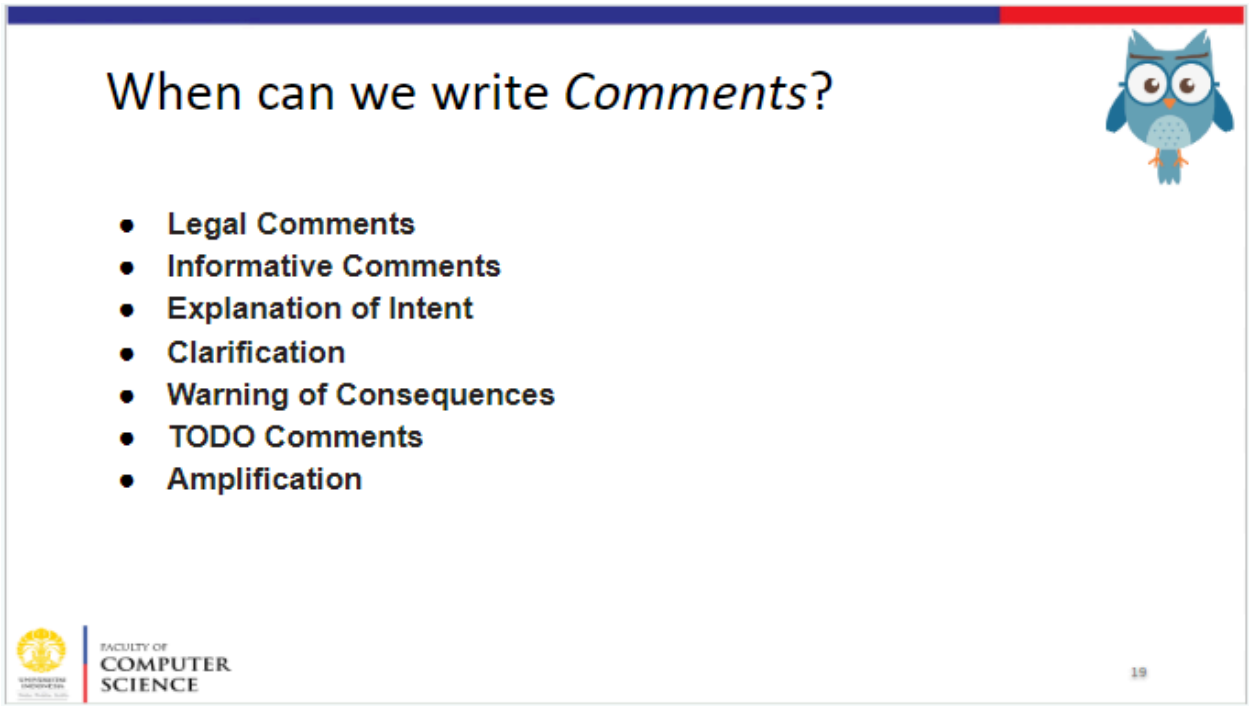
The first rule of functions should be small, and the second rule of functions should be smaller than that. *How short should a function be?* There is no standard about how many lines of code, but in eighties a function should be no bigger than a screen full. Combined with the second principle, *do one thing,* a function will be shorter. We have to make sure that functions should do one thing, do it well, and do it only. We will learn about Single responsibility principles (SRP) which is related with this criterion.

The next principle is *using descriptive names.* Uncle Bob said "*Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name*." A good function has no side effects, such as unexpected changes to the variables of its own class or does other *hidden* things. The last principle is *Command Query Separation.* Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion.

Comments



Uncle Bob said in his book that comments are always failures. We need them because we cannot always figure out how to express ourselves without them. When we need to write a comment, we have to think another possibility to express the comment in the code. One of the more common motivations for writing comments is bad code. So, don't comment bad code, rewrite it. Explain yourself in code without comments. Inaccurate comments are far worse than no comments at all.



Some comments are necessary. Here are the examples:

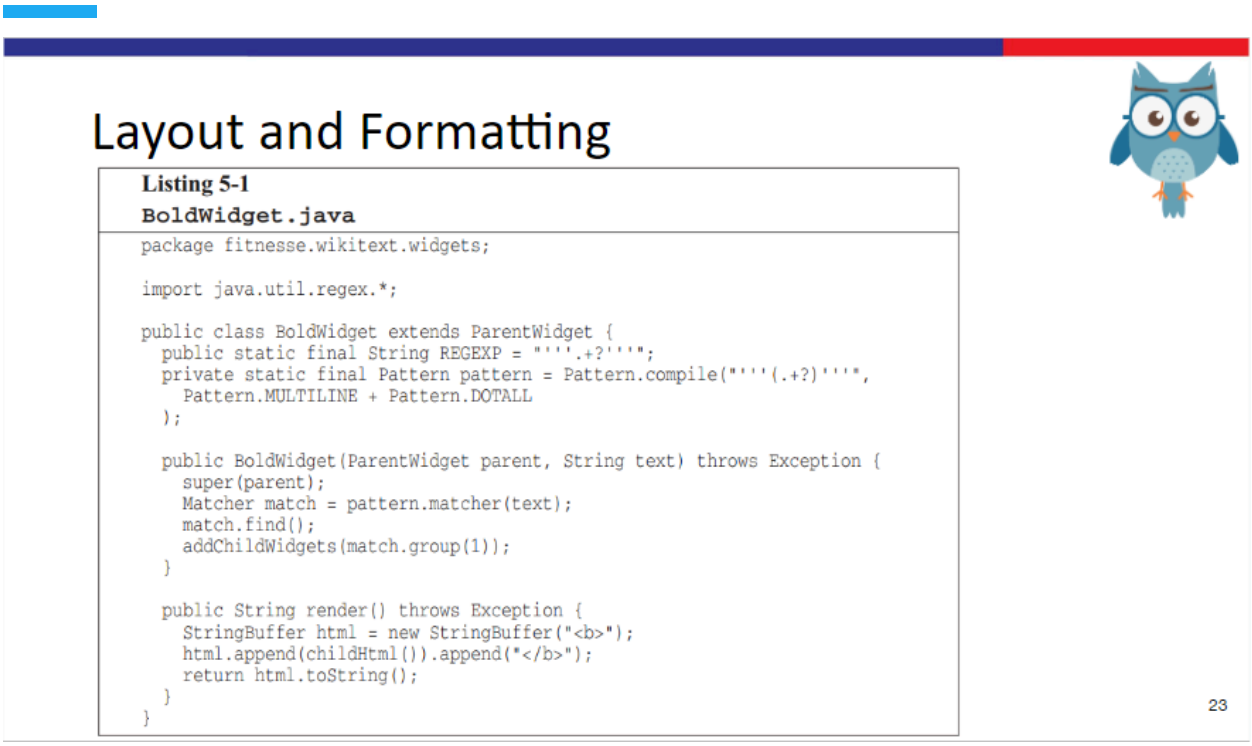
1. Legal comments, such as copyright and authorship, are required at the start of each source code.
2. Informative comments, such as date format required in the input.
3. Explanation of intents are sometimes required to describe the intent behind decision. We might not agree with the programmer's solution to the problem, but at least you know what he was trying to do.
4. Clarification is needed in the code that you cannot alter, such as a part in standard library. A helpful clarifying comment can be useful.
5. Warning of consequences is useful to warn other programmers about particular consequences, such as *don't run unless you have some time to kill.*
6. TODO comments explain why the function has a degenerate implementation and what that function's future should be.
7. Amplification can be used to strengthen the importance of something that may otherwise seem Inconsequential.

Most comments fall into bad comments. Here are the examples:

1. Mumbling. Any comment that forces you to look in another module for the meaning of that comment has failed to communicate to you and is not worth the bits it consumes.
2. Redundant comments are not more informative than the code. They do not justify the code or provide intent or rationale.
3. Misleading comments. A programmer sometimes makes a statement in his comments that is not precise enough to be accurate.
4. Mandated comments, such as Javadoc in every function or comment in every variable, are not always useful. This can clutter up the code or lend to general confusion.
5. Journal comments are usually created to document the changes log. Nowadays, we have version control systems that maintains these log entries.
6. Noise comments restate the obvious and provide no new information. We can ignore this kind of comments.

See the example at the first part. The author of this part may have written the comment first (unlikely) and then written the code to fulfill the comment. The author should then have refactored the code, as shown in the second part, so that the comment could be removed.

Have you ever seen lines of commented-out of code? We won't have the courage to delete it. We will think it is there for a reason and is too important to delete. See the example above. Why are those three lines of code commented? Are they important? Were they left as reminders for some imminent change? Or are they just cruft that someone commented-out years ago and has simply not bothered to clean up. We have a good version control systems that will remember the history of the code. We don't need to comment it out anymore. Delete unused code and we will not lose it.

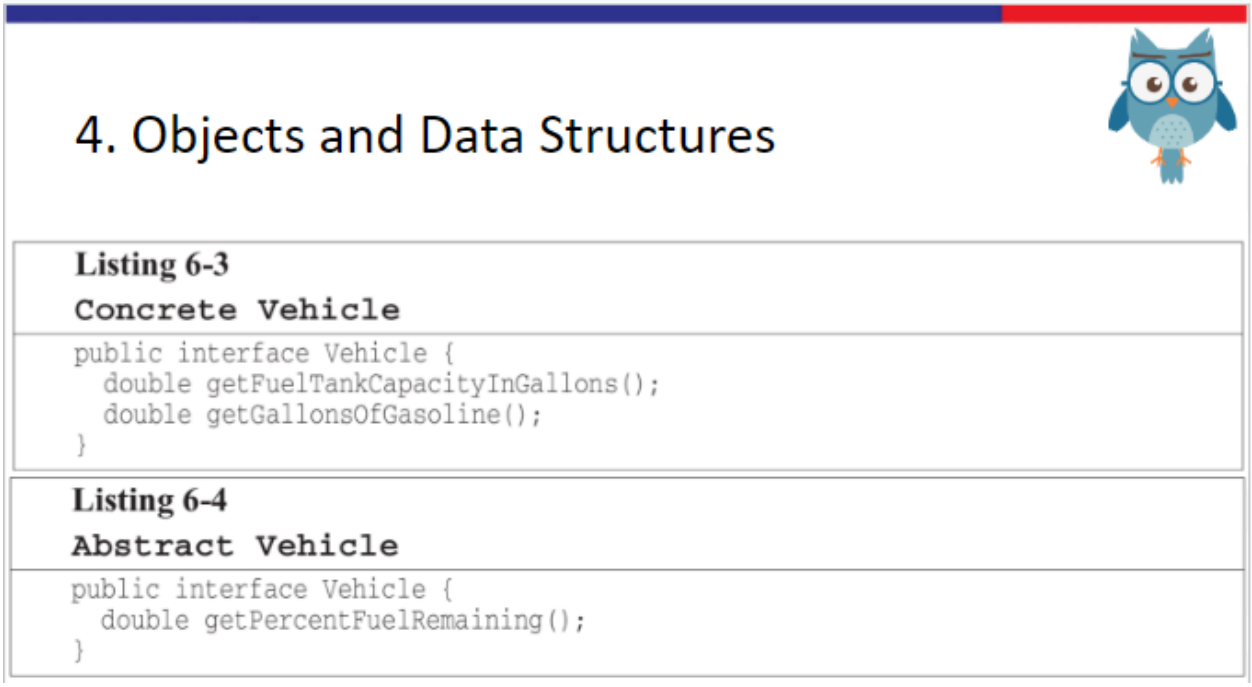


See Listing 5-1. There are blank lines that separate the package declaration, the import(s), and each of the functions. This extremely simple rule has a profound effect on the visual layout of the code. Each blank line is a visual sign that identifies a new and separate concept. As you scan down the listing, your eye is drawn to the first line that follows a blank line. Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines.

Layout and formatting is important. Indentation, variable declaration, vertical ordering are examples of code styles and formatting that are too important to ignore. Code formatting is about communication, and communication is the professional developer's first order of business. A team of developers should agree upon a single formatting style, and then every member of that team should use that style.

Tips: Use automatic styling or formatting checker (linter). You can use Checkstyle and/or OpenRewrite to ensure code format consistency in a Java-based project.

Objects and Data Structure

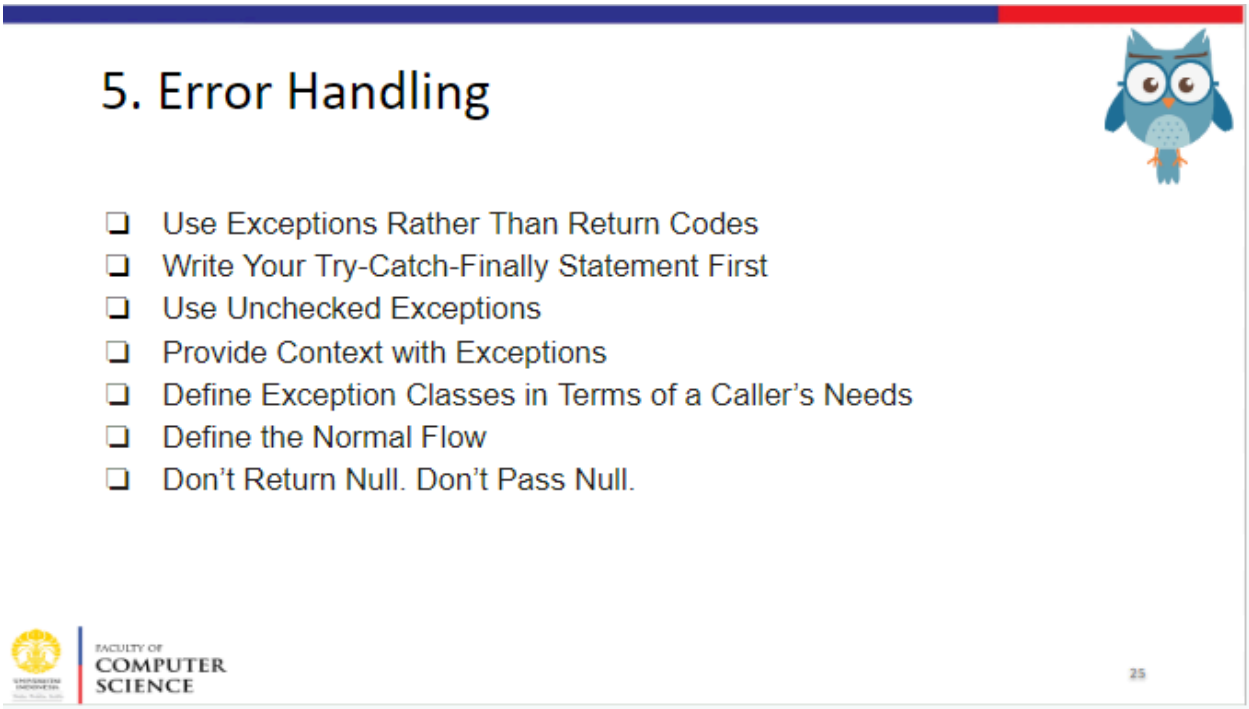


We keep our variables private because we don't want anyone else to depend on them. We want to keep the independence to change their type or implementation. Getters and setters are often used to expose private variables as if they were public. Uncle Bob said that hiding implementation also deals with abstractions. Do not simply push class's variables out through getters and setters. We can expose abstract interfaces that allow to manipulate the essence of the data.

Objects expose behavior and hide data. We can add new kinds of objects without changing existing behaviors. Data structures expose data and have no significant behavior. We can add new behaviors to existing data structures but makes it hard to add new data structures to existing functions.

What is the difference between Listing 6-3 and Listing 6-4? The first uses concrete terms to communicate the fuel level of a vehicle, whereas the second does so with the abstraction of percentage. The first case just consists of accessors of variables. In the second case, we have no clue at all about the form of the data. Which one do you prefer? The second option is preferable because the details of data do not need to be exposed. We can express our data in abstract terms. This is not merely accomplished by using interfaces and/or getters and setters.

Error Handling



Clean code is readable, but it must also be robust. We can write robust clean code if we see error handling as a separate concern, something that is viewable independently of our main logic. Our program can go wrong, and programmers are responsible to handle the possible error. Be careful, a lot of error handling might be confusing. Uncle Bob puts several considerations to handle error effectively, as shown in the slide above.

Listing 7-1

DeviceController.java

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

26

See Listing 7-1, what's wrong with these approaches? Previously, many languages did not have exceptions. An error code or error flag is returned to the method's caller. The caller should check for errors and sometimes it is not easy to remember about checking the errors. Throws an exception in methods that can detect error is better. See the next slide for the improvement.

Listing 7-2
DeviceController.java (with exceptions)

```
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());
    ...
  }

  ...
}
```

27

Notice how much cleaner it is. The improvement does not only deal with error handling, but also the function. The code is better because two concerns are separated, the algorithm for device shutdown and error handling. We can look at each of those concerns and understand them independently.

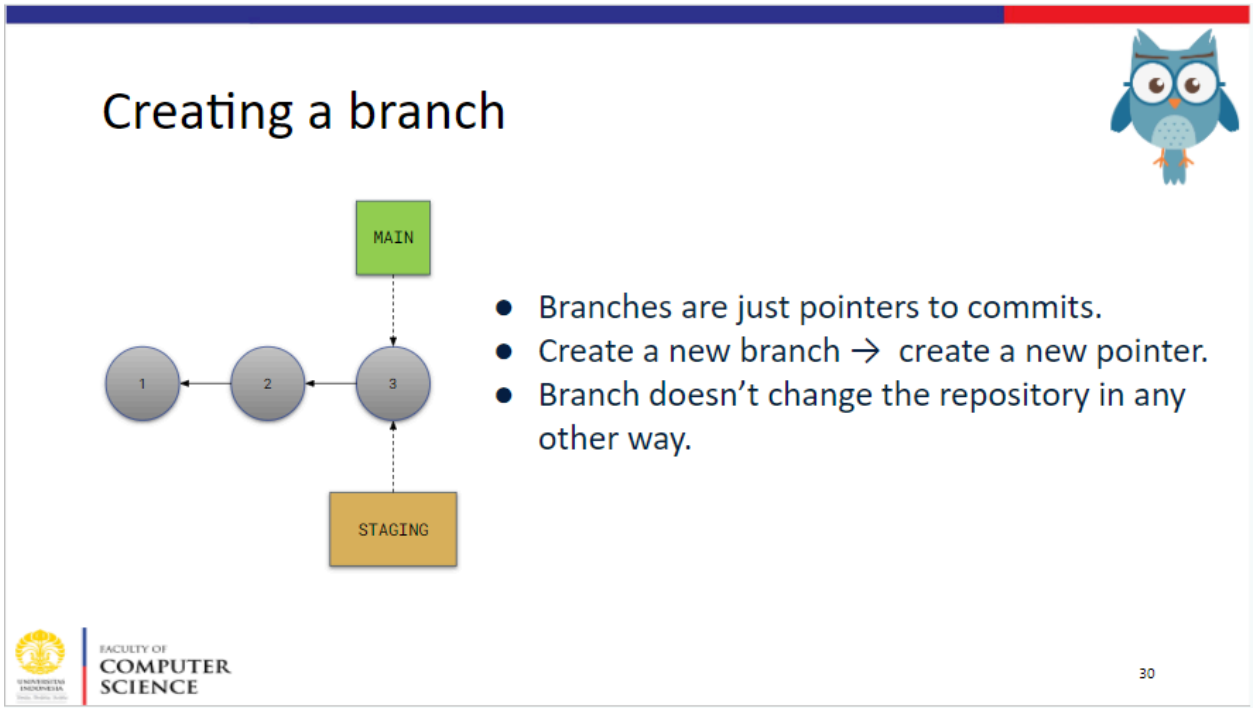
2. Git Flow

In another course, you have used `Git` to support the development process. If you forget about basic git commands, please read it again. In this module, we will learn how to use git for teamwork development.

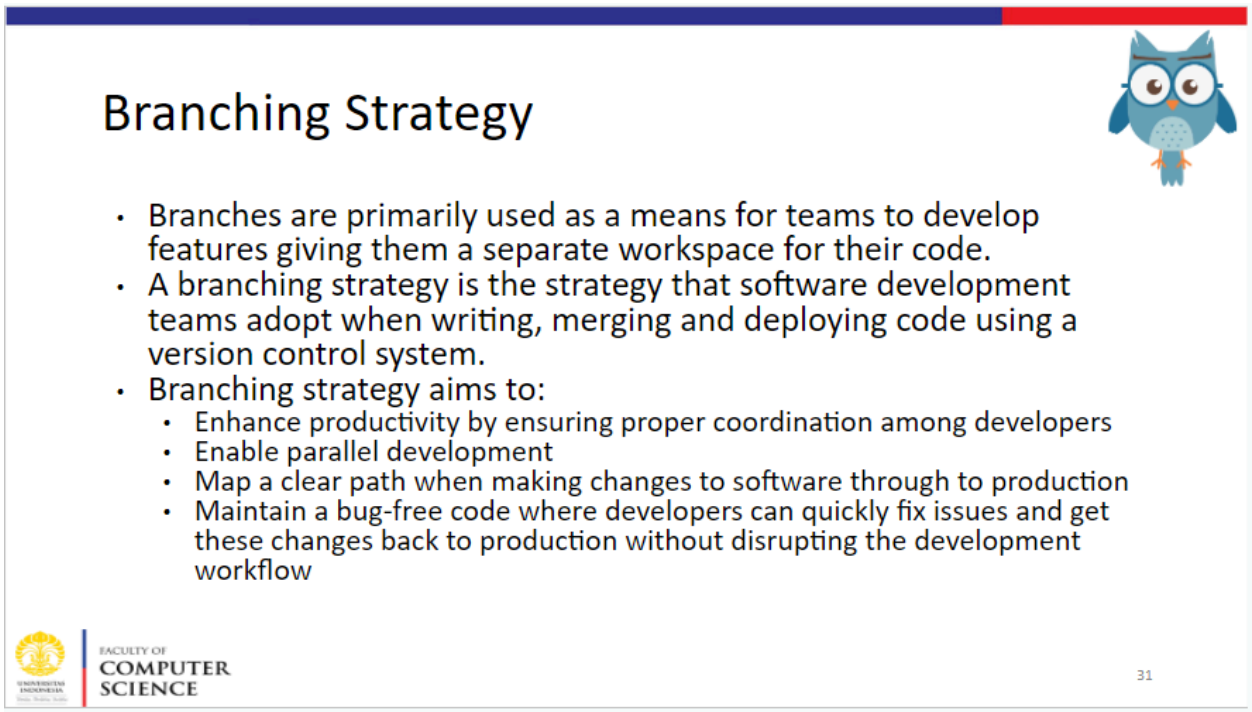
We start with branching mechanism in `Git`. A branch represents an independent line of development. Branches serve as an abstraction for the edit/stage/commit process. New commits are recorded in the history for the current branch, which results in a fork in the history of the project.

The `git branch` command lets you create, list, rename, and delete branches.

For example: Supposed that we have **main** branch. We can create a new branch **staging** by executing `git branch staging`.



Branches are just pointers to commits. When we create a branch, Git is creating a new pointer. So, a branch does not change the repository in any other way. As shown in the Figure above, two branches (main and staging) are pointing into the same series of commits.

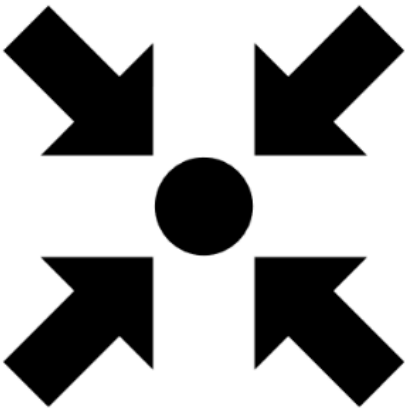


Branches are primarily used as a means for teams to develop features giving them a separate workspace for their code. These branches are usually merged back to a main branch upon completion of work. In this way, features (and any bug and bug fixes) are kept apart from each other.

A branching strategy is the strategy that software development teams adopt when writing, merging and deploying code. Having a branching strategy is necessary to avoid conflicts when merging and to allow for the easier integration of changes into the main branch. The objectives of branching strategy are mentioned in the above slide.

Workflow?

- A codebase can be considered as the “critical section” in a software construction process that concurrently modified by developers
  - Thus, requires **a set of procedures, or recipes,** to manage the changes into the codebase in a predictable manner

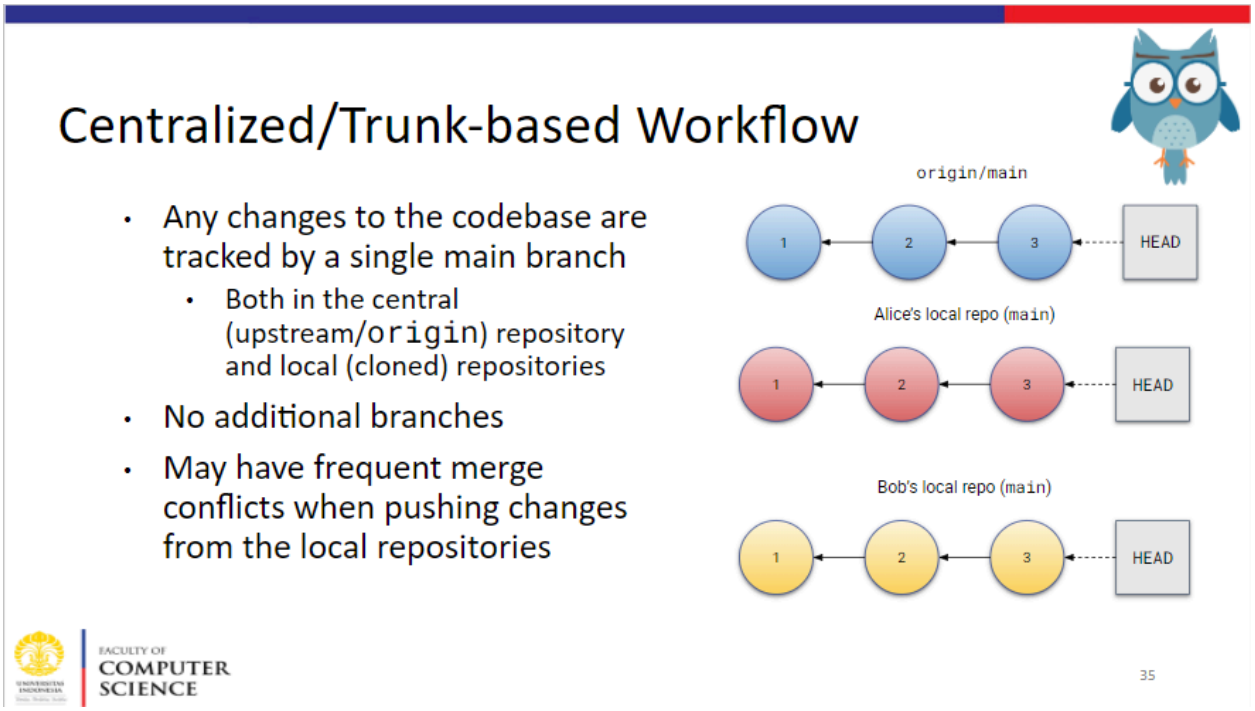


FACULTY OF COMPUTER SCIENCE

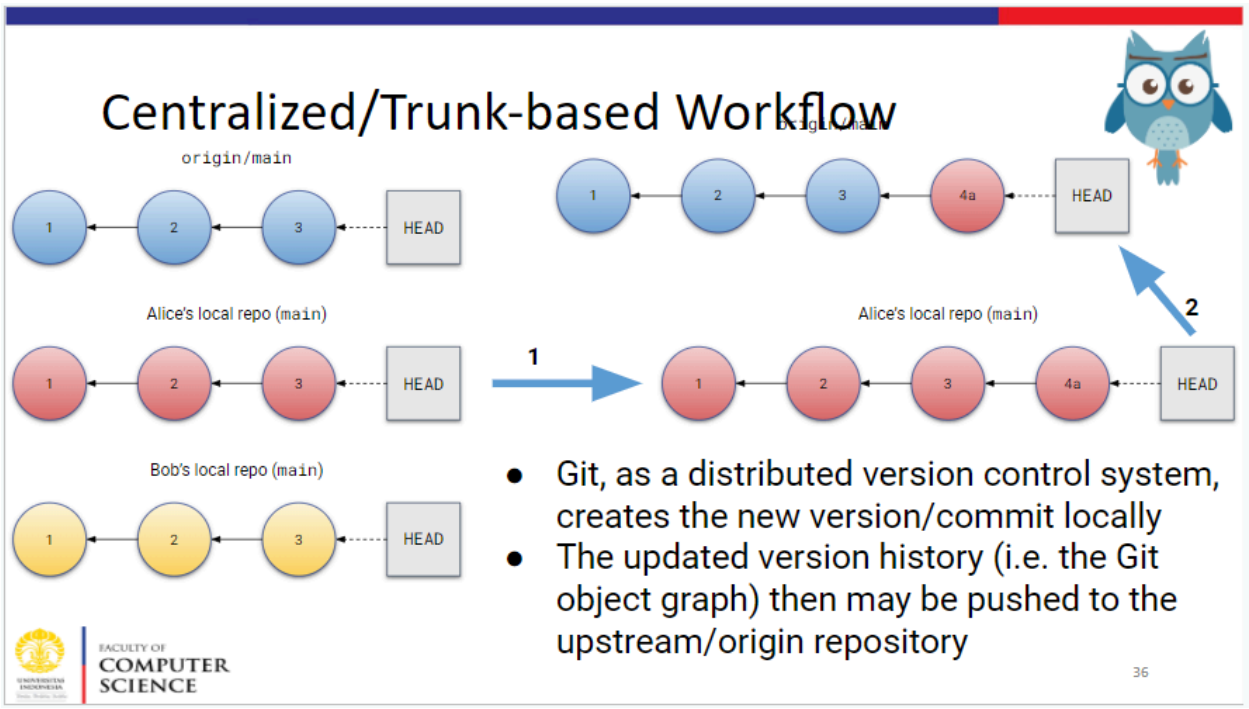
32

A codebase can be considered as the critical part in the development process. Any conflict might occur because the codebase can be concurrently modified by developers. Git workflow is a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner. Therefore, the changes in the codebase are managed systematically. We will learn several kinds of Git workflow in this module. These workflows are designed to be guidelines rather than rules. A workflow can be mixed and matched with other workflows to suit your individual needs.

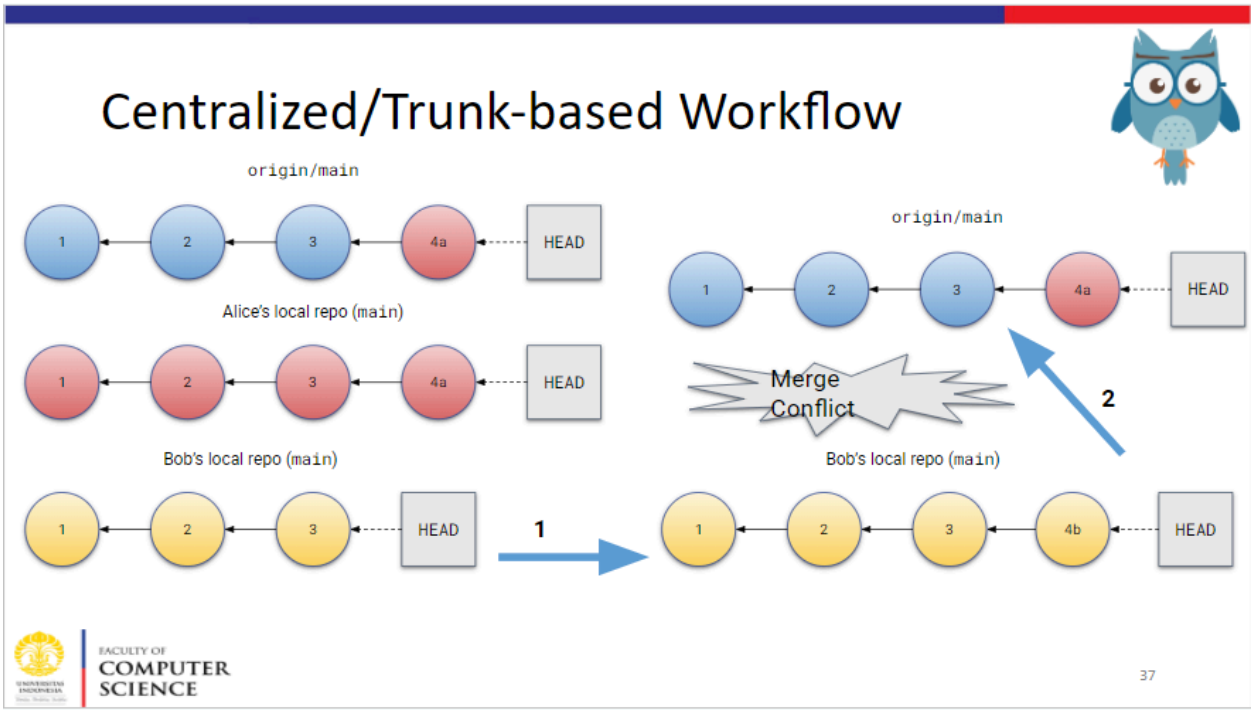
Centralized / Trunk-based Workflow



The Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. The default development branch is called **main** and all changes are committed into this branch. This workflow does not require any other branches besides the **main**. Therefore, this workflow may have frequent merge conflicts when pushing changes from the local repositories. As shown in the Figure above, supposed that Alice and Bob work on separate features. Using centralized workflow, they share their contribution via centralized repository without creating any new branches.

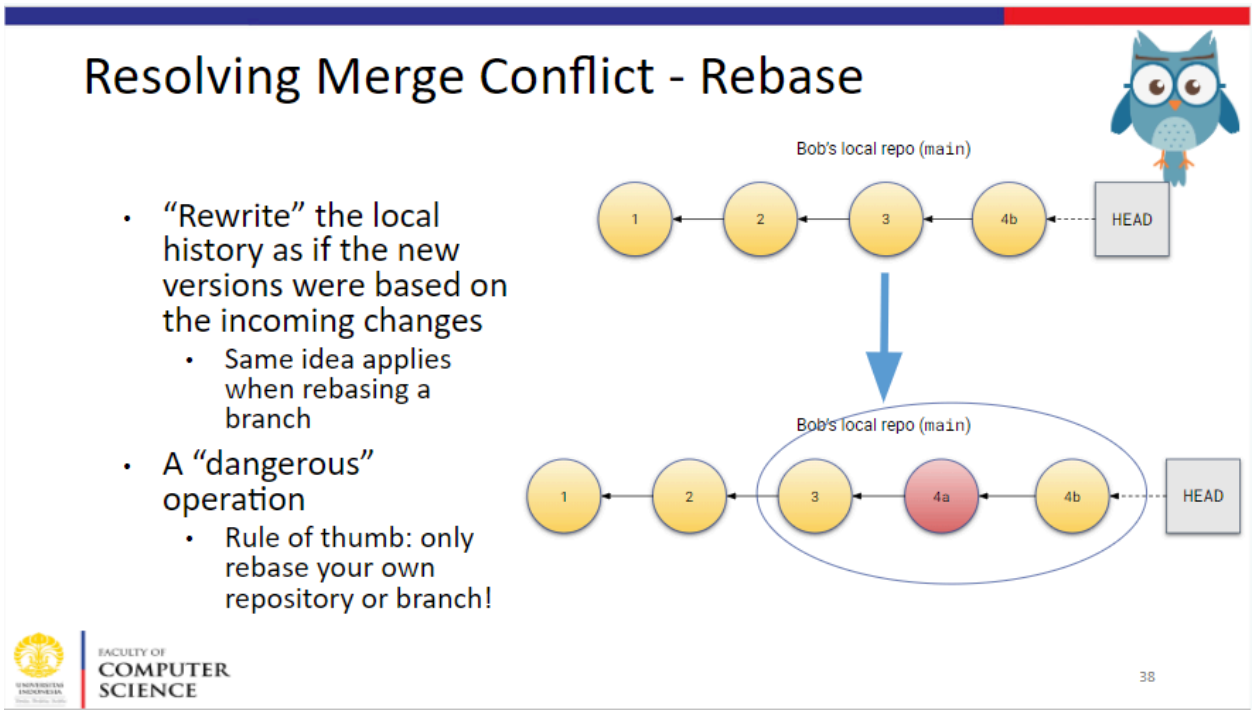


Alice works on her feature and Bob works on his feature. Alice and Bob can develop features using standard git commands. Remember that since these commands create local commits, Alice can repeat this process as many times as she wants without worrying about what's going on in the central repository. Once Alice finishes her feature, she pushes local commits to the central repository (4a). Since the central repository has not been updated since Alice cloned it, this will not cause any conflicts and the push will work as expected.

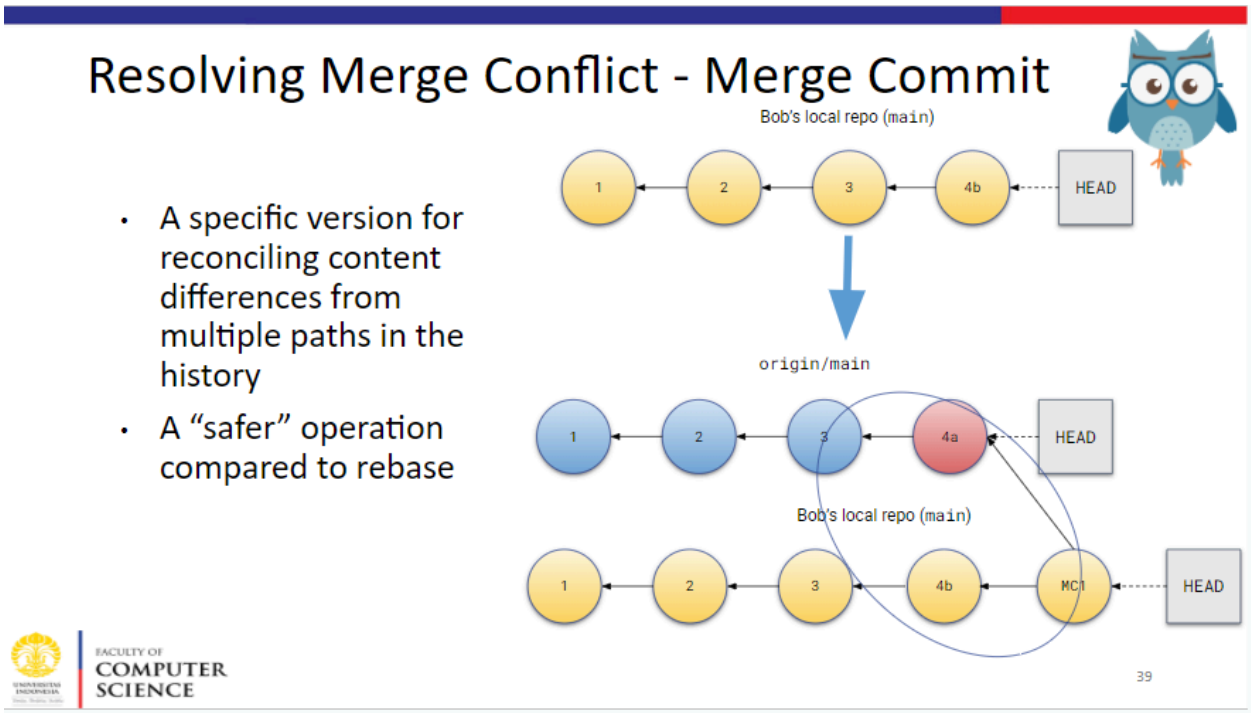


Meanwhile, Bob is working on his own feature in his own local repository. Like Alice, he doesn't care what's going on in the central repository, and he really does not know what Alice is doing in her local repository, since all local repositories are private. What happens if Bob tries to push his feature (4b) after Alice has successfully published her changes to the central repository? Since Bob's local history has diverged from the central repository, Git will refuse the request. This error prevents Bob from overwriting official commits. He needs to pull Alice's updates into his repository, integrate them with her local changes, and then try again.

We will see how to solve the conflict.

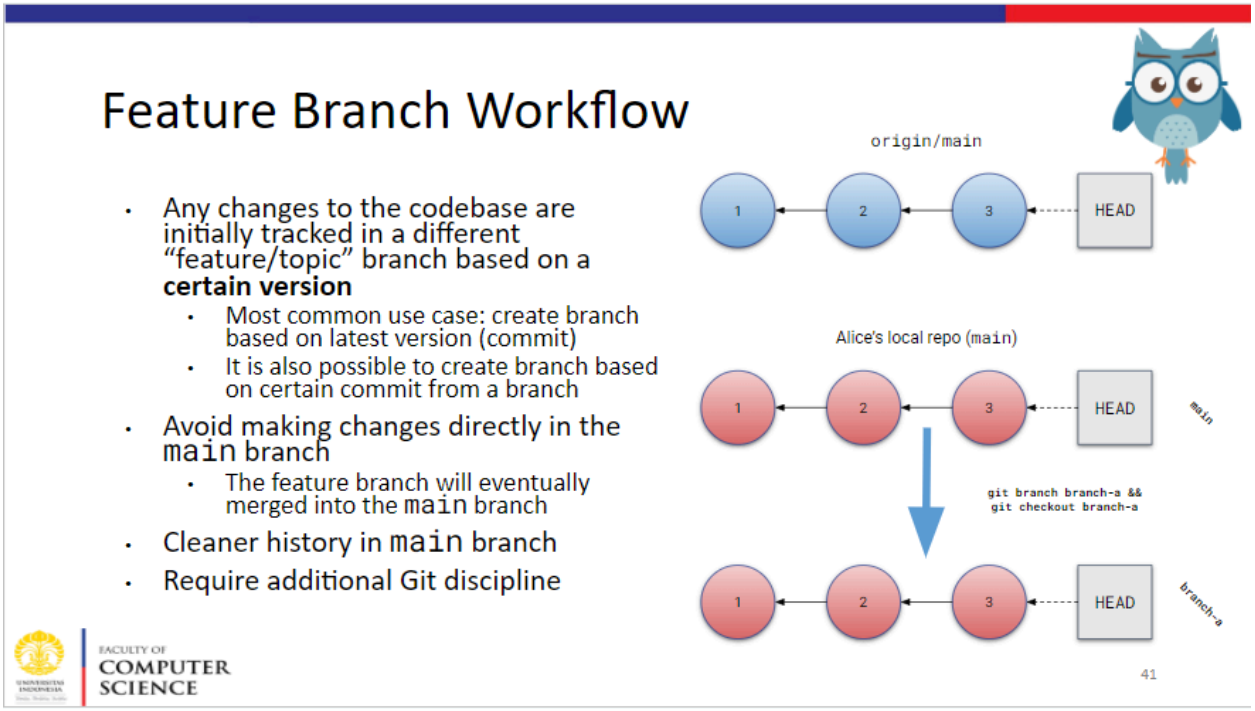


The first approach is Bob rebases on top of Alice's commits. Bob can use git pull to incorporate upstream changes into his repository. This command pulls the entire upstream commit history into Bob's local repository and tries to integrate it with his local commits. We can add `--rebase` option to the pull command (`git pull --rebase origin main`). The `--rebase` option tells Git to move all of Bob's commits to the tip of the main branch after synchronizing it with the changes from the central repository. As shown in the Figure, commit 4a is added to Bob's local repo before commit 4b.

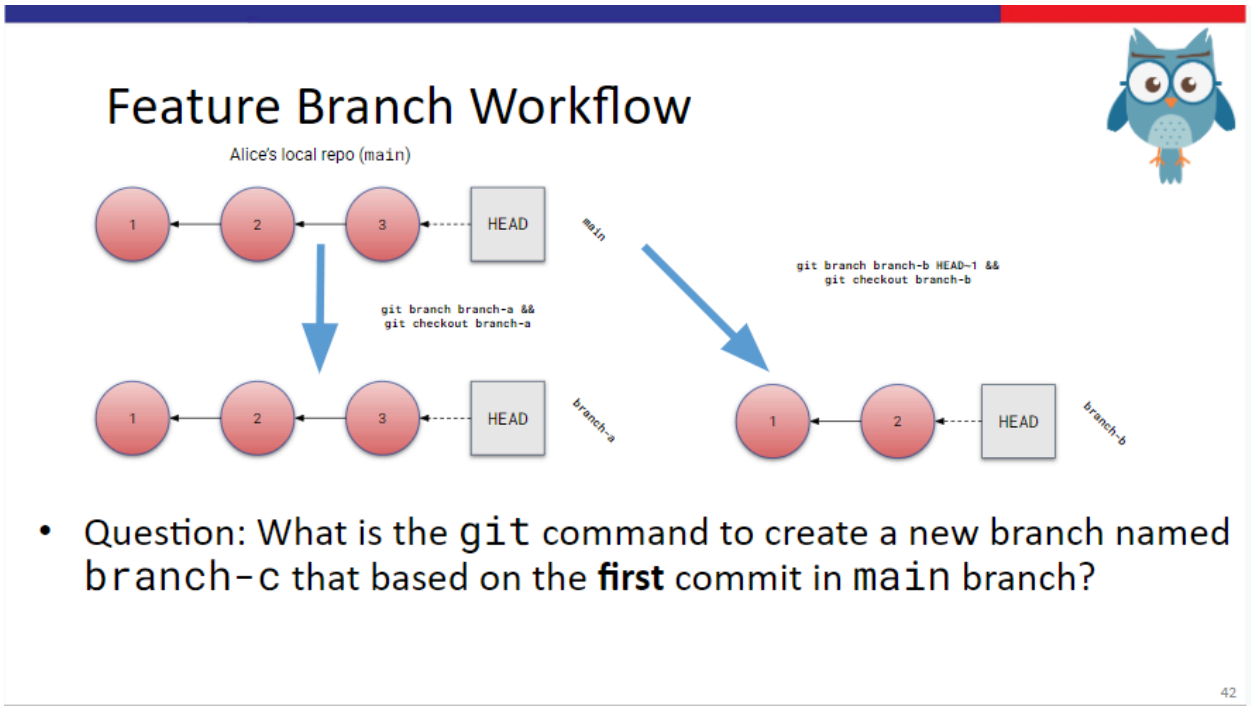


We can resolve the merge conflict using merge commit. When creating a merge commit Git will attempt to merge the separate histories auto magically for you. Supposed that Bob wants to merge his work 4b to the origin/main (that already contains Alice's work 4a). Git merging combines sequences of commits into one unified history of commits MC1. After Bob's done synchronizing with the central repository, he will be able to publish his changes successfully.

Feature Branch Workflow

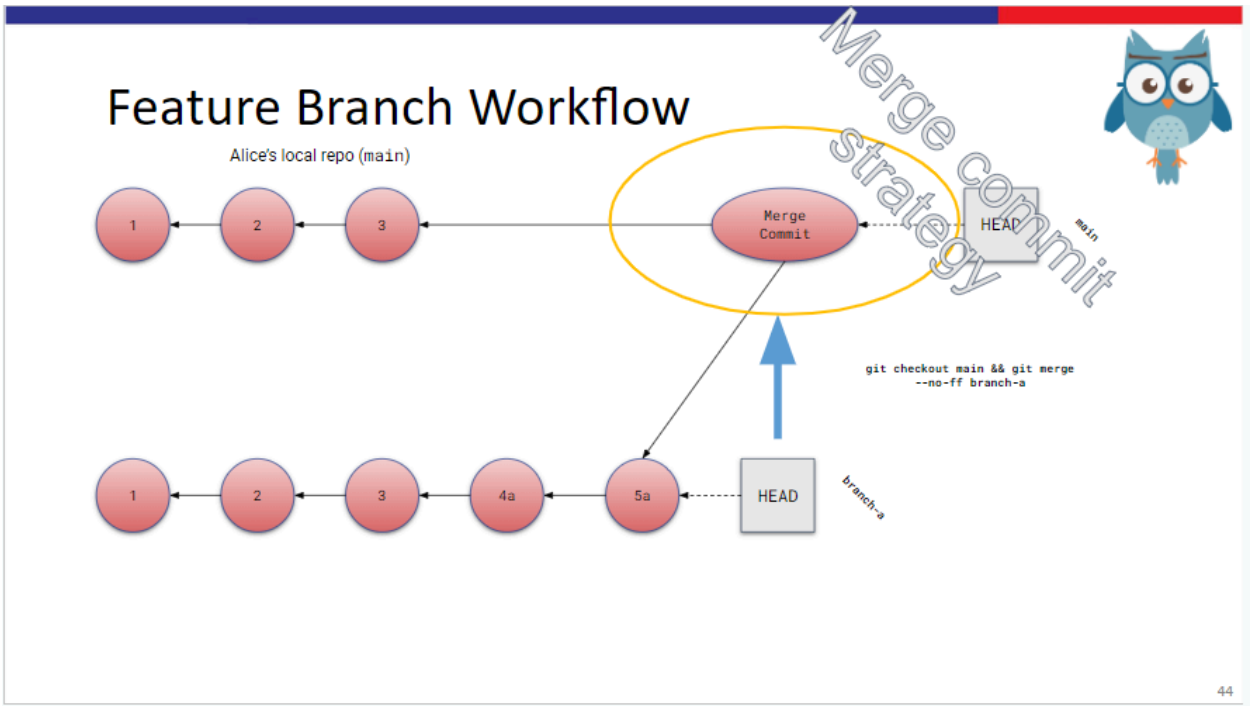


In Feature Branch Workflow, all feature development should take place in a dedicated branch instead of the main branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. Any changes to the codebase are tracked in a different feature branch. It also means the main branch will never contain broken code, which is a huge advantage for continuous integration environments. For example, before Alice starts developing a feature, she creates a new branch using command: `git branch branch-a` and then `git checkout branch-a`. This checks out a branch called `branch-a` based on `main`.

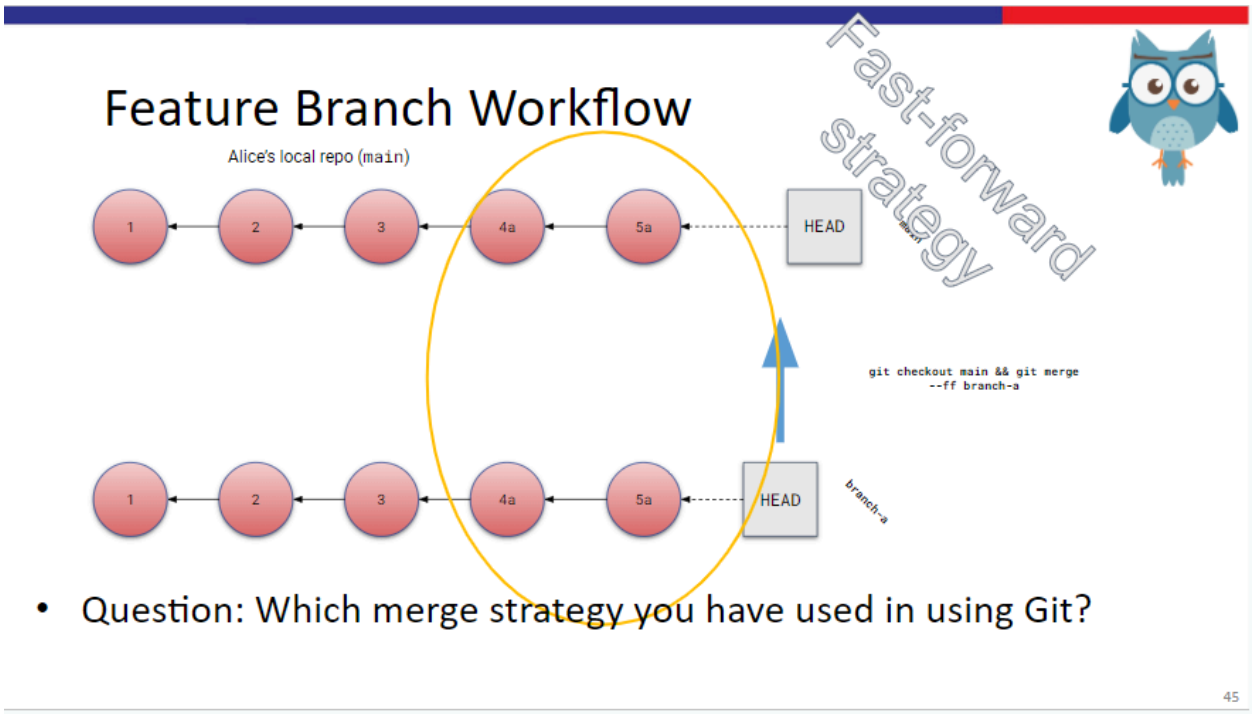


We can also create a new branch from specific commit in the `main` branch. For example, we want to create `branch-b` based on the second commit in `main` branch. We can use command: `git branch branch-b HEAD-1` and then `git checkout branch-b`.

Question: what is the git command if we want to create a new branch (branch-c) based on the first commit in main branch?

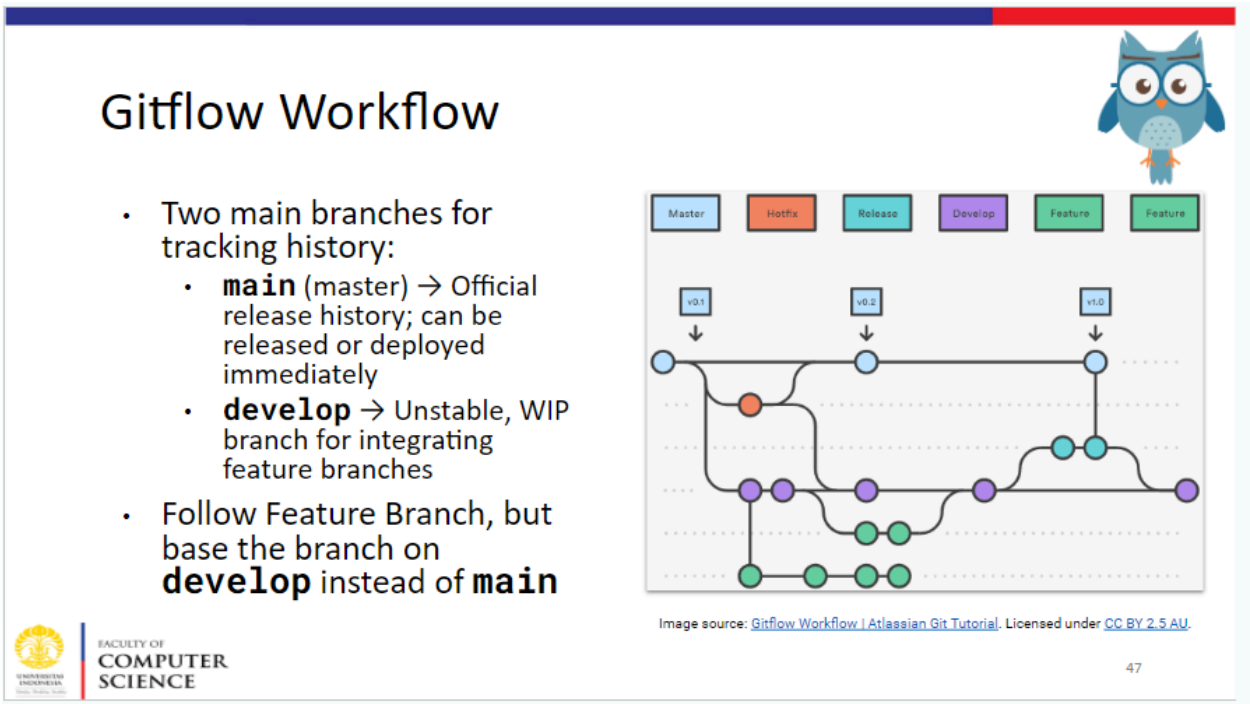


Alice adds a few commits (4a, 5a) to her feature, and she pushes these commits in her feature branch. Once she completes her feature, she plans to merge it into main. First, move to main branch (`git checkout main`) and then merge main branch with branch-a (`git merge –no-ff branch-a`). Using merge commit strategy, a new commit (Merge Commit) is created in the main branch.



We can also use fast-forward strategy in feature branch workflow. A fast-forward strategy can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one.

Other Workflows: Gitflow, Forking



Gitflow is an alternative Git branching model that involves the use of feature branches and multiple primary branches. Instead of a single main branch, this workflow uses two branches to record the history of the project. The `main` branch stores the official release history, and the `develop` branch serves as an integration branch for features. Git workflow follows Feature Branch workflow. However, feature branches use `develop` as their parent branch. When a feature is complete, it gets merged back into `develop`. Features should never interact directly with `main` branch.

Gitflow-specific Branches

- Release
  - Prepare for release/deploy, which will be done once the changes merged into `main`
- Hotfix
  - Fix issues (e.g. bugs) in the current release/deploy
  - The result is introduced back into `main` & `develop` branch

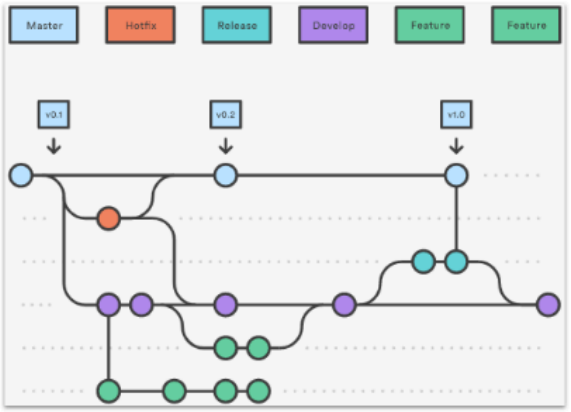


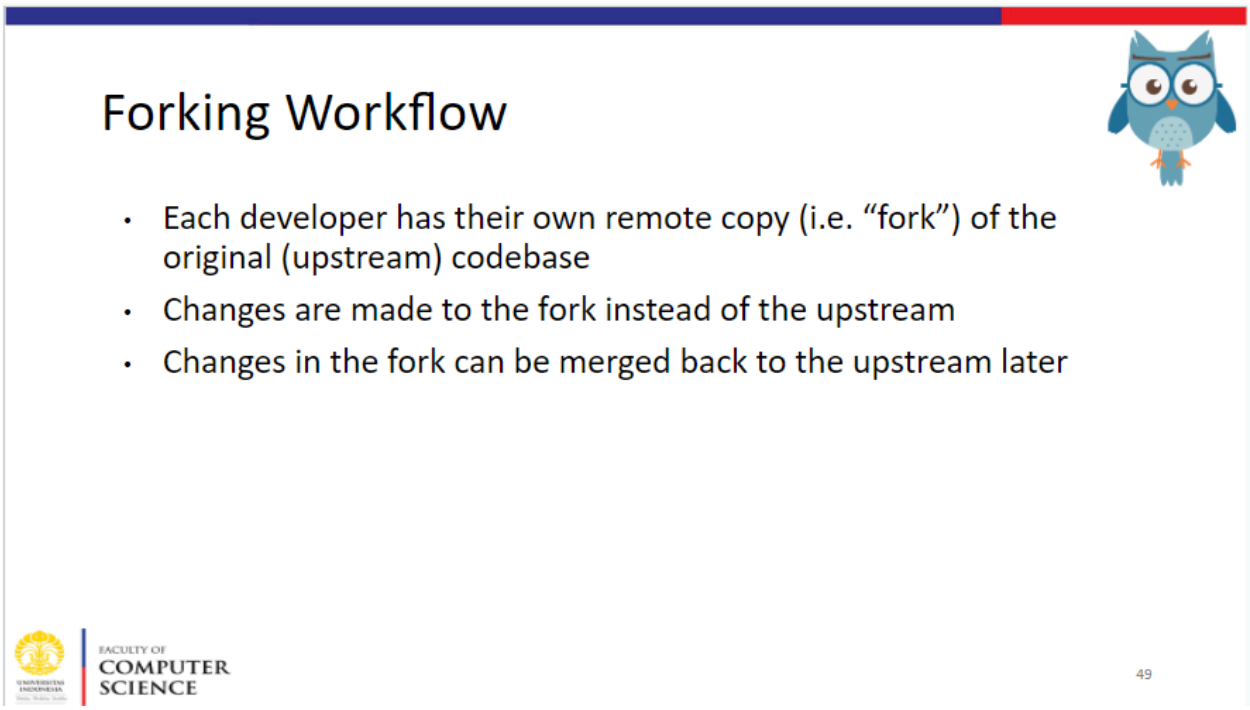
Image source: Gitflow Workflow | Atlassian Git Tutorial. Licensed under CC BY 2.5 AU.

FACULTY OF COMPUTER SCIENCE

48

`Release` branch can be created when `develop` has acquired enough features for a release. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. The `release` branch gets merged into `main` and tagged with a version number. In addition, it should be merged back into `develop`, which may have progressed since the release was initiated.

`Hotfix` branches are used to quickly fix the production releases. `Hotfix` branches are created based on `main` instead of `develop`. As soon as the fix is complete, it should be merged into both `main` and `develop` (or the current release branch).



The Forking Workflow is different than other popular Git workflows. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer their own server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one. The Forking Workflow is most often seen in public open-source projects.

The main advantage of the Forking Workflow is that contributions can be integrated without the need for everybody to push to a single central repository. Developers push to their own server-side repositories, and only the project maintainer can push to the official repository.

Recommended Workflow

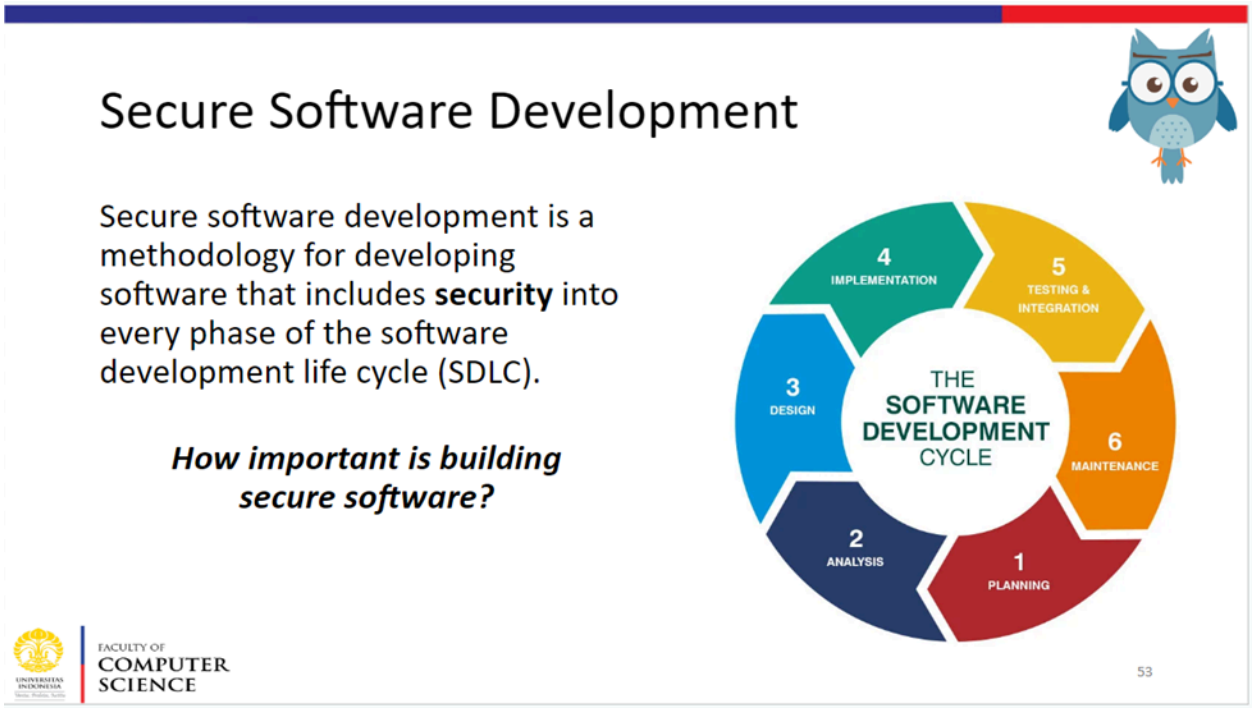
	Pros	Cons	Suitable For
Trunk	Simple, no branches; May have a very clean history if each commit resulted in a successful build; Simple CI/CD pipelines;	Frequent merge conflicts; Possible to have broken build at certain revision; Cluttered history;	Solo dev; small team;
Feature Branch	Simpler than Gitflow (assuming the main branch is the prod. environment/release branch); Cleaner history on the main branch; Merge conflict may only happen when merging branches;	Needs Git discipline in the dev. team (e.g., an ideal merge commit should be self-contained, descriptive, and will not break the build);	Project-based development (e.g., one-off project or "beli-putus" project); Releasable/deployable product is based on a single main branch;
Feature Branch + Environment/Release Branches	Similar to the pros of Feature Branch; Include environment/release branches; Can maintain several codebase versions	Similar to the cons of Feature Branch; Extra work on setting up CI/CD pipelines due to multiple environment and/or release branches	Product-based development (e.g., maintaining a product line for several customers);
Gitflow	Well-established branching rules (one of the oldest Git-based workflow); Can be considered as the more complex version of Feature Branch;	Similar to the cons of Feature Branch + Environment/Release Branches; Extra red tape in merging branches;	Project-based or product-based development; Dev. team with high Git discipline
Forking	Isolated development from the upstream repo (inc. the CI/CD pipelines and deployment environments if exist)	Limited visibility, esp. when the forks are private; Need to set up separate CI/CD pipelines and deployment environments;	Open-source project; contractor/outsource; solo dev or dev. team w/ DevOps competencies

50

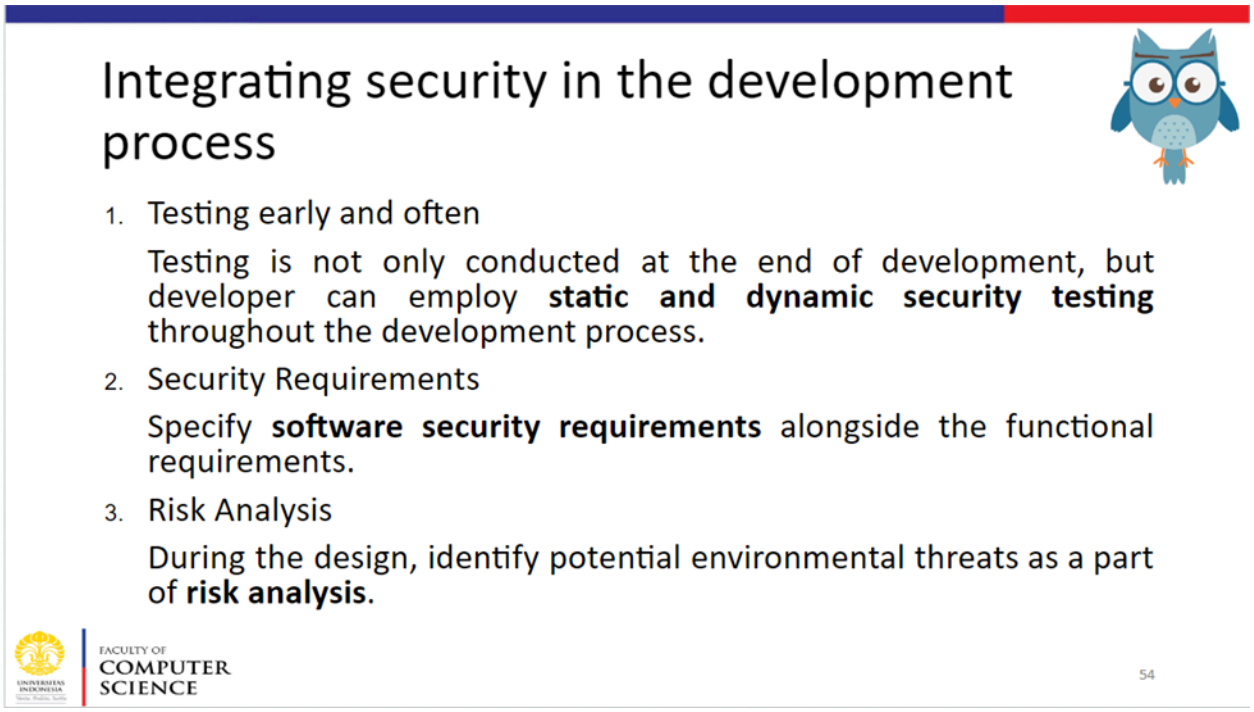
The table above summarizes the comparison between Trunk (centralized workflow), Feature Branch, Gitflow, and Forking. Each workflow has pros and cons. Which workflow do you prefer? When evaluating a workflow for our team, it's most important that we consider our team's culture and behavior. We want the workflow to enhance the effectiveness of our team and not be a burden that limits productivity.

3. Secure Coding

Security is about controlling who can interact with your information, what they can do with it, and when they can interact with it. Software security encompasses the measures and practices taken to ensure the safety, protection, and integrity of individuals, assets, information, systems, and environments from various threats and risks. Security can be applied to different domains, including physical security, information security, network security, and more.

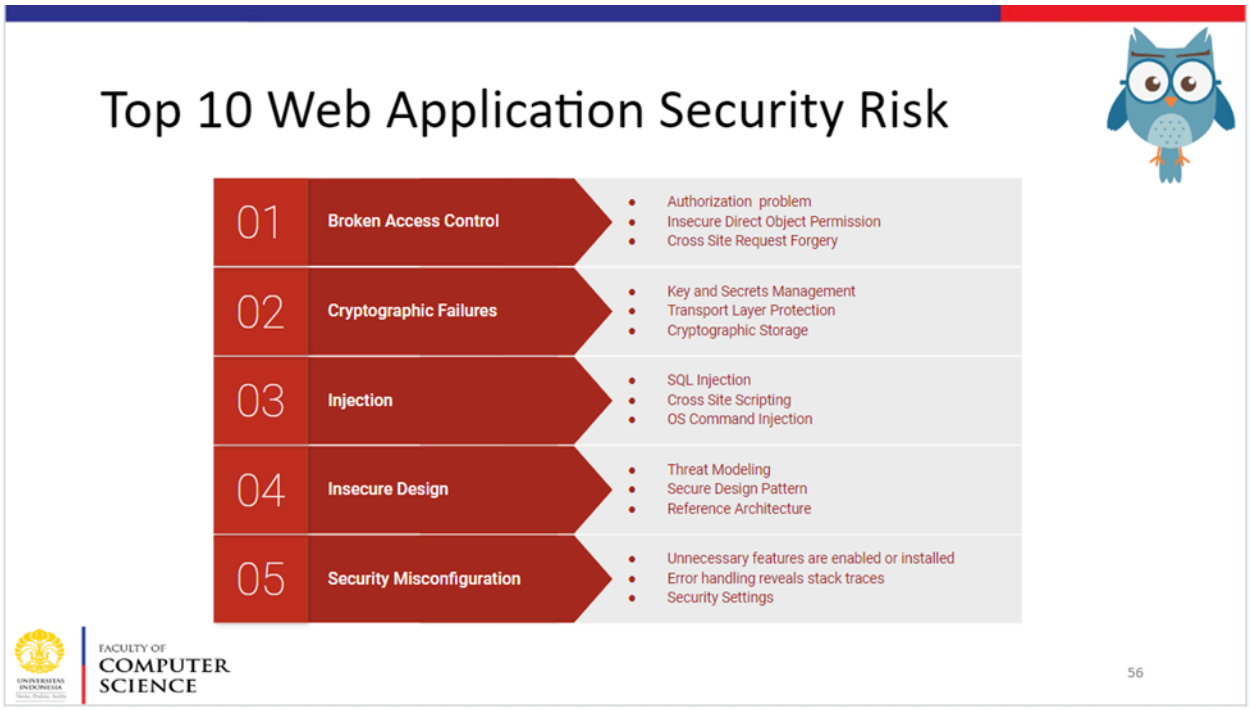


Secure software development is a methodology for building applications with a focus on integrating security measures throughout the entire software development lifecycle (SDLC). There are six main stages in SDLC: planning, analysis, design, implementation, testing & integration, and maintenance. In secure SDLC, security actions are built into each stage, not only in the implementation or testing stages. How to integrate security in the development process?



To integrate security actions in the development process, we can do the following actions:

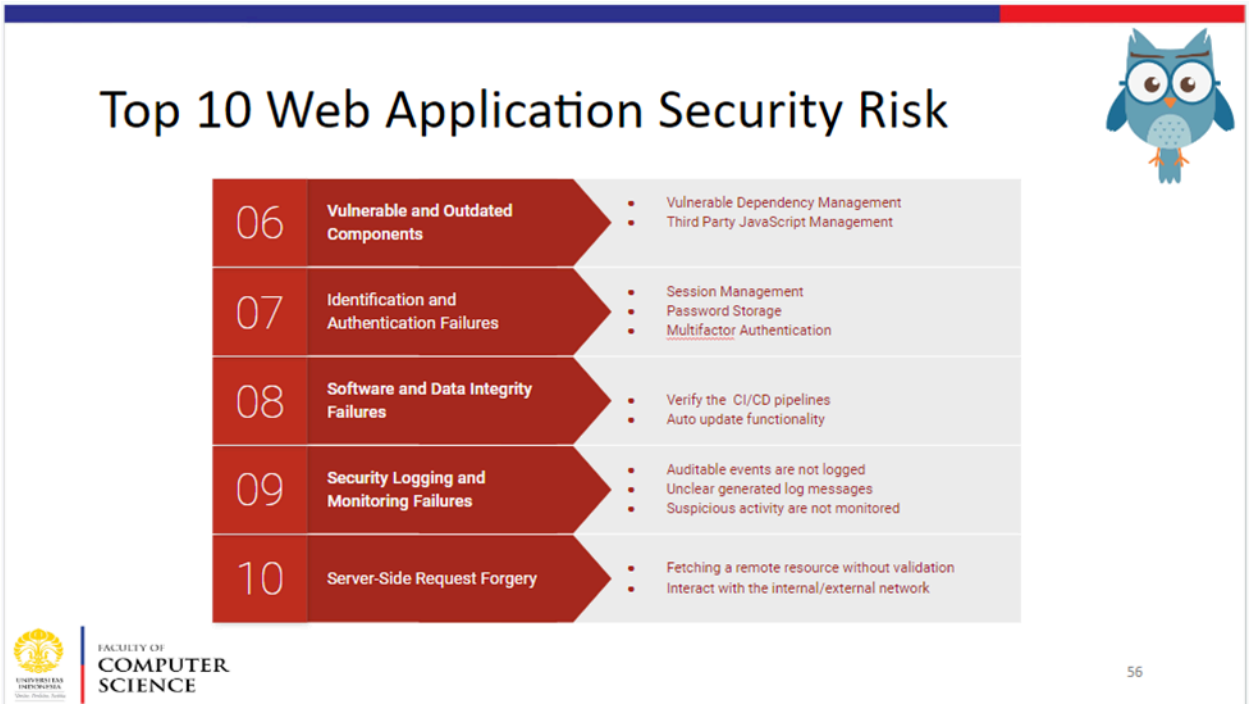
1. Testing early and often
   In traditional SDLC, testing is only conducted at the end of development. As a consequence, issues or problems are not detected in the early stages of the development process. Testing early and often can help the system to prevent and detect security issues as soon as possible.
2. Security Requirements
   Software security requirements can be integrated with functional requirements. This integration ensures that security considerations are embedded into the design and functionality of the application.
3. Risk analysis
   Incorporating security considerations into risk analysis is a fundamental aspect of creating a comprehensive and effective security strategy for software development. Identifying potential attacks or threats allows developers and stakeholders to prioritize security aspects and allocate resources effectively.



A website security risk is the potential result of a malicious attack. Based on OWASP consensus about the most critical security risks to web applications, the Top 10 Risks in 2021 are:

1. Broken Access Control is the most critical security risk that relates to authentication and authorization problems. It might happen when the product does not restrict, or incorrectly restricts, access to a resource.

2. Cryptographic Failures occur when the cryptographic security control is either broken or not applied, and the data is exposed to unauthorized actors. Don't store sensitive data unnecessarily and make sure to encrypt all sensitive data at rest.
3. Injection may be caused by a lack of input validation and sanitization. Some of the more common injections are SQL injection, cross-site scripting, and operating system (OS) command injection.
4. Insecure Design recognizes risks related to design flaws. In practice, a secure development lifecycle encourages the identification of security requirements, the periodic use of threat modeling, and the consideration of existing secure libraries and frameworks.
5. Security Misconfiguration occurs when system or application configuration settings are missing or are erroneously implemented, allowing unauthorized access.



6. Vulnerable and Outdated Components refer to when open-source or proprietary code contains software vulnerabilities or is no longer maintained.

7. Identification and Authentication Failures are security vulnerabilities that can occur when a system or application fails to identify or authenticate a user correctly. This can allow attackers to gain unauthorized access to systems and data.

8. Software and Data Integrity Failures relate to code and infrastructure that do not protect against integrity violations.

9. Security Logging and Monitoring Failures occur when a system or application fails to log or monitor security events properly. This can allow attackers to gain unauthorized access to systems and data without detection.

10. Server-Side Request Forgery occurs whenever a web application is fetching a remote resource without validating the user-supplied URL. These exploits allow an attacker to coerce the application to send a constructed request to an unexpected destination.

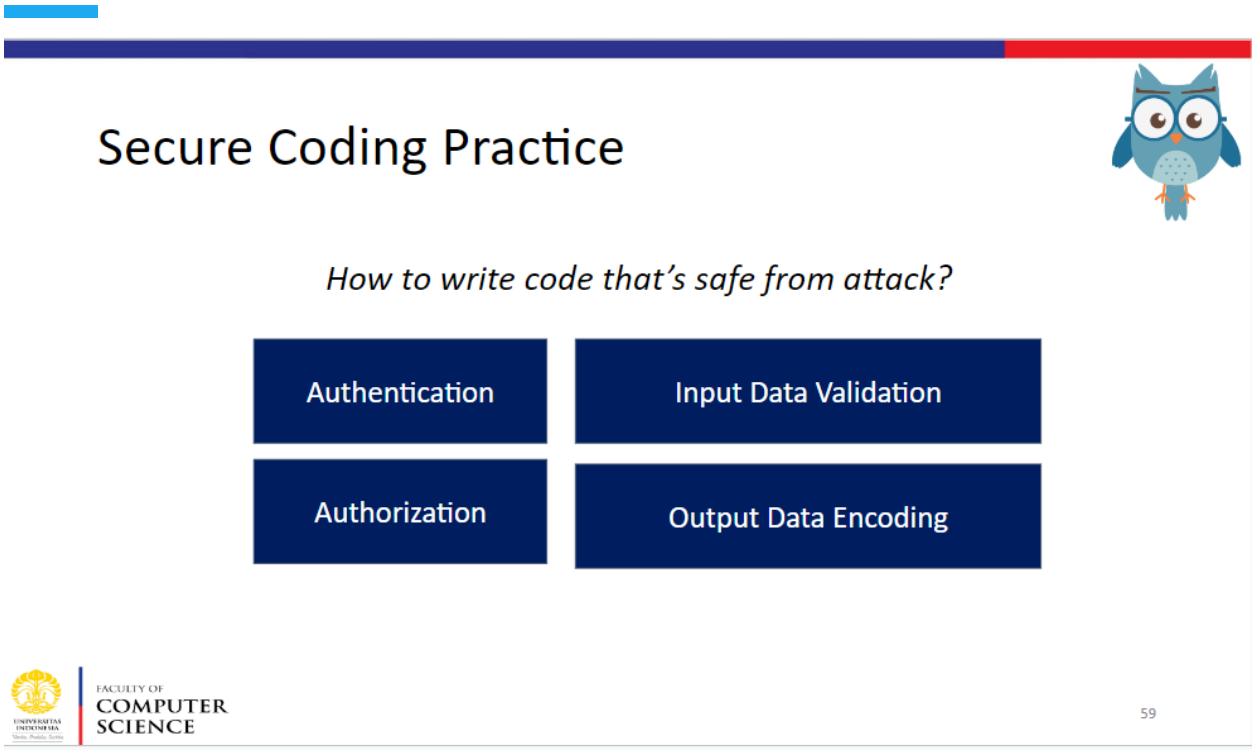
Secure Coding

- Secure coding: the practice of coding or programming that protects the applications from security threats and vulnerabilities.
- The goal is to develop applications that are resistant to various types of attacks and can withstand malicious activities.
- Secure coding is crucial for maintaining the confidentiality, integrity, and availability of data and applications.

FACULTY OF COMPUTER SCIENCE

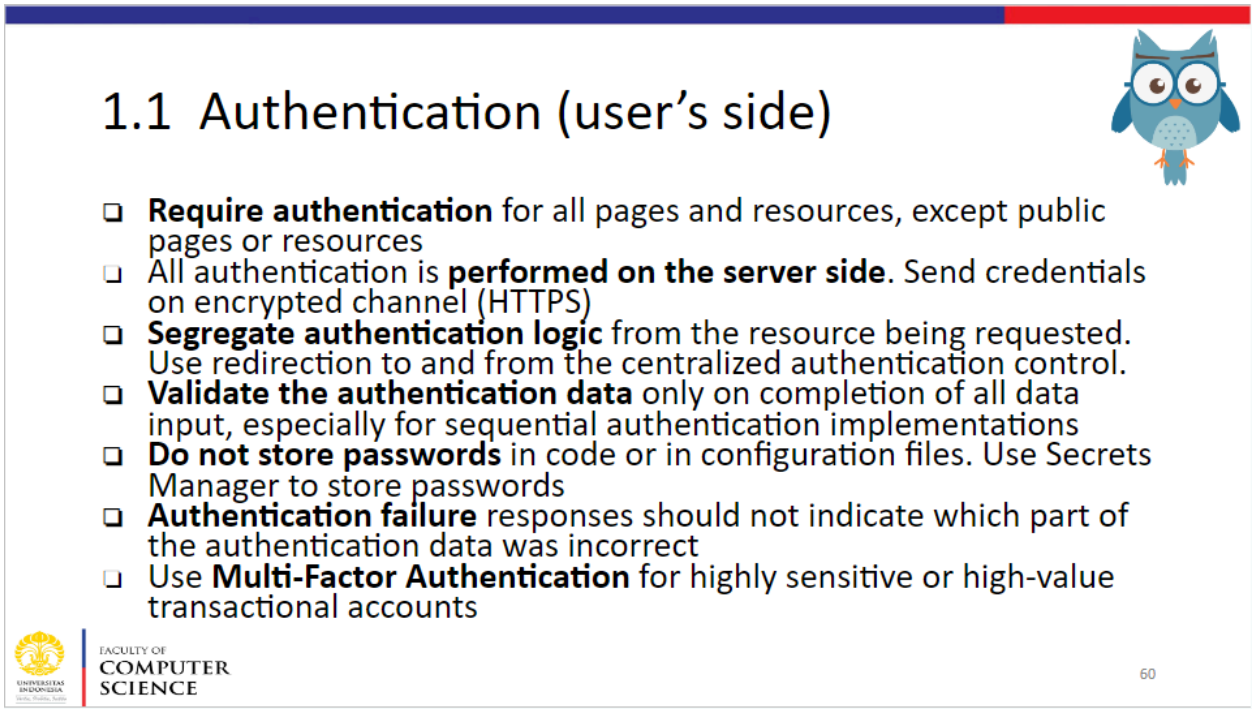
57

Secure coding refers to the practice of writing computer programs, applications, or software in a way that protects them from security threats and vulnerabilities. The goal of secure coding is to develop applications against various types of attacks and unauthorized access that could lead to data breaches, service disruptions, or other security incidents. Secure coding is a crucial process for maintaining the confidentiality, integrity, and availability of data and systems. Secure coding is essential for protecting information, systems, and users from potential threats and vulnerabilities. Developers should follow secure coding practices to minimize vulnerabilities in the code.

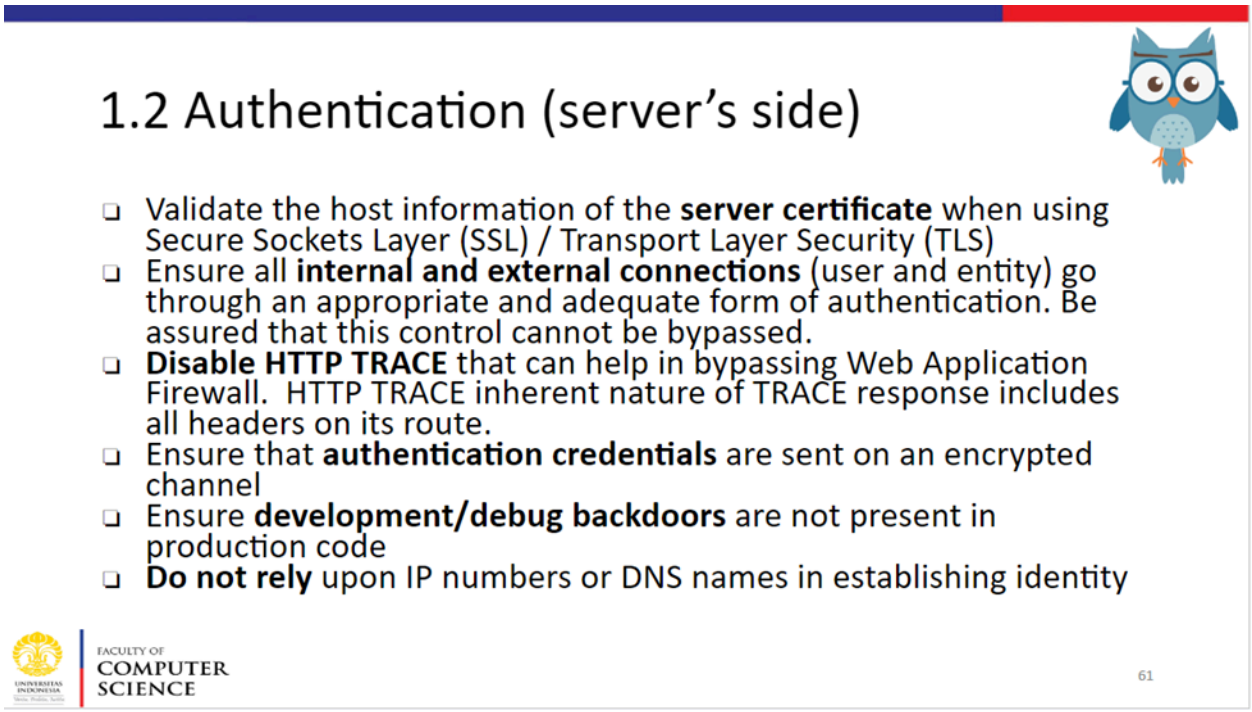


Previously, we have learned the top 10 security risks in web applications. The Open Web Application How to write code that is safe from attack? Security Project (OWASP) provides guidelines for secure coding practices to help developers build more resilient and secure web applications. There are four main categories of Secure Coding Practice in OWASP developer guide: Authentication, Authorization, Input Data Validation, and Output Data Encoding.

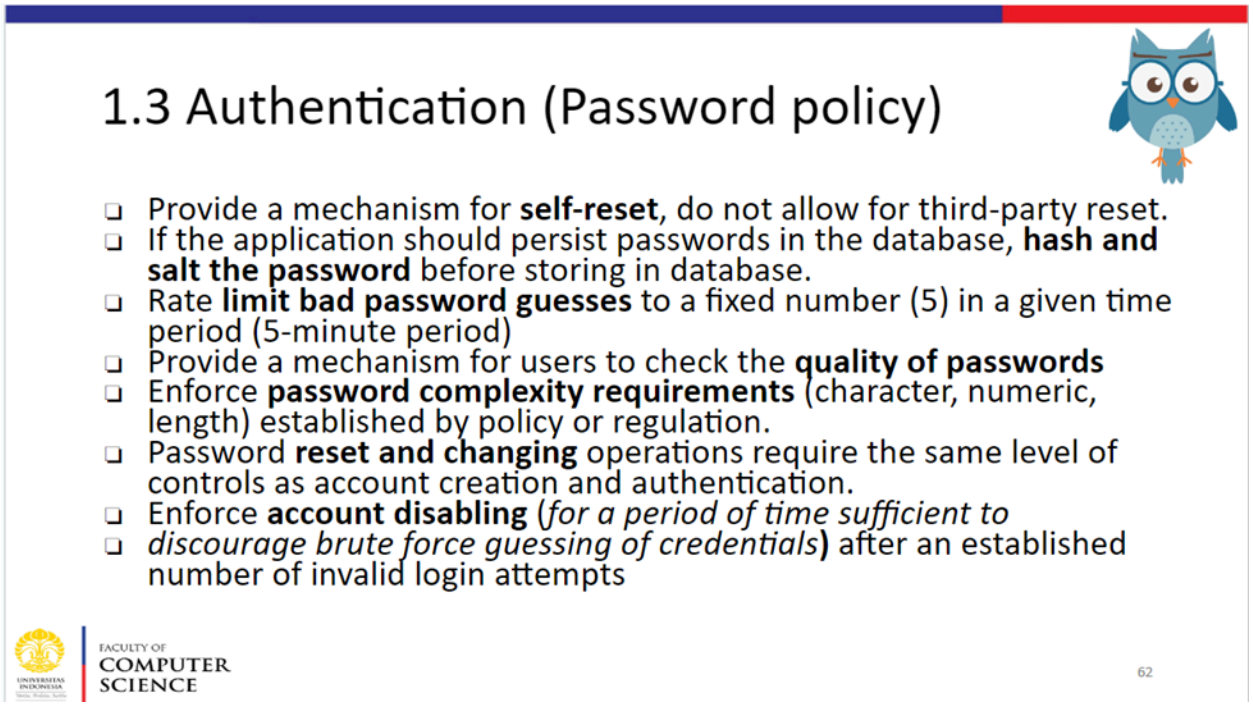
Authentication



Authentication is the process of verifying that an individual, entity, or website is who/what it claims to be. In secure coding, we should consider authentication from the user's side, as mentioned on the checklist above. Authentication does not only deal with username and password but also how to protect our system from unexpected attacks.

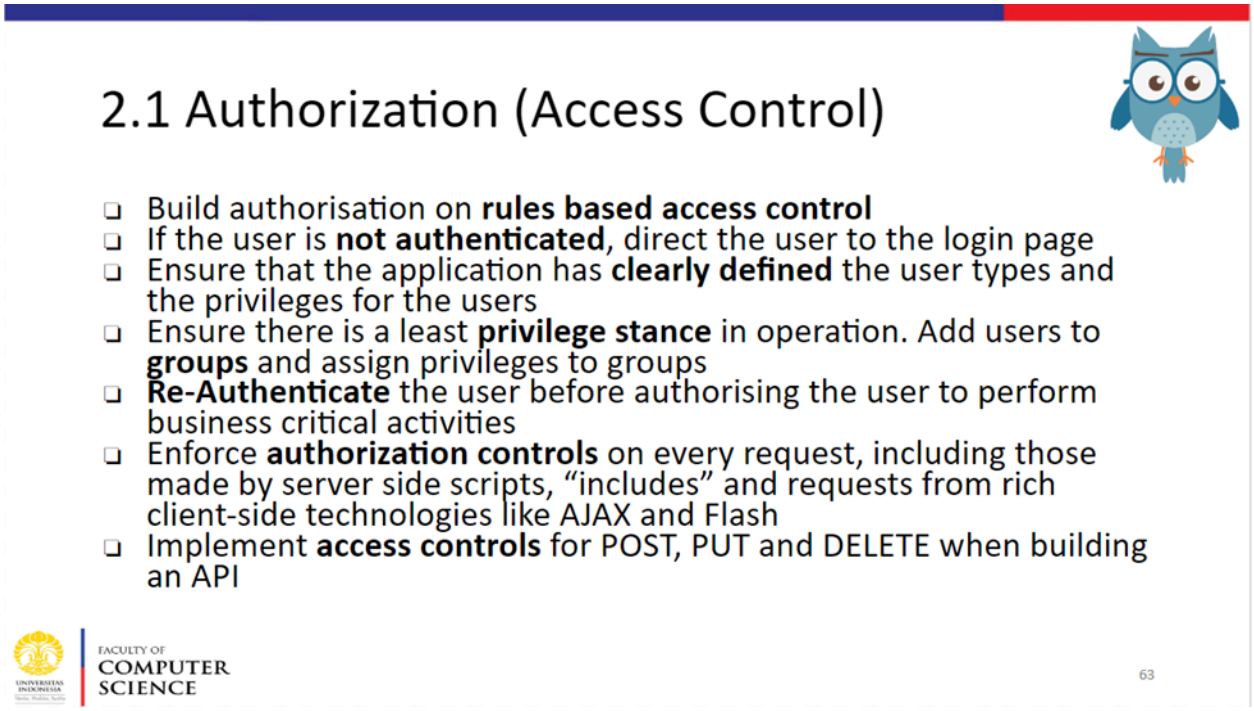


Authentication must be managed from the server side to prevent malicious attacks. For example, the server must validate the SSL certificate which is crucial for ensuring the security of data exchanged between a user's web browser and a website's server. A checklist for secure coding practice in Authentication from the server's side is defined by OWASP, as mentioned above.

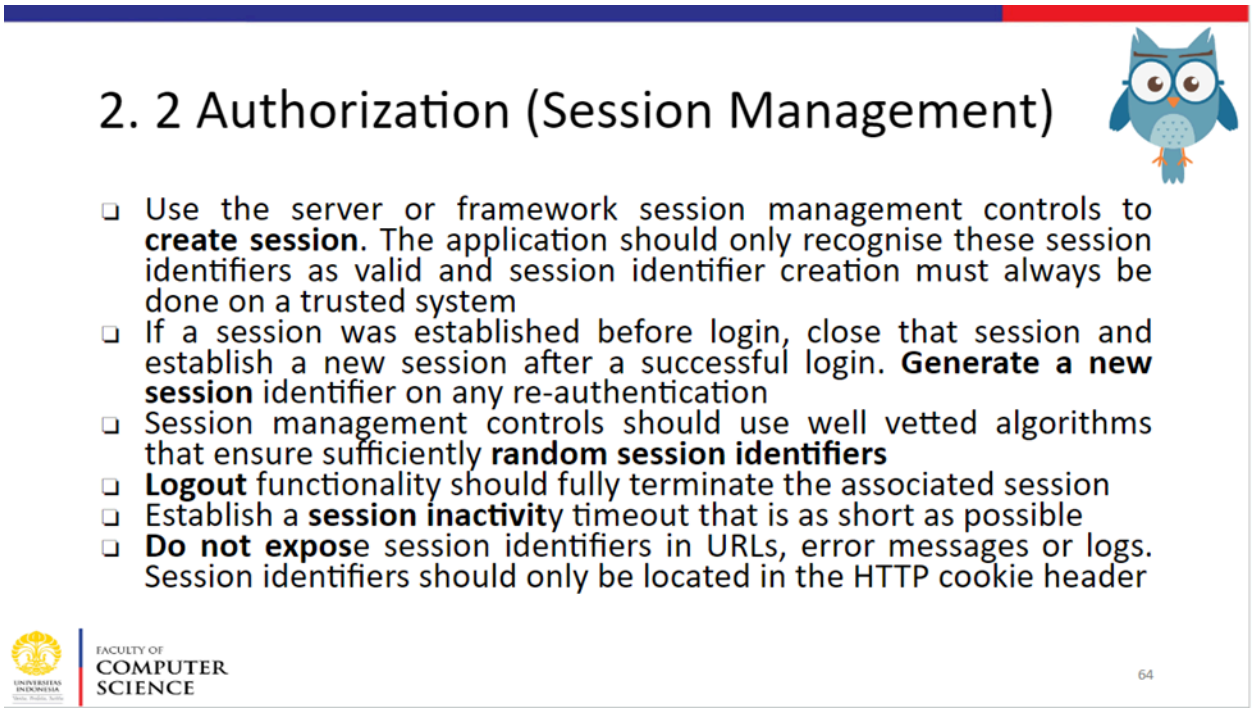


Have you ever forgotten your password to access a system? Different applications may have different mechanisms to reset the password. Passwords are a primary means of authenticating users and verifying their identities. A strong password policy ensures that only authorized individuals can access a system, application, or network. However, not all users know how to create a strong password that protects their account. By implementing and enforcing strong password policies, developers can significantly enhance overall software security. The checklist above is an example of password policies that can be applied in secure coding practice.

Authorization

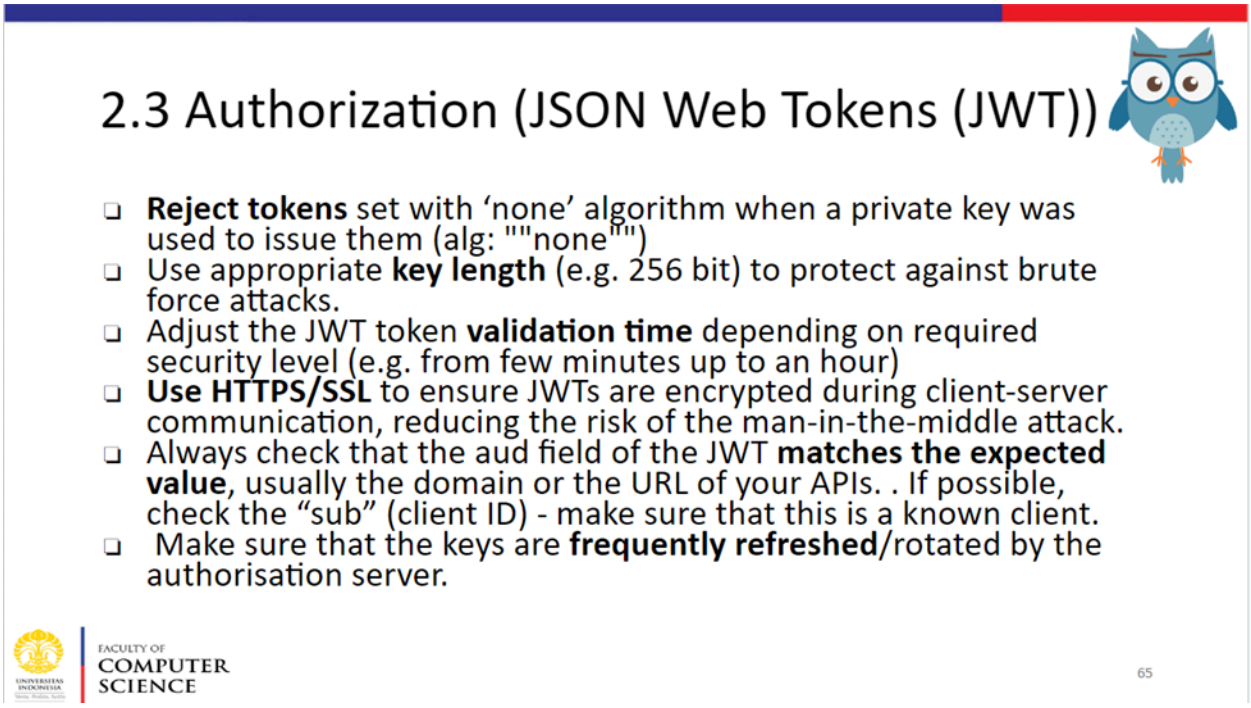


Authorization is distinct from authentication which focuses on verifying an entity's identity. Authorization focuses on determining the permissions or privileges of an authenticated entity. As a student at Fasilkom UI, you are authorized to read resources in SCeLe. How to manage the authorization to improve the software security? OWASP provides guidelines for Authorization from access control perspectives, as defined in the checklist above. Effective access control mechanisms play a crucial role in preventing unauthorized access.



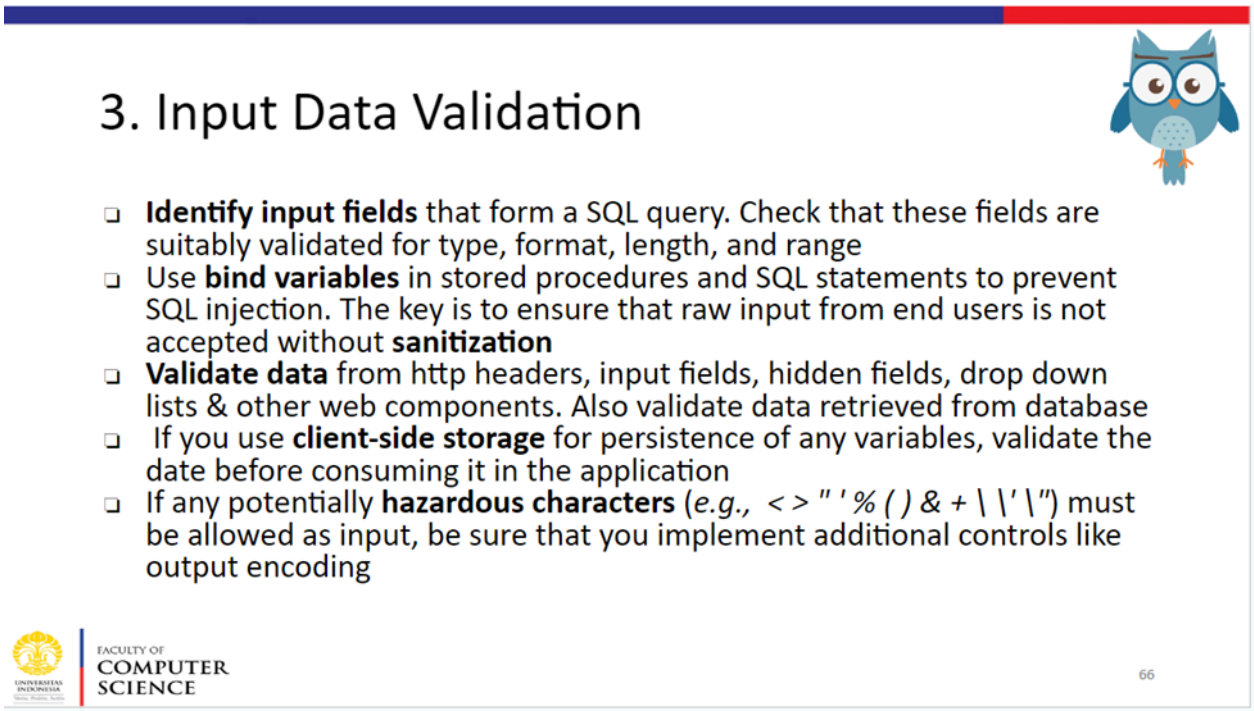
A session refers to the period during which a user interacts with a web application or a service. A session typically starts when a user logs in or establishes some form of connection and continues until the user logs out, closes the browser, or the session otherwise expires due to inactivity.

Session management is a critical aspect of web application security, focusing on how user sessions are established, maintained, and terminated. By implementing secure session management practices, web applications can minimize the risk of session-related security vulnerabilities, protect user privacy, and ensure a more resilient defense against attacks such as session hijacking or fixation. OWASP provides guidelines on how to manage sessions to improve software security, as mentioned above.



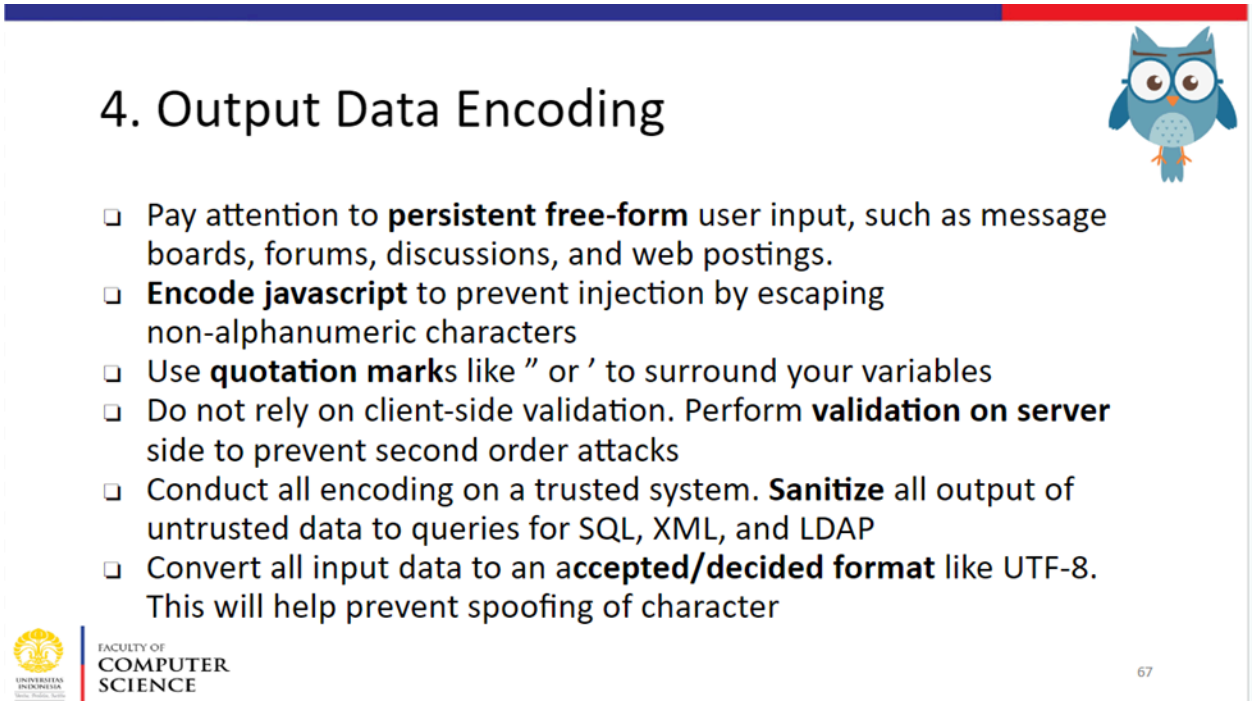
JSON Web Tokens (JWT) are a widely used method for representing claims securely between two parties. JWT can carry information about a user's roles, permissions, and other attributes. This information is used for authorization, allowing services to make access control decisions based on the claims in the JWT. The checklist above may help you to prevent vulnerabilities that might be caused by JWT. Developers should carefully consider the choice of algorithms, validate tokens on the server side, and follow security guidelines to mitigate potential risks associated with JWT usage.

Input Data Validation



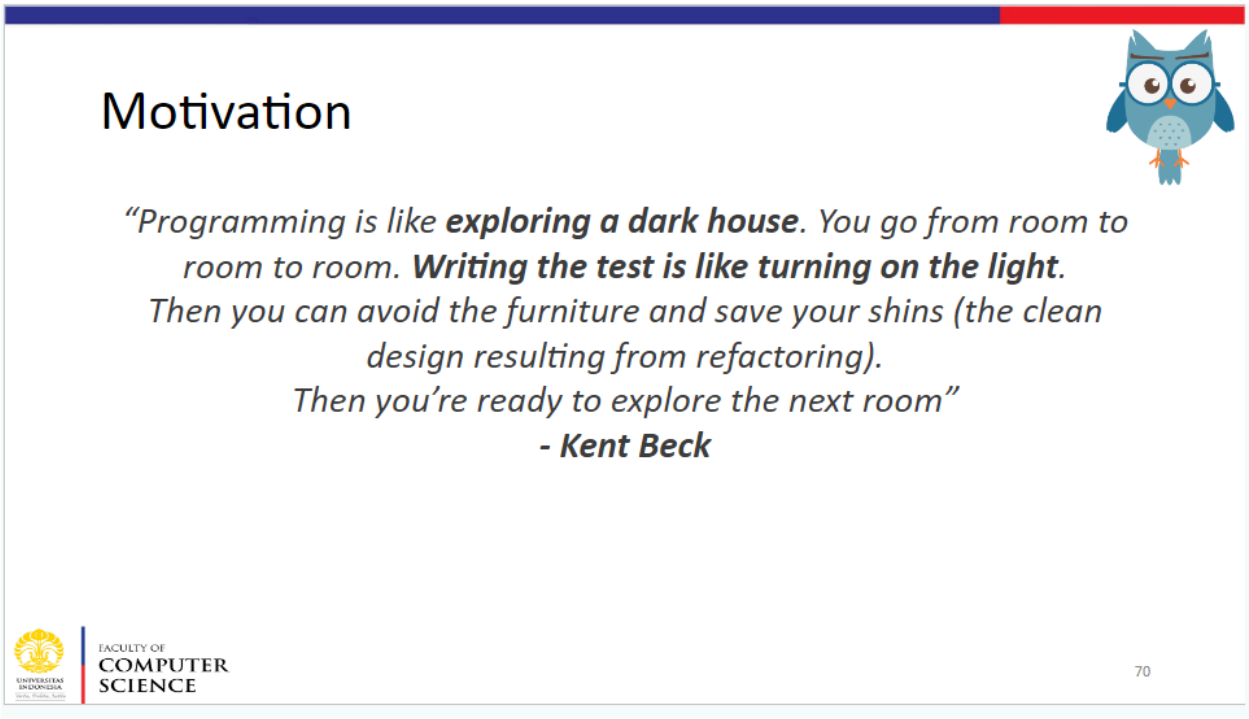
What's wrong with input data? Input data can be a source of various security vulnerabilities and issues if not properly validated and handled. Input data validation ensures that the data received by an application is of the expected type, format, and range, reducing the risk of injection attacks, data manipulation, and other security issues. To mitigate these risks, developers should implement thorough input validation, sanitize user inputs, and follow secure coding practices for input data validation, as mentioned above.

Output Data Encoding



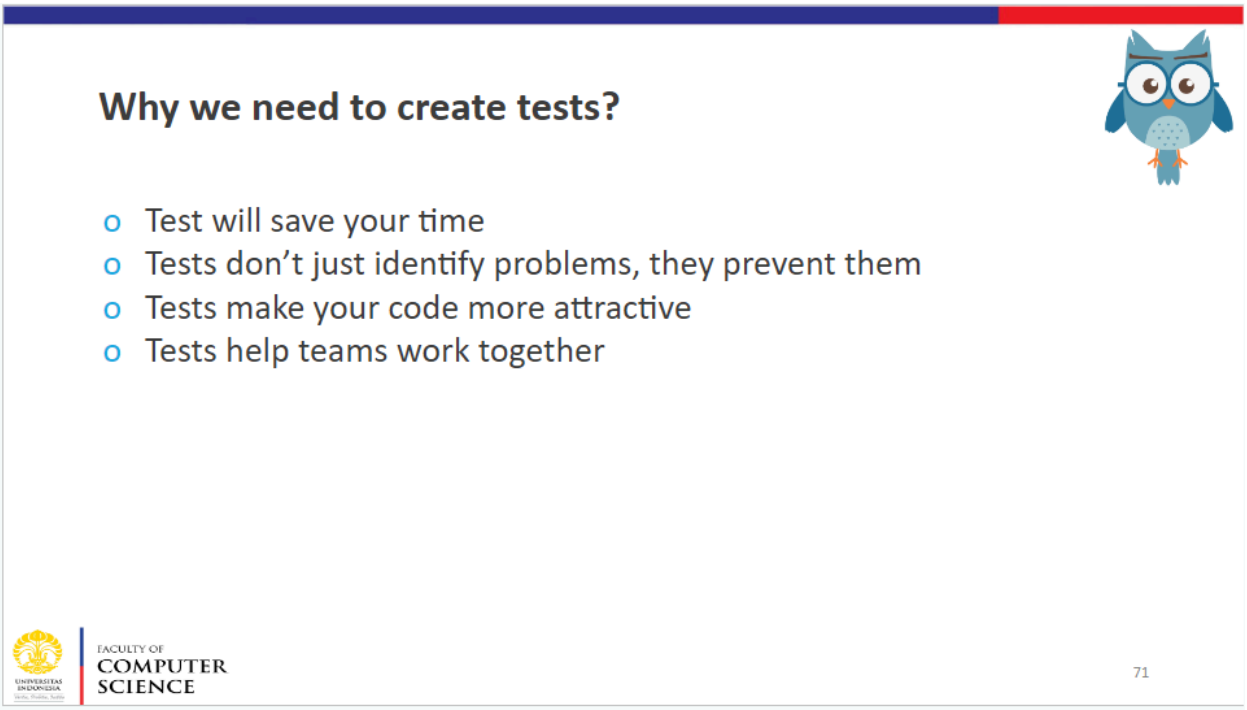
The purpose of output data encoding in the context of secure coding is to prevent security vulnerabilities, particularly those related to injection attacks. Output data encoding involves transforming data into a safe and properly formatted representation before displaying or incorporating it into a different context.

4. Testing



Kent Beck said that programming is like exploring a dark house. How do you feel? Confused or scared? Writing the test is like turning on the light. The test helps light your way to understand the path, and what kinds of stuff are in the room. Then you are ready to explore the next room, to write other programs without being confused or scared anymore.

So, testing is an important part of programming. Let's see in more detail, why we need to create a test.



1. Testing can save time in the software development process. While writing tests does require an upfront investment of time and effort, the benefits outweigh the costs in the long run.
2. Writing a test not only helps you to identify possible problems but also prevents them. So, we can minimize bugs in our source code
3. Tests make your code more attractive to readers. Tests could improve the code consistency, readability, and maintainability.
4. Testing can also support promoting collaboration and teamwork within a development team. It helps in creating a shared understanding among team members about the system's functionality and requirements. Thus, it helps team members review the code. Integrating tests into a continuous integration (CI) pipeline ensures that code changes are automatically validated.

What Should You Test?

- Success & alternative scenario
  - Test functions under conditions that let them perform their responsibility successfully or led to alternative result
  - How about getters/setters? → If they contain additional logic, test them; Otherwise, don't bother
- Error conditions (including exceptions)
- Boundary conditions

FACULTY OF COMPUTER SCIENCE

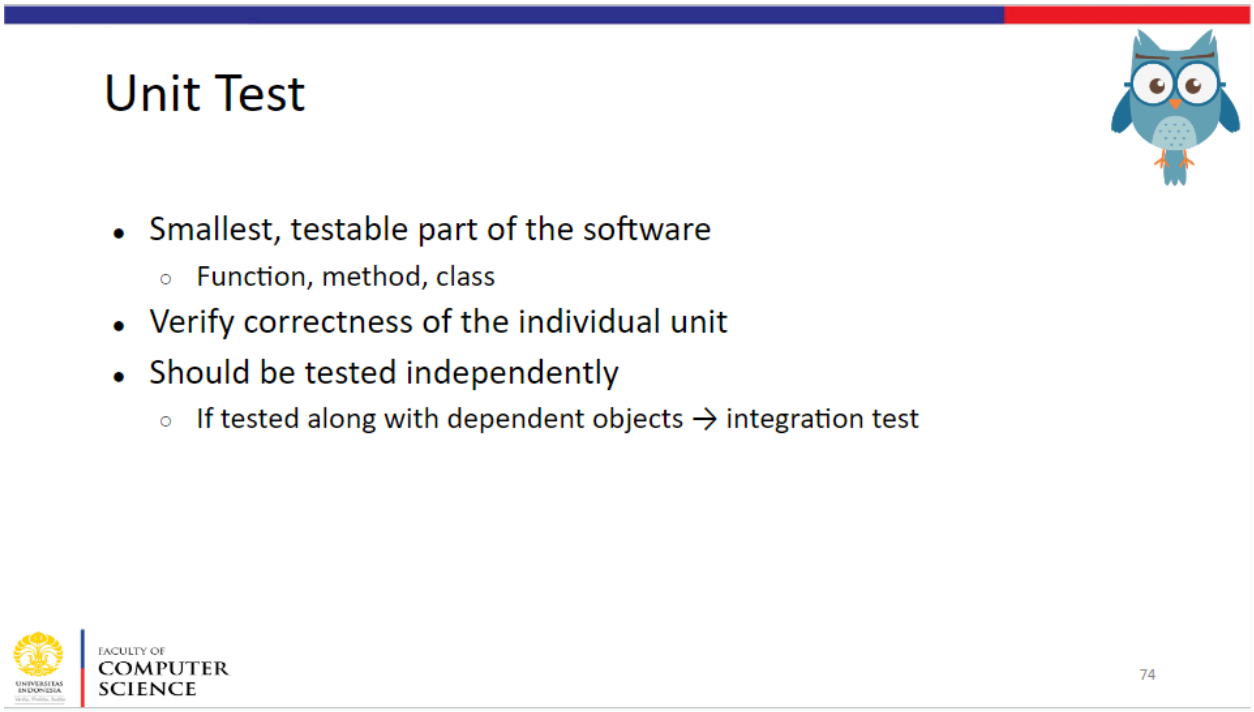
72

If you are still confused about what kind of things you should test, here are the general guidelines:

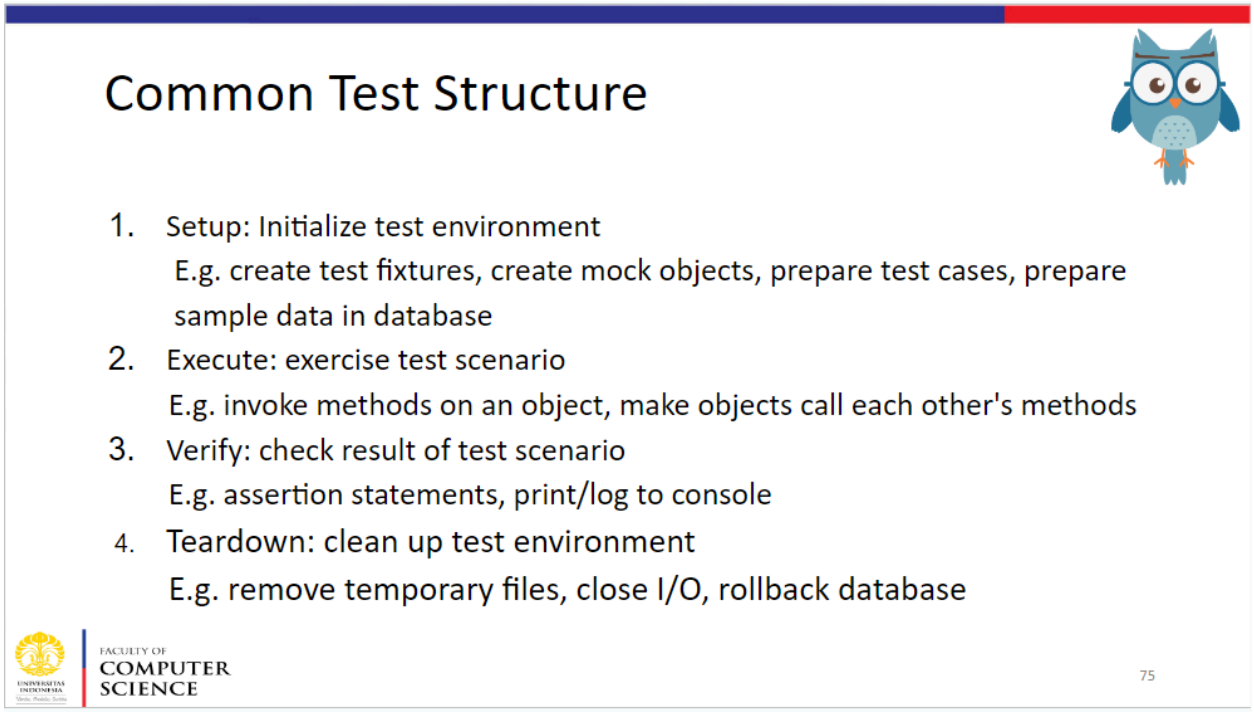
1. Test success and alternative scenario
   When creating tests for software, it is s essential to cover both success scenarios (positive tests) and alternate scenarios (negative tests) to ensure comprehensive coverage and robustness
2. Error conditions
   You should identify error conditions or situations where unexpected or undesired events occur. Identifying and handling these error conditions is essential in testing. As we learned in clean code, proper error handling also improves the readability and robustness of applications.
3. Boundary conditions
   Testing boundary conditions is often called edge case or corner case testing. Edge cases involve testing scenarios at the extremes or boundaries of the input space. For example, verifying the behavior when a user inputs the minimum or maximum allowed characters in a text field.

In this module we will learn two kinds of testing: Unit Test and Functional Test.

Unit Test

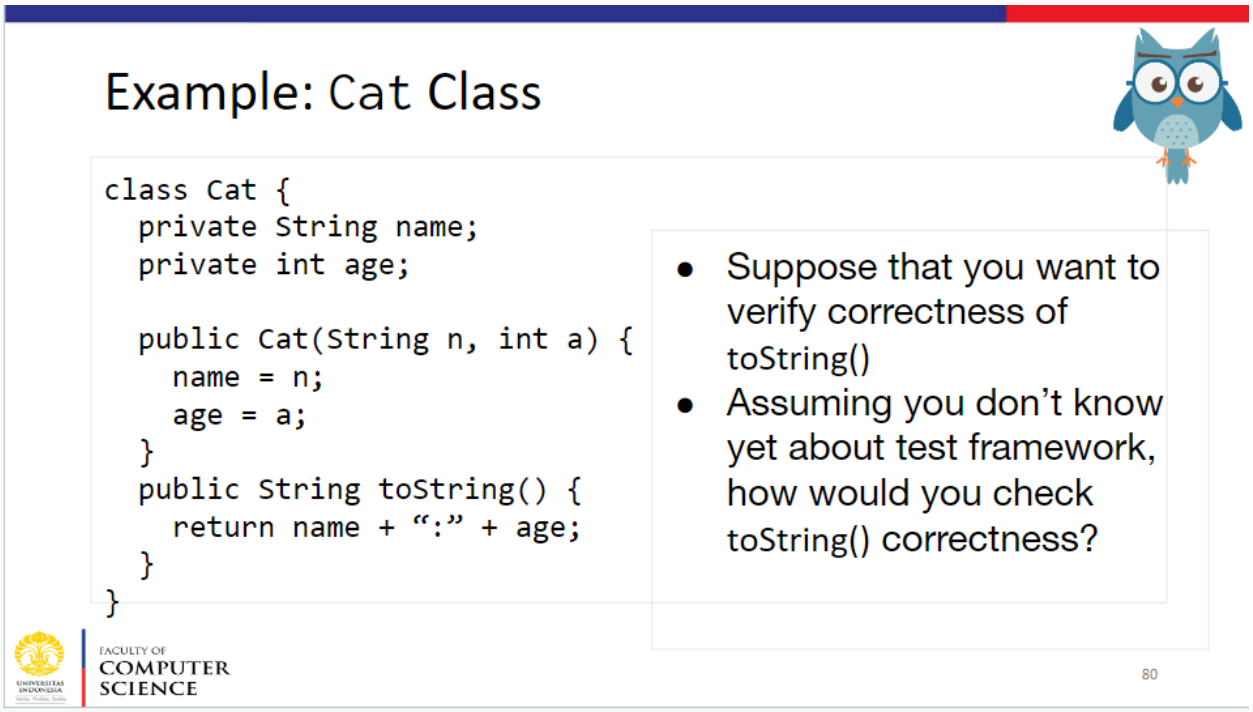


Unit test is a mechanism in software testing to validate the smallest, testable part of the software, such as function, method, or class. The goal of unit testing is to verify the correctness of the individual unit. Unit tests focus on isolating a specific unit of code for testing, so each unit should be tested independently. If a unit depends on other parts, we have to make sure that the testing process is not affected by the behavior of those other parts.



How to create the unit test? The common structure of Unit Tests is as follows:

1. **Setup.** Before conducting the test, you should initialize the test environment. The setup method is executed before each test in the test suite. For example: creating test fixtures (setup database or data fixtures) and creating mock objects.
2. **Execute.** This stage is the process to run the test scenario. We can invoke methods on an object or make objects call each other's method.
3. **Verify**. After executing the test scenario, we should check the result. In unit tests, there are assertion statements to verify whether the test result is matched with the expected result or not.
4. **Teardown.** The purpose of the teardown method is to clean up or reset any changes made to the environment during the test, ensuring a clean state for the next test.



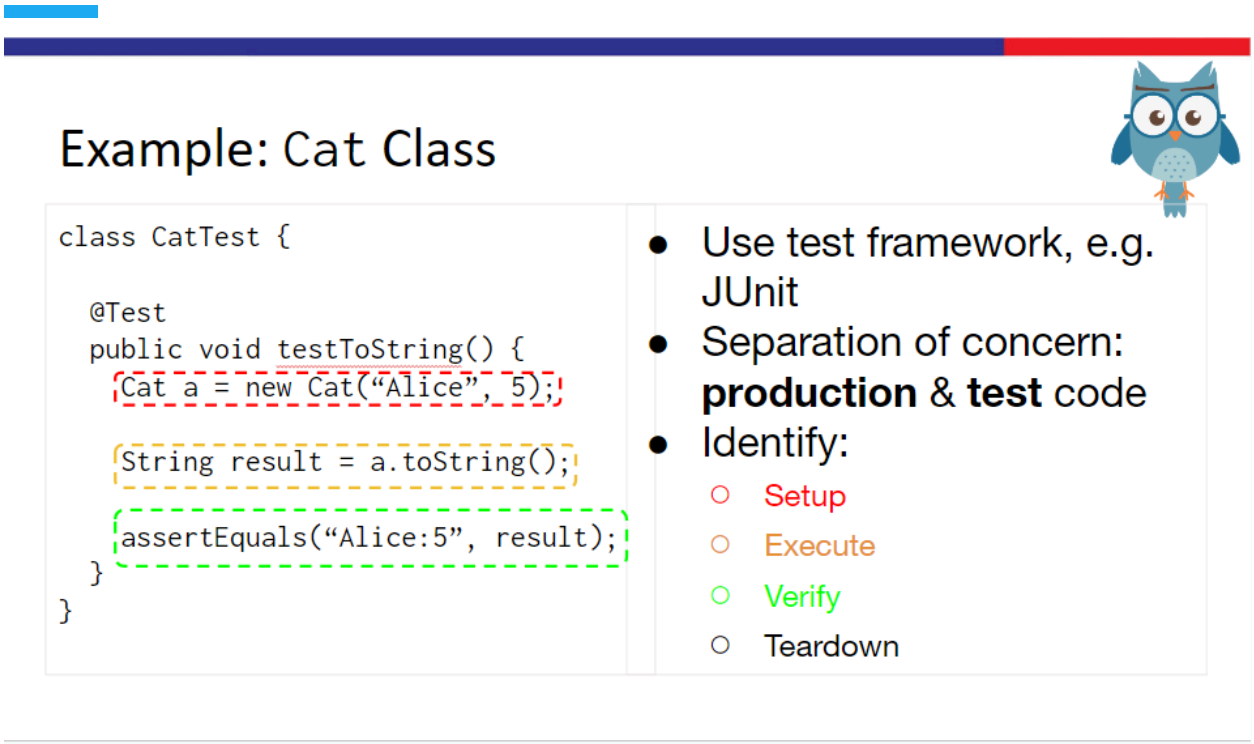
Suppose that we have a `Cat` class, as shown in the example above. This class has two fields, a constructor method, and one another method. Do you have any idea what should we test? Ok, let's test **toString()** method to verify that it will return the name and age as defined in the method. If you don't know yet about the test framework, what will you do?

Example: Cat Class

```
class Cat {
  // Omitted for brevity
  public static void main(String[]
args) {
    Cat a = new Cat("Alice", 5);

    String result = a.toString();

    System.out.println(result);
  }
}
```

- Main method can be used for testing a class
- Identify:
  - Setup
  - Execute
  - Verify
  - Teardown
- Hard to maintain when there are many tests

Without knowing the test framework, we can use the main method to test the `Cat` class. As we can see in the example above, first we create an object of `Cat` and then call the method toString(). This method call is a part of executing the test. To verify the result, we print the return value of the method call. What is the problem? It could be difficult to maintain all the testing processes if all tests are placed and performed in the main class.



Most programming languages have a unit test framework, such as unittest in Python or JUnit in Java. This framework provides a structured and systematic way to write, organize, and execute unit tests. As shown in the example, there are setup, execute, and verify stages. We can create a new Class `CatTest`, and then create a new method to test a specific method. For example, we create `testToString()` to test the toString method. At the setup stage, we create a new object and then execute the method call at the execute stage. As mentioned before, there are assertion statements in the unit test. As shown in the example, there is a method `assertEqual` to compare the expected result with the method result. We have to define the expected result in the assertion statements as a reference for verification process.

Example: Tests in Django (1)

```
class Lab2AddonUnitTest(TestCase):

    def test_lab_2_addon_url_is_exist(self):
        response = Client().get('/lab-2-addon/')
        self.assertEqual(response.status_code, 200)
    # ...
```

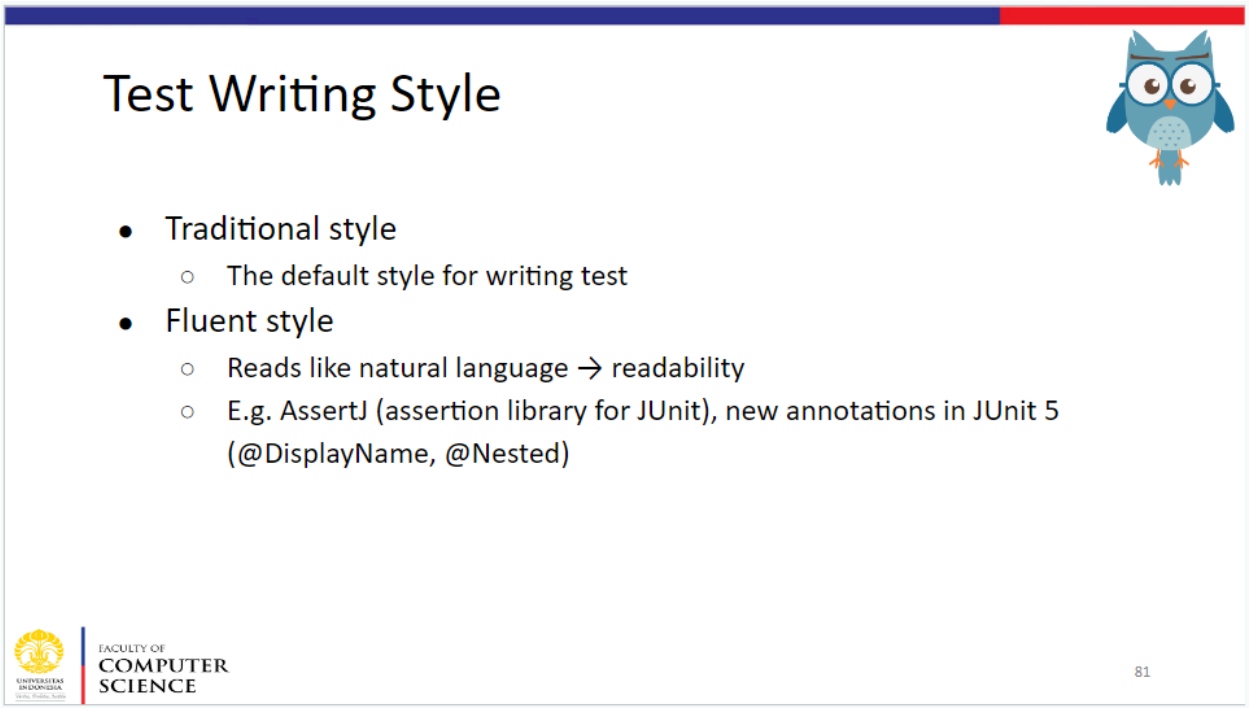
- **Pay attention to the convention used in the test framework!**
- It is possible that a test does not contain all phases mentioned in common test structure
- What were omitted in the test above?

You have learned Django in the Course “Pemrograman Berbasis Platform”. The example above shows an example of the Unit Test in Django. Each framework may have different conventions in writing the test, so pay attention to those conventions. Which part was being tested in the example above? What were omitted in the test above?

Example: Tests in Django (2)

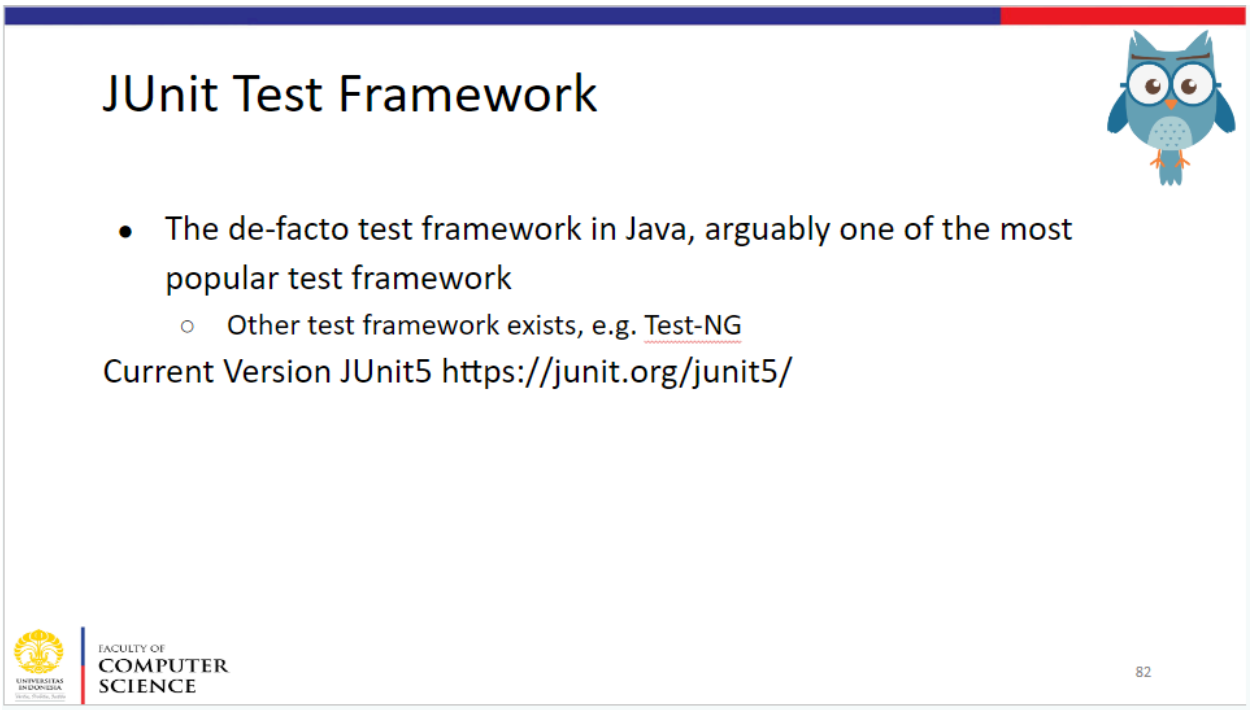
```
def test_lab2_addon_bio_shown_in_page(self):
        request = HttpRequest()
        response = index(request)
        html_response = response.content.decode('utf8')

        # Checking all family member is shown in page
        for bio in bio_dict:
            self.assertIn('<td class="subject">' + bio['subject'] +
'</td>', html_response)
            self.assertIn('<td class="value">' + bio['value'] + '</td>',
html_response)
```

Another example of Testing in Django is shown above. There are three main stages in the test, setup, execute, and verify. The unit tests verify the smallest part of our application. As shown in the example, the unit test checks whether the page contains the expected bio subject and value.



About the writing style, there are two test writing styles:

1. Traditional style follows the standard or default style for writing tests. Each test is written as a sequence of steps that explicitly outline the test setup, execution, and assertions.
2. In fluent style, the test involves chaining together test actions and assertions, providing a more natural language flow that reads like a specification. See more examples in AssertJ.



In this module, we will explain more about a Unit Test Framework in Java, Junit. JUnit is a widely-used open-source testing framework for Java programming language. It provides a platform for writing and running tests to ensure the correctness of Java applications.

The current version of JUnit is JUnit5, see https://junit.org/junit5/.

Writing JUnit 5 Tests

- Ensure JUnit is available in the classpath
  - Set manually or via build system e.g. Gradle
- Prepare separate directory to contain test code that mirror the directory structure of production code
  - E.g. `/src/test/java`

Before you write a unit test using JUnit, you have to make sure that JUnit is available in the classpath. You can set it manually, or if you are using a build system, such as Gradle, you can set up the dependency management. After finishing all the setup, you can write your test in a specific directory for managing all tests. For example, place all your test files on `/src/test/java`

Writing JUnit 5 Tests

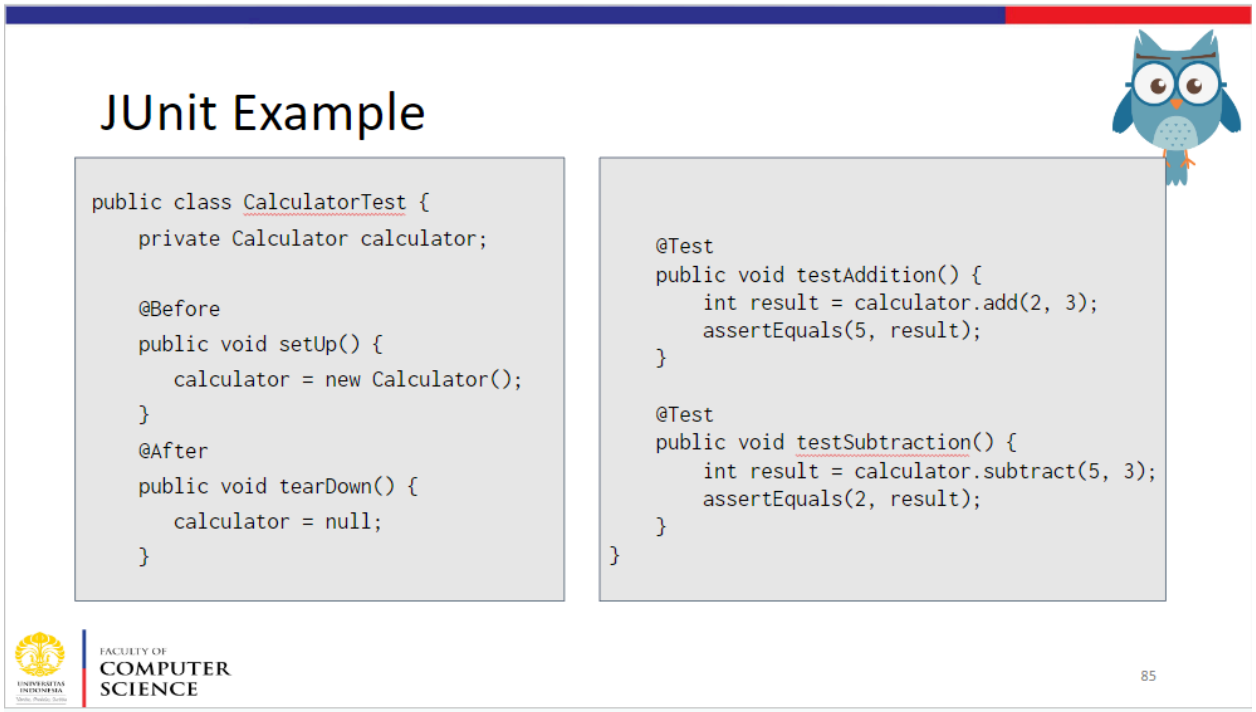
- Test cases are defined as methods in a class
  - Method annotated with `@Test`
- Setup and teardown can be extracted into their own method (annotated with `@Before`, `@After`) if all test cases in the class share the same setup & teardown logic

FACULTY OF COMPUTER SCIENCE

84

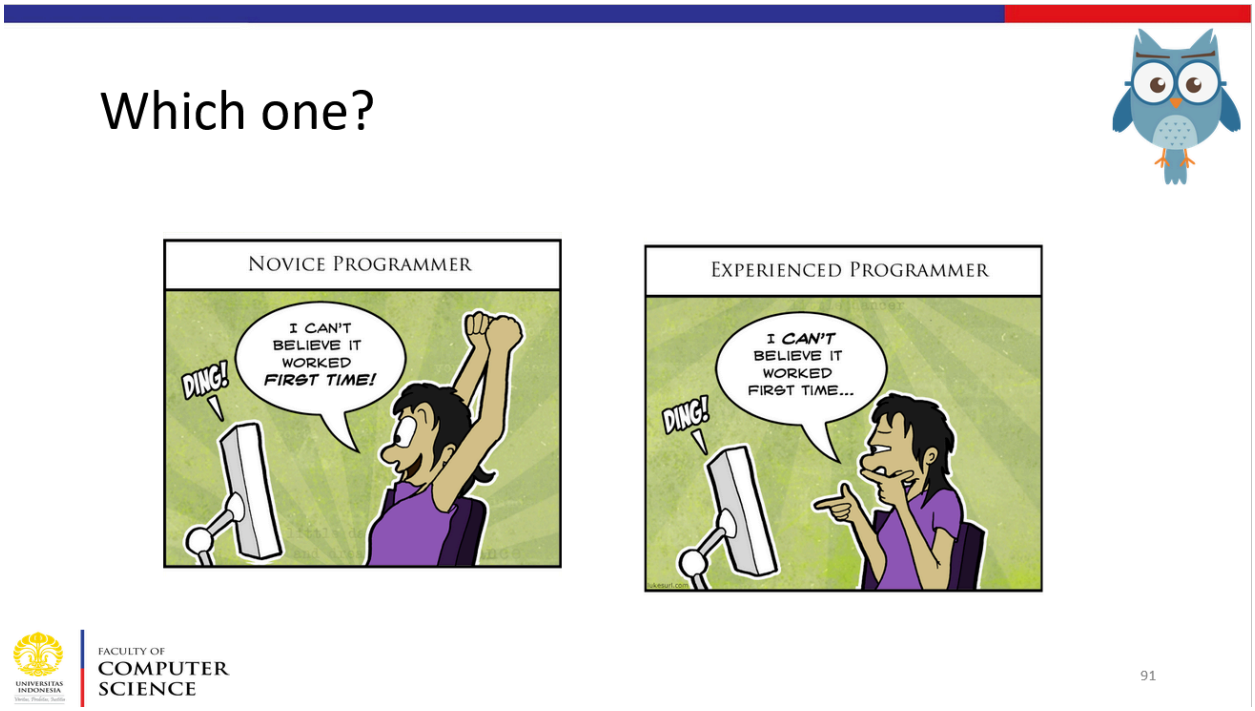
JUnit supports some annotations for configuring the test. For example:

1. JUnit uses annotation @Test to denote that a method is a test method.
2. Annotation @Before and @After are used to denote the setup and teardown methods



An example of JUnit for a simple Calculator is shown above. There are methods setUp and tearDown with annotations @Before and @After. In the example, we want to test methods add and subtract. The test method is annotated with @Test. Method testAdditon is used to test the addition, and method testSubtraction is to test the subtraction. For each test, there are steps to call the tested method and then verify the result using assertions.

Functional Test



Suppose that you recently finished a programming task and then tested your code. You found out that your code works perfectly. Based on your experience, which one of the comic panels above feels more related to you? Is it the left panel? Or the right one?

While it is pretty joyful to know that our code works, we need to be critical of the quality of our work. Try to remind yourself whether you have tested the code with other possible inputs, or see if you have implemented not only the normal path but also the exception handling.

Additionally, remember that software is created based on requirements given by a user. A user may have evolving needs over time, thus requiring the software to change. Can you make sure that your code still works fine after any changes made to the software? That is why testing is essential to help safeguard our software throughout the development process.

As software evolves, changes and additions may introduce new bugs or impact existing functionality. Regression tests are designed to catch these issues by retesting previously developed and working features whenever new changes are made. Should we do the regression test manually?

Functional Test

- Acceptance tests, or end-to-end tests.
- Ensure the software as a whole works
- Verify functionality from user's point-of-view
- Treat the software as a black-box
  - E.g. in GUI-based app, functional test operate by interacting with GUI elements

FACULTY OF COMPUTER SCIENCE

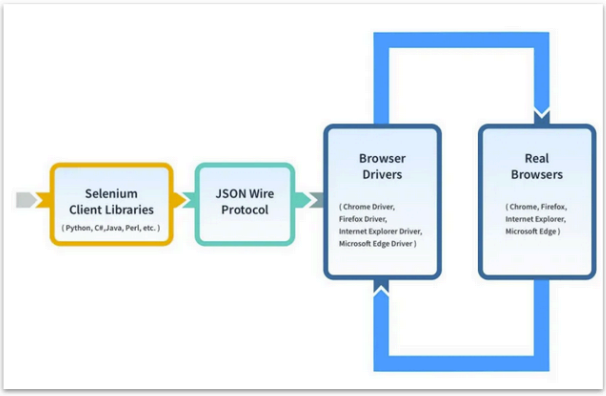
93

A functional test is a type of testing that focuses on verifying that the software functions according to specified requirements. It is often called an acceptance test or end-to-end test. It involves testing the individual functions or features of a system to ensure they meet the functional specifications.

Different from a unit test that verifies the functionality from the implementation, a functional test verifies functionality from the user's perspective. In software engineering, this test is categorized as black box testing.

Selenium & WebDriver API

- An API for automating **interactions** within a Web browser
  - Simulates a "user" interacting with the UI elements in a Web page
- Each Web browser provides their own WebDriver API
- Selenium is one of the most popular library for interfacing with the WebDriver API



Source: https://medium.com/thinkcru/automated-end-to-end-testing-with-selenium-and-cypress-io-19afda5c91ab, which was taken from Browserstack.com

94

We can use a tool support for functional testing, such as Selenium. Selenium is a popular open-source framework for automating web browsers. It provides a suite of tools for web application testing and supports various programming languages, including Java, Python, C#, Ruby, and others. Selenium allows testers and developers to write scripts in a programming language of their choice to automate interactions with web browsers.

Selenium WebDriver allows us to interact with web browsers programmatically. The WebDriver API is a collection of interfaces and classes that provide a programming interface for interacting with web browsers. WebDriver enables developers to write scripts to automate browser actions such as navigating to URLs, interacting with web elements (e.g., clicking buttons, filling forms), and extracting information from web pages.

Example: A Basic Functional Test in Java & Spring Boot

- Add Selenium-related dependencies into the project
- Configure the build system to be able to run unit tests and functional tests independently
  - We follow a convention where all functional tests have `FunctionalTest` as suffix in their Java class name

build.gradle.kts

```
val seleniumJavaVersion = "4.14.1"
val seleniumJupiterVersion = "5.0.1"
val webdrivermanagerVersion = "5.6.3"

dependencies {
    // Other dependencies omitted for brevity
    testImplementation("org.seleniumhq.selenium:selenium-java:$seleniumJavaVersion")
    testImplementation("io.github.bonigarcia:selenium-jupiter:$seleniumJupiterVersion")
    testImplementation("io.github.bonigarcia:webdrivermanager:$webdrivermanagerVersion")
}
```

build.gradle.kts

```
tasks.register<Test>("unitTest") {
    description = "Runs unit tests."
    group = "verification"

    filter {
        excludeTestsMatching("*FunctionalTest")
    }
}

tasks.register<Test>("functionalTest") {
    description = "Runs functional tests."
    group = "verification"

    filter {
        includeTestsMatching("*FunctionalTest")
    }
}
```

95

As an example, let's try to create a basic functional test in a Java & Spring Boot-based project. Suppose that the project is configured using the Gradle build system, which has the configuration located at build.gradle.kts file. To add Selenium-related dependencies, declare the required Selenium packages as `testImplementation` statements in `dependencies` scope. We use three Selenium-related packages:

1. selenium-java ➔ Selenium library written in Java that interfaces with a Web browser
2. selenium-jupiter ➔ JUnit 5 test runner extension for writing functional test in Selenium more convenient (e.g., automated browser-specific WebDriver injection into a test case method via parameter)
3. webdrivermanager ➔ A helper library for automatically managing browser-specific WebDriver during test execution

Example: A Basic Functional Test in Java & Spring Boot

- Create a new JUnit 5 test suite class with name suffixed with FunctionalTest
  - Example: `HomePageFunctionalTest`
- Add the required annotation(s) on the test suite class declaration
  - @SpringBootTest → runs the test suite on Spring Framework's test environment (or context)
  - @ExtendWith → provides additional capability to the JUnit 5 test runner

HomePageFunctionalTest.java

```
package id.ac.ui.cs.advprog.eshop.functional;

import io.github.bonigarcia.seljup.SeleniumJupiter;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.By;
import org.openqa.selenium.chrome.ChromeDriver;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.springframework.boot.test.context.SpringBootTest.WebEnvironment.RANDOM_PORT;

@SpringBootTest(webEnvironment = RANDOM_PORT)
@ExtendWith(SeleniumJupiter.class)
class HomePageFunctionalTest {
```

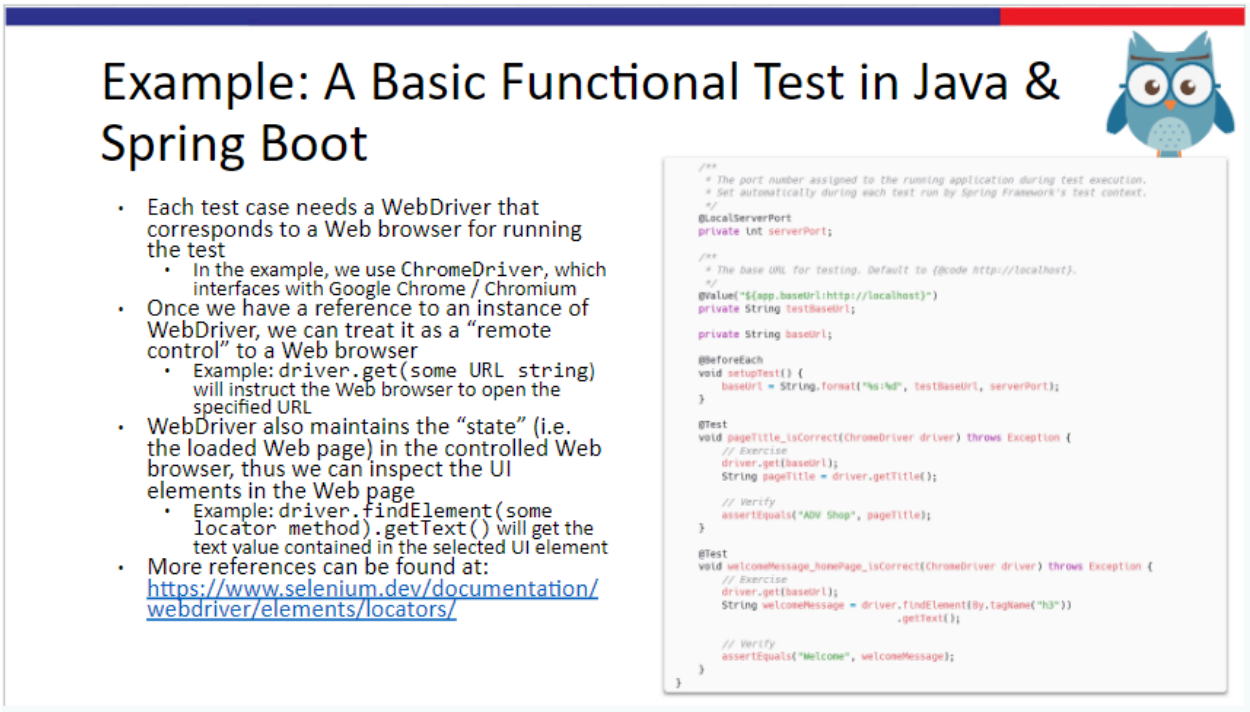
96

After finishing the setup, how to create a functional test using Selenium?

1. Create a new test class with a name suffixed with FunctionalTest, e.g., HomePageFunctionalTest
2. Add the required annotations on the test class declaration

   @SpringBootTest ➔ to run the test suite (integration) on Spring Framework's test environment

   @ExtendWith ➔ provides additional capability to the JUnit 5 test runner



Let's go into the details of functional tests, as shown in the example above

1. Choose the WebDriver based on your browser preference. In the example, we use `ChromeDriver`
2. The WebDriver is like a remote control to the web browser. We can ask the WebDriver to do some tasks in the web browser, such as opening a specific URL
3. WebDriver also maintains the state in the controlled web browser, so we can inspect the UI elements in the web page. For example, we can ask the driver to findElement(By.id("id")) or findElement(By.tagName("h3"))
   More information: https://www.selenium.dev/documentation/webdriver/elements/locators/

5. Tutorial & Exercise

We will use Spring Boot, a popular Java web framework, as an exercise. In this tutorial and exercise, there are several to-do checklists that you should achieve. The checklists are marked with blue font color.

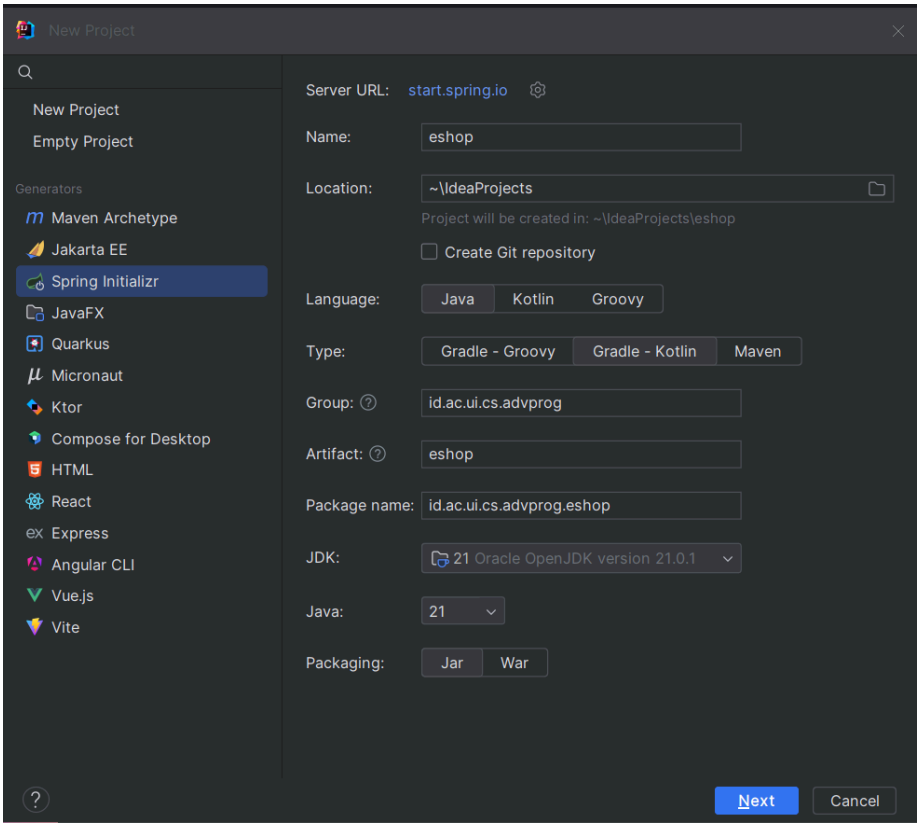
Preparation

Before starting the development process, please install the following environments:

1. **Java 21**. If you have installed Java, please check the installed version before continuing. You can check the version by running java -version in Command Prompt/Terminal. Make sure the version is 21.X.X (for example, 21.0.2).
2. **Git**
3. IDE **IntelliJ** Ultimate

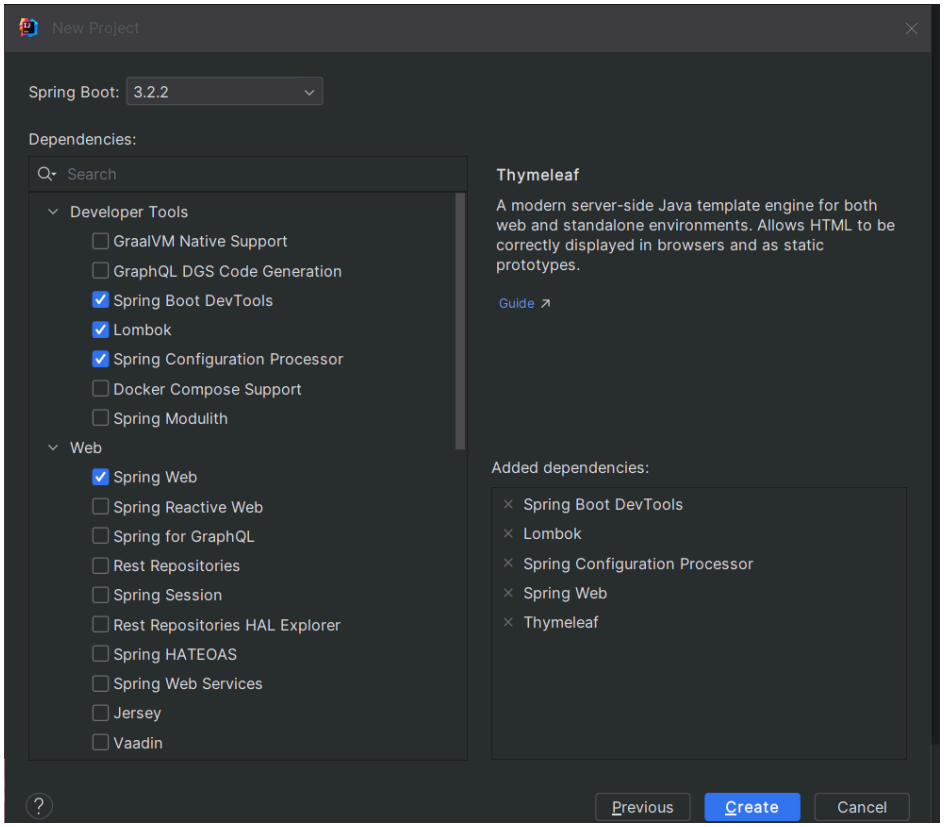
Create Spring Boot Project

1. Open IntelliJ Ultimate
2. Choose "**New Project**" Menu
3. Choose "**Spring Initializer**" and fill in the following properties.
   Project name: eshop | **Language**: Java | **Type**: Gradle Kotlin |
   Group: id.ac.ui.cs.advprog | **JDK and Java**: 21

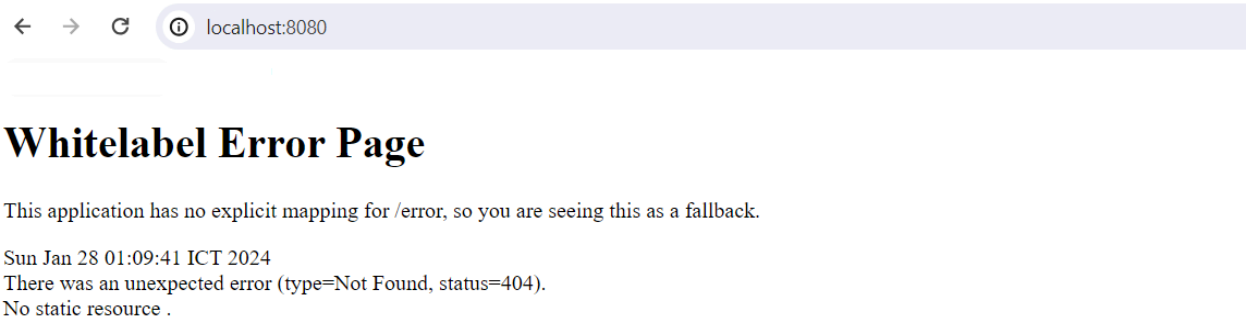


4. Choose "**Next**"

5. Choose Spring Boot **version 3.2.2**, then add the following **dependencies:** "Spring Boot DevTools", "Lombok", "Spring Configuration Processor", "Spring Web", dan "Thymeleaf".



6. Choose "**Create"** and wait for the process. IntelliJ will download the required dependencies, such as downloading Gradle (build tools). See the process on the bottom right of IntelliJ.
7. Once your project is ready, you can push the "Run" (triangle button) on the upper right of IntelliJ.
8. Open "localhost:8080" and you can see the following result.

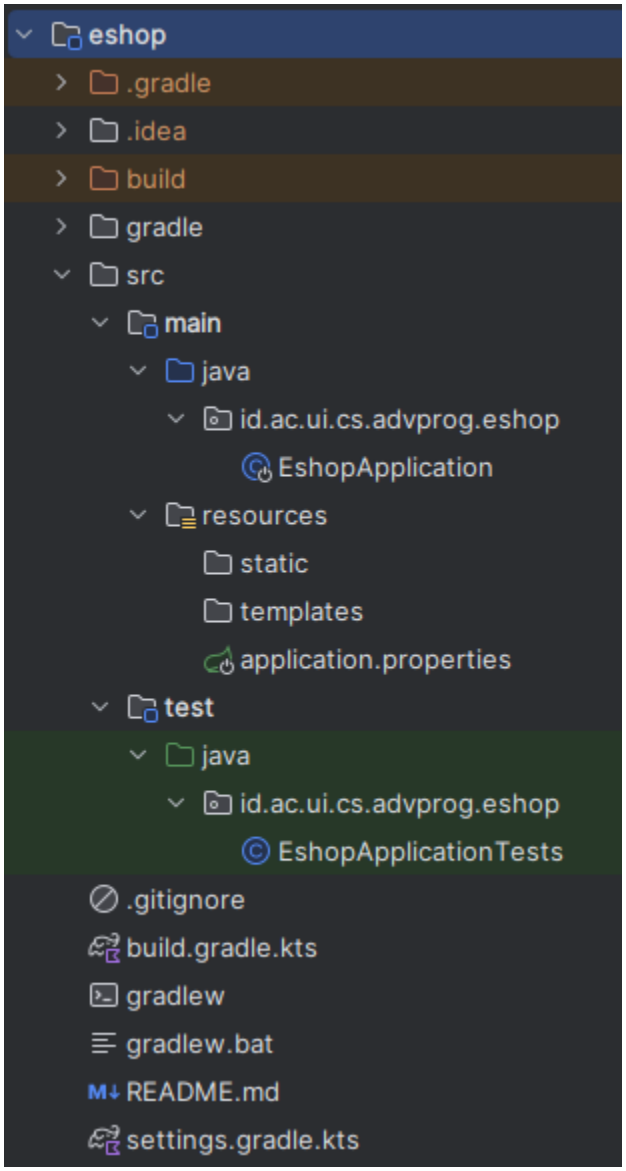


- [ ] Create a Git repository, commit, and push your source to your own Git Repository

Additional notes: you can also create a new Spring Boot project via https://start.spring.io/

Spring Boot Project Structure

The structure of the Spring Boot project is shown in the following picture.



As shown in the structure, there are several primary directories in the Spring Boot:

1. `src` directory contains the application source code. This directory is separated into `main` and `test`.

2. `main` directory is responsible to manage content and application logic. This directory is splitted into `java` and `resource`.

3. `java` directory consists of implementation of the business logic We can also employ MVC pattern in `java` directory.

4. `resource` directory manages the files that supports the application, such as static files (HTML, CSS, image) and application.properties.

5. `test` directory is used to manage the testing source code.

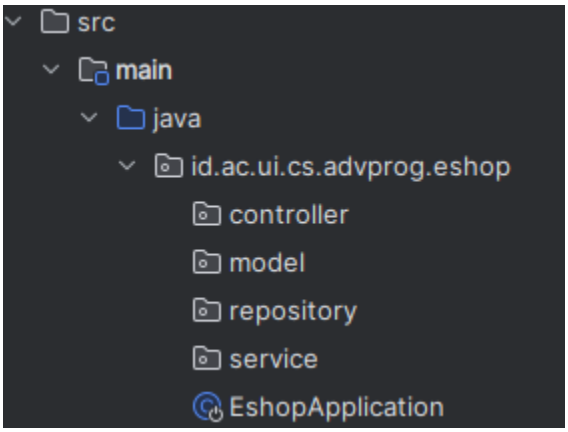
There are additional files related to Git (.gitignore) and Gradle (build.gradle, gradlew, gradlew.bat, settings.gradle). As we use Gradle as build automation tool, these files are required for building and running the application.

Additional notes:

In Django, you have learned about Model View Template (MVT) pattern. In Spring Boot, we use Model View Controller (MVC) pattern. Model in MVT is similar to the Model in MVC, View in MVT is similar to the Controller in MVC, and Template in MVT is similar to view in MVC.

Development

Assume that we want to create an EShop application that manages sales in ADV Shop. Before starting the development, create four packages inside the java directory. We split the application logic into four packages: **controller, model, repository,** and **service**. The model layer is used to represent the problem domain (data). The controller manages requests and responses and forwards the request to the service layer. The service layer implements the business logic and it might call the repository layer to access the database.



At the first iteration, you have to display a list of products of ADV Shop. We will implement features to add a new product and then display the products.

- [ ] Create a new branch list-product in your Git repository and move to this branch

1. **Model.** The first thing to do is model the problem domain. A product has three main attributes: id, name, and quantity. Create **Product.java** in the model package.

```java
package id.ac.ui.cs.advprog.eshop.model;

import lombok.Getter;
import lombok.Setter;

@Getter @Setter
public class Product {
    private String productId;
    private String productName;
    private int productQuantity;
}
```

- [ ] Commit Product.java to your Git repository

2. **Repository.** After creating the model layer, create **ProductRepository.java** in the repository package that manages the data operations. At this stage, we do not connect to the database system yet. We use a simple data structure to create and display the product.

```java
package id.ac.ui.cs.advprog.eshop.repository;

import id.ac.ui.cs.advprog.eshop.model.Product;
import org.springframework.stereotype.Repository;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

@Repository
public class ProductRepository {
    private List<Product> productData = new ArrayList<>();

    public Product create(Product product) {
        productData.add(product);
        return product;
    }

    public Iterator<Product> findAll() {
        return productData.iterator();
    }
}
```

- [ ] Commit ProductRepository.java to your Git repository

3. **Service**. Next, create **ProductService.java** in the service package as a logic implementation.

```java
package id.ac.ui.cs.advprog.eshop.service;

import id.ac.ui.cs.advprog.eshop.model.Product;
import java.util.List;

public interface ProductService {
    public Product create(Product product);
    public List<Product> findAll();
}
```

Then, create **ProductServiceImpl.java** in the same package.

```java
package id.ac.ui.cs.advprog.eshop.service;

import id.ac.ui.cs.advprog.eshop.model.Product;
import id.ac.ui.cs.advprog.eshop.repository.ProductRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

@Service
public class ProductServiceImpl implements ProductService {

    @Autowired
    private ProductRepository productRepository;

    @Override
    public Product create(Product product) {
        productRepository.create(product);
        return product;
    }


    @Override
    public List<Product> findAll() {
        Iterator<Product> productIterator = productRepository.findAll();
        List<Product> allProduct = new ArrayList<>();
        productIterator.forEachRemaining(allProduct::add);
        return allProduct;
    }
}
```

☐ Commit ProductService.java and ProductServiceImpl.java to your Git repository

4. **Controller**. The last part, create **ProductController.java** in the controller package. The controller receives requests from the client and then maps the request to the service layer.

```java
package id.ac.ui.cs.advprog.eshop.controller;

import id.ac.ui.cs.advprog.eshop.model.Product;
import id.ac.ui.cs.advprog.eshop.service.ProductService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@Controller
@RequestMapping("/product")
public class ProductController {

    @Autowired
    private ProductService service;

    @GetMapping("/create")
    public String createProductPage(Model model) {
        Product product = new Product();
        model.addAttribute( attributeName: "product", product);
        return "createProduct";
    }

    @PostMapping("/create")
    public String createProductPost(@ModelAttribute Product product, Model model) {
        service.create(product);
        return "redirect:list";
    }

    @GetMapping("/list")
    public String productListPage(Model model) {
        List<Product> allProducts = service.findAll();
        model.addAttribute( attributeName: "products", allProducts);
        return "productList";
    }
}
```

- [ ] Commit ProductController.java to your Git repository

5. **Templates**

   We just finished creating the business logic. Now, prepare the HTML to display the result to the user. Create two HTML files, first to create a new product and second to display the list of products.

   Copy and paste the following code to the template directory:

   CreateProduct.html: https://pastecode.io/s/it2wam3i
   ProductList.html: https://pastecode.io/s/z9s80o5c

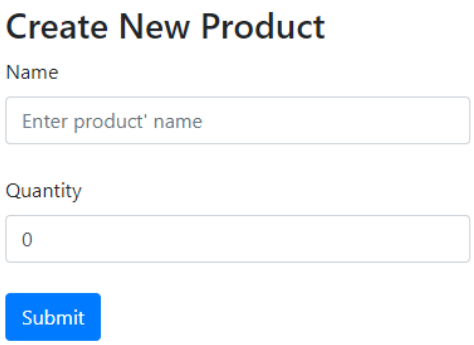
   You may use your own HTML pages.

   - [ ] Commit HTML templates to your Git repository

Run Application

1. Press Shift+F10 or push the “Run” (triangle button) on the upper right of IntelliJ to run your application.
2. Open http://localhost:8080/product/list

3. Click **Create Product** button and it redirects to http://localhost:8080/product/create



4. Create a new product and click **Submit.** The list of products is updated with a new product.

Exercise 1

- [ ] Push all your (committed) source code on **list-product** branch!
- [ ] If everything goes well, merge **list-product** branch to **main!**
- [ ] Create a new branch **edit-product** and **delete-product,** and implement new features **Edit and Delete Product.**
- [ ] Commit and Push your work to branch **edit-product** and **delete-product**

Reflection 1

You already implemented two new features using Spring Boot. Check again your source code and evaluate the coding standards that you have learned in this module. Write clean code principles and secure coding practices that have been applied to your code. If you find any mistake in your source code, please explain how to improve your code. **Please write your reflection inside the repository's README.md file.**

Testing in Spring Boot

Preparation

Before creating unit tests and functional tests, add dependencies to unit test and functional to Gradle build script (file `build.gradle.kts`)

1. Add the following variables to file `build.gradle.kts`:

```
val seleniumJavaVersion = "4.14.1"
val seleniumJupiterVersion = "5.0.1"
val webdrivermanagerVersion = "5.6.3"
val junitJupiterVersion = "5.9.1"
```

2. Complete the dependencies by adding the following requirements in **dependencies** part:

```
    testImplementation("org.seleniumhq.selenium:selenium-java:$seleniumJavaVersion")
    testImplementation("io.github.bonigarcia:selenium-jupiter:$seleniumJupiterVersion")
    testImplementation("io.github.bonigarcia:webdrivermanager:$webdrivermanagerVersion")
    testImplementation("org.junit.jupiter:junit-jupiter-api:$junitJupiterVersion")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:$junitJupiterVersion")
```

3. Add the following codes after **dependencies** part

```
tasks.register<Test>( name: "unitTest") { this: Test
    description = "Runs unit tests."
    group = "verification"

    filter { this: TestFilter
        excludeTestsMatching("*FunctionalTest")
    }
}

tasks.register<Test>( name: "functionalTest") { this: Test
    description = "Runs functional tests."
    group = "verification"

    filter { this: TestFilter
        includeTestsMatching("*FunctionalTest")
    }
}
```

4. Replace the last part with the following code:

```
62    tasks.withType<Test>().configureEach { this: Test
63        useJUnitPlatform()
64    }
```

- [ ] Commit and push your source code to your Git repository (main branch)

Unit Test

We will add unit tests to our eShop project. At this stage, we create Unit Tests to check the implementation in Model and Repository.

- [ ] Create a new branch unit-test and checkout to this branch

1. **Test Model**. Create a new file **ProductTest.java** in `src/test/java/id/ac/ui/cs/advprog/eshop/model/`

```
1   package id.ac.ui.cs.advprog.eshop.model;
2
3   import org.junit.jupiter.api.BeforeEach;
4   import org.junit.jupiter.api.Test;
5
6   import static org.junit.jupiter.api.Assertions.*;
7   class ProductTest {
        no usages
8       Product product;
9       @BeforeEach
10      void setUp() {
11          this.product = new Product();
12          this.product.setProductId("eb558e9f-1c39-460e-8860-71af6af63bd6");
13          this.product.setProductName("Sampo Cap Bambang");
14          this.product.setProductQuantity(100);
15      }
16      @Test
17      void testGetProductId() {
18          assertEquals( expected: "eb558e9f-1c39-460e-8860-71af6af63bd6", this.product.getProductId());
19      }
20
21      @Test
22      void testGetProductName() {
23          assertEquals( expected: "Sampo Cap Bambang", this.product.getProductName());
24      }
25
26      @Test
27      void testGetProductQuantity() {
28          assertEquals( expected: 100, this.product.getProductQuantity());
29      }
30  }
```

- [ ] Commit ProductTest.java to your Git repository

There are three tests in ProductTest.java to verify the model Product.java. All of the tests verify the *getter* methods. These unit tests only check the positive scenario. Would you like to add the negative scenario?

Check more about various kinds of assertions in JUnit in the documentation: https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html

2. **Test Repository**. Create a new file ProductRepositoryTest.java in `src/test/java/id/ac/ui/cs/advprog/eshop/repository/`

```java
package id.ac.ui.cs.advprog.eshop.repository;

import id.ac.ui.cs.advprog.eshop.model.Product;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.mockito.InjectMocks;
import org.mockito.junit.jupiter.MockitoExtension;
import org.springframework.boot.test.mock.mockito.MockBean;

import java.util.Iterator;

import static org.junit.jupiter.api.Assertions.*;

@ExtendWith(MockitoExtension.class)
class ProductRepositoryTest {

    6 usages
    @InjectMocks
    ProductRepository productRepository;
    @BeforeEach
    void setUp() {
    }
    @Test
    void testCreateAndFind() {
        Product product = new Product();
        product.setProductId("eb558e9f-1c39-460e-8860-71af6af63bd6");
        product.setProductName("Sampo Cap Bambang");
        product.setProductQuantity(100);
        productRepository.create(product);

        Iterator<Product> productIterator = productRepository.findAll();
        assertTrue(productIterator.hasNext());
        Product savedProduct = productIterator.next();
        assertEquals(product.getProductId(), savedProduct.getProductId());
        assertEquals(product.getProductName(), savedProduct.getProductName());
        assertEquals(product.getProductQuantity(), savedProduct.getProductQuantity());
    }

```

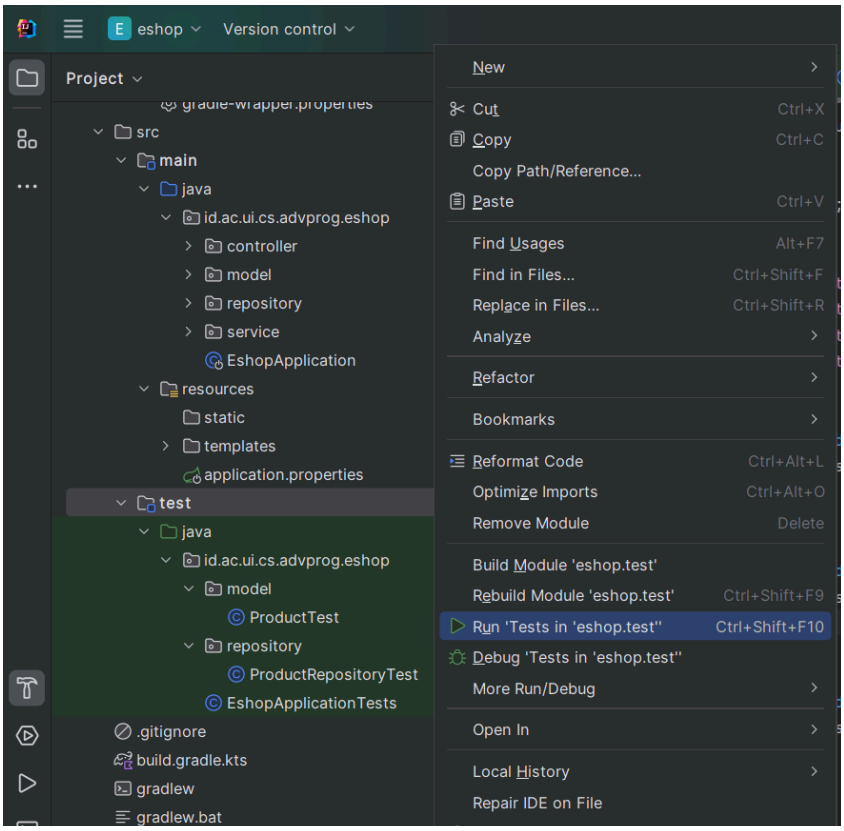
Con’t

```
    @Test
    void testFindAllIfEmpty() {
        Iterator<Product> productIterator = productRepository.findAll();
        assertFalse(productIterator.hasNext());
    }
    @Test
    void testFindAllIfMoreThanOneProduct() {
        Product product1 = new Product();
        product1.setProductId("eb558e9f-1c39-460e-8860-71af6af63bd6");
        product1.setProductName("Sampo Cap Bambang");
        product1.setProductQuantity(100);
        productRepository.create(product1);

        Product product2 = new Product();
        product2.setProductId("a0f9de46-90b1-437d-a0bf-d0821dde9096");
        product2.setProductName("Sampo Cap Usep");
        product2.setProductQuantity(50);
        productRepository.create(product2);

        Iterator<Product> productIterator = productRepository.findAll();
        assertTrue(productIterator.hasNext());
        Product savedProduct = productIterator.next();
        assertEquals(product1.getProductId(), savedProduct.getProductId());
        savedProduct = productIterator.next();
        assertEquals(product2.getProductId(), savedProduct.getProductId());
        assertFalse(productIterator.hasNext());
    }
}
```

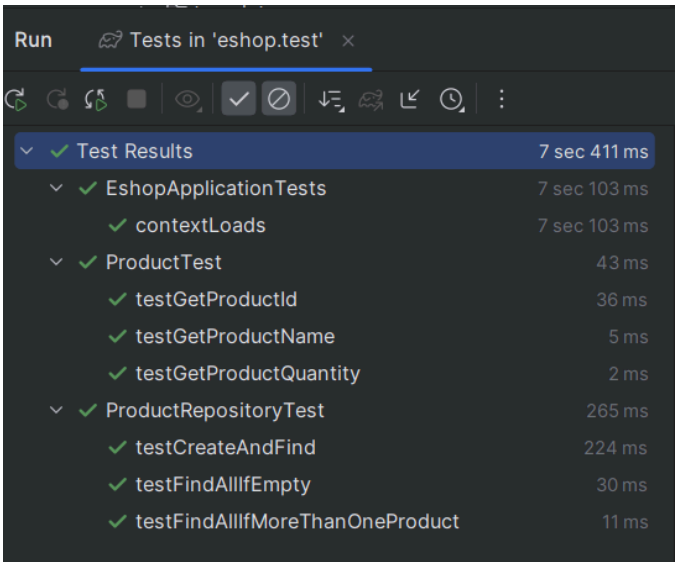
- [ ] Commit ProductRepositoryTest.java into your Git repository

3. **Run your test**

   Right-click on the test directory, and select **Run Tests in “eshop test”**. As an alternative, press `Ctrl+Shift+F10`



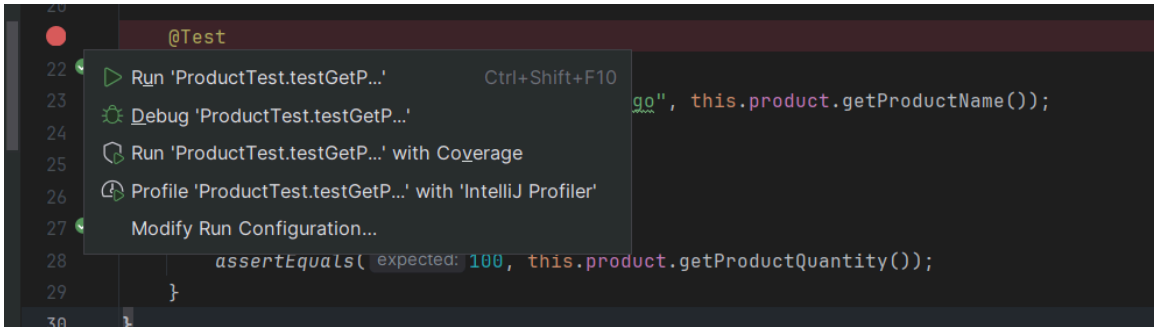
You can see the result at the bottom left panel in IntelliJ



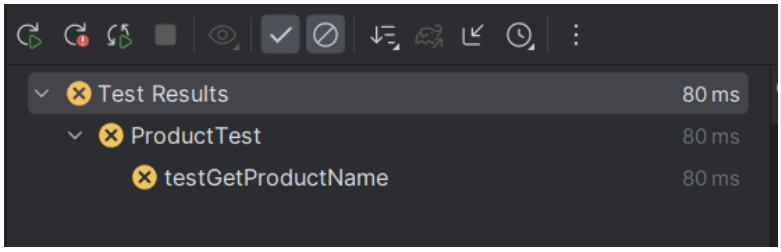
As we can see on the test report, all of the tests are passed. Check again your source code ProductTest.java, there is a mark in the line of the unit test. The green checklist (denotes the successful test.

```
21      @Test
22      void testGetProductName() {
23          assertEquals( expected: "Sampo Cap Bambang", this.product.getProductName());
24      }
25
```

4. Let's try to make the failed test. Change the name "Sampo Cap Bambang" in Line 23 to "Sampo Cap Bango". If you only change one method, you can run the changed method. Right-click the checklist (next to line number) and then choose "Run ProductTest.testGetProductName"

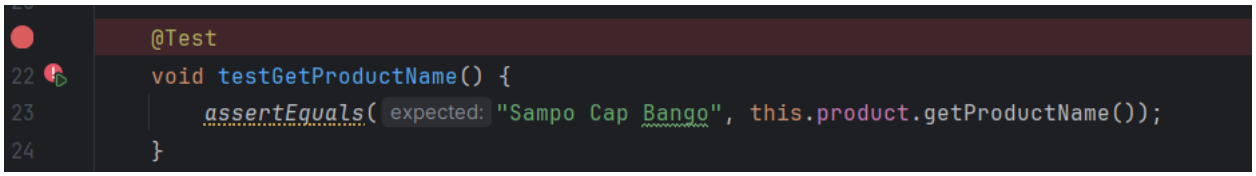


As expected the test fails, and you can see the report in IntelliJ




```
> Task :test FAILED
ProductTest > testGetProductName() FAILED
    org.opentest4j.AssertionFailedError at ProductTest.java:23
1 test completed, 1 failed
FAILURE: Build failed with an exception.
```

Check your source code again, after executing the failed test, there is a mark red warning next to the line number.



Change back your test case to “Sampo Cap Bambang” and don’t forget to rerun the test.

Functional Test

Let's try writing a basic functional test that verifies the **title** of the home page and the initial **welcome message**. You do not have to include any dependencies such as Selenium since they have been added in previous tasks.

- [ ] Create a new branch functional-test and checkout to this branch

1. Create a new functional test suite as a Java class named `HomePageFunctionalTest` at `id.ac.ui.cs.advprog.eshop.functional` Java package. Make sure the class declaration, annotations, and imported packages match with the following screenshot:

HomePageFunctionalTest.java

```java
package id.ac.ui.cs.advprog.eshop.functional;

import io.github.bonigarcia.seljup.SeleniumJupiter;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.openqa.selenium.By;
import org.openqa.selenium.chrome.ChromeDriver;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.web.server.LocalServerPort;

import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.springframework.boot.test.context.SpringBootTest.WebEnvironment.RANDOM_PORT;

@SpringBootTest(webEnvironment = RANDOM_PORT)
@ExtendWith(SeleniumJupiter.class)
class HomePageFunctionalTest {
```

2. Fill in the body of `HomePageFunctionalTest` with content as follows:

```
    /**
     * The port number assigned to the running application during test execution.
     * Set automatically during each test run by Spring Framework's test context.
     */
    @LocalServerPort
    private int serverPort;

    /**
     * The base URL for testing. Default to {@code http://localhost}.
     */
    @Value("${app.baseUrl:http://localhost}")
    private String testBaseUrl;

    private String baseUrl;

    @BeforeEach
    void setupTest() {
        baseUrl = String.format("%s:%d", testBaseUrl, serverPort);
    }

    @Test
    void pageTitle_isCorrect(ChromeDriver driver) throws Exception {
        // Exercise
        driver.get(baseUrl);
        String pageTitle = driver.getTitle();

        // Verify
        assertEquals("ADV Shop", pageTitle);
    }

    @Test
    void welcomeMessage_homePage_isCorrect(ChromeDriver driver) throws Exception {
        // Exercise
        driver.get(baseUrl);
        String welcomeMessage = driver.findElement(By.tagName("h3"))
                                      .getText();

        // Verify
        assertEquals("Welcome", welcomeMessage);
    }
}
```

Remember the test case writing convention expected by JUnit 5. Every method that represents a test case must be annotated with @Test in order to make it executable by JUnit 5's test runner. Each test case should also perform procedures to exercise the actual function to be tested and verify the result with the expected output.

3. Look at the pageTitle_isCorrect() test case. It asks the browser (Google Chrome/Chromium) via WebDriver API to open the URL of the running application. Then, it queries the page title (i.e. the value at <title> tag) and verifies if the title is the same as the expected title (i.e., "ADV Shop").

   Before you proceed to write the next test case, run the test case locally by running it from the IDE (IntelliJ) or from the terminal/shell (./gradlew functionalTest). You will see the Web browser window briefly shown opening the locally running application. If the test passes,

it means the Web browser controlled by WebDriver was able to verify the page title is the same as the expected title. Otherwise, make sure the title of the home page is “ADV Shop”.

4. Now look at the welcomeMessage_homePage_isCorrect test case. It behaves similarly to the previous test case. However, it performs a locator method to get a reference to a <h3> element in the HTML page and obtain the text value from the element. Once it obtained the text value, it checks whether the value is the same as the expected value, i.e. “Welcome”. To see if the test works, try to run it from the IDE or terminal using the same command mentioned in the previous task.

- [ ] Save your work as a new Git commit into the functional-test branch.

Exercise 2

- [ ] Push the unit-test branch to your online Git repository
- [ ] Create a new unit test to verify features **Edit** and **Delete** Product. Don't forget to test positive and negative scenarios.
    - [ ] Do not forget to save your result as a new Git commit in unit-test branch and push the latest commit to your online Git repository
- [ ] Push the functional-test branch to your online Git repository
- [ ] Write a new functional test suite Java class named CreateProductFunctionalTest.java that verifies Create Product feature when performed by the user. In other words, you need to implement the functional test to simulate **user interactions** when a user tries to add a product and ensure that they can see the new product in the product list. You can refer to Selenium documentation about "Interacting with web elements" (link: https://www.selenium.dev/documentation/webdriver/elements/interactions/) to see some examples and documentation about the methods you can use in completing this task.
    - [ ] Do not forget to save your result as a new Git commit in functional-test branch and push the latest commit to the online Git repository

Reflection 2

1. After writing the unit test, how do you feel? How many unit tests should be made in a class? How to make sure that our unit tests are enough to verify our program? It would be good if you learned about code coverage. *Code coverage is a metric that can help you understand how much of your source is tested*. If you have 100% code coverage, does that mean your code has no bugs or errors?
2. Suppose that after writing the CreateProductFunctionalTest.java along with the corresponding test case, you were asked to create another functional test suite that verifies the **number of items in the product list**. You decided to create a new Java class similar to the prior functional test suites with **the same setup procedures and instance variables**.

   What do you think about the cleanliness of the code of the new functional test suite? Will the new code reduce the code quality? Identify the potential clean code issues, explain the reasons, and suggest possible improvements to make the code cleaner! **Please write your reflection inside the repository's README.md file.**