**day-2-git-and-github**

**Assignment**

**1**

**Version control.** is a system that tracks changes made to files over time, enabling developers to manage different versions of their projects. This is vital in software development, as it helps maintain project integrity and prevents data loss.

**Key Concepts in Version Control:**

Repository: A central location where all project files and their change history are stored.

**Commit:** A snapshot of the project at a specific moment, typically accompanied by a descriptive message.

**Branch:** A separate line of development that allows developers to work on different features or fixes without interference.

**Merge:** The process of integrating changes from one branch into another.

**GitHub.** Is a widely used cloud-based platform that utilizes Git, a leading version control system? It provides a user-friendly interface for managing repositories, collaborating with others, and tracking project history.

**Benefits of Version Control:**

Collaboration: Multiple developers can work on the same project simultaneously, minimizing the risk of overwriting each other's contributions.

Backup: The repository stores project history, acting as a safeguard against mistakes or data loss.

Change Tracking: Version control keeps a detailed record of changes, which helps debug and understand the project’s evolution.

Experimentation: Developers can create branches to test new features without affecting the main codebase.

Code Review: Platforms like GitHub streamline the code review process, promoting quality and consistency before integrating changes.

**2**

Setting up a new repository on GitHub is a straightforward process that involves several key steps. Here’s a detailed guide on how to do it, along with some important decisions you’ll need to make along the way.

Steps to Set Up a New Repository on GitHub:

1. Sign In to GitHub:

- If you don’t have an account, create one at [github.com](https://github.com). If you already have an account, simply log in.

2.Create a New Repository:

- Click the "+" icon in the upper right corner of the GitHub homepage and select "New repository."

3. Fill in Repository Details:

- Repository Name: Choose a concise and descriptive name for your repository. It should reflect the purpose of the project.

- Description (optional): Provide a brief overview of what the repository is about. This helps others understand the project at a glance.

4. Choose the Repository Visibility:

- Public: Anyone can see this repository. It’s a good choice for open-source projects.

- Private: Only you and selected collaborators can access this repository. This is ideal for personal projects or sensitive work.

5. Initialize the Repository (Optional):

- Add a README file: A README provides important information about your project. It’s often the first file users will see.

- Add a .gitignore file: This file tells Git which files or directories to ignore in the repository (e.g., temporary files or build artifacts).

- Choose a license: If you’re making your project public, select a license to clarify how others can use your code. Popular options include MIT, Apache 2.0, and GPL.

6. Create the Repository:

- Click the "Create repository" button to finalize the setup.

**Important Decisions to Make:**

- Repository Name: Ensure it is unique and meaningful, as this will be the primary identifier for your project.

- Visibility: Consider whether your project should be public or private based on your goals and audience. If you plan to collaborate with others openly, the public may be better. For sensitive or personal projects, choose private.

- README and .gitignore: think about the initial setup. A README is critical for onboarding new users or collaborators. A .gitign,ore is important for keeping your repository clean and focused.

- License: If you want others to use your work, pick an appropriate license. This can affect how your project is shared and modified in the future.

**3**

README file is a critical component of any GitHub repository. It serves as the primary source of information for anyone interacting with your project, whether they are collaborators, users, or contributors. A well-crafted README can greatly enhance the usability and accessibility of your repository.

**Importance of the README File**

1. First Impressions: The README is often the first file users see when they visit a repository. A clear and informative README can create a positive initial impression and encourage others to explore further.

2. Guidance for Users: It provides essential information about the project, including its purpose, functionality, and how to use it. This is particularly important for open-source projects, where new users may be unfamiliar with the codebase.

3. Collaboration Tool: A comprehensive README can help streamline collaboration by outlining how others can contribute, the project's workflow, and any guidelines or standards that should be followed.

4. Documentation: It serves as a form of documentation, summarizing important details that users and developers may need to reference. This can reduce confusion and help maintain consistency within the project.

**Key Elements of a Well-Written README**

1. Project Title: Start with the name of your project, prominently displayed at the top.

2. Description: Provide a brief overview of what the project does, its purpose, and the problems it aims to solve.

3. Installation Instructions: Clearly outline the steps needed to install and set up the project. Include prerequisites and dependencies to ensure users can easily get started.

4. Usage Examples: Offer practical examples demonstrating how to use the project. Code snippets or screenshots can be particularly helpful here.

5. Contributing Guidelines: Detail how others can contribute to the project. Include information about submitting issues, pull requests, and any coding standards to follow.

6. License Information: Clearly state the license under which the project is released. This informs users of their rights regarding usage, distribution, and modification.

7. Contact Information: Provide details on how to reach you or the project maintainers for questions or support.

8. Acknowledgments: If applicable, recognize any contributors or resources that played a significant role in the project.

**Contribution to Effective Collaboration**

A well-structured README enhances collaboration in several ways:

- Clarity: By clearly stating the project’s goals and how to contribute, it minimizes misunderstandings among collaborators.

- Onboarding: New contributors can quickly get up to speed with the project, reducing the learning curve and fostering a more inclusive environment.

- Consistency: When guidelines for contributing are explicitly outlined, it helps maintain a consistent approach across the codebase, making collaboration smoother.

- Engagement: A comprehensive README encourages community involvement. Users who feel informed and welcomed are more likely to contribute.

**4**

When using GitHub, understanding the differences between public and privaterepositories is essential for determining how to share and manage your projects. Each type has its own advantages and disadvantages, especially when it comes to collaboration.

**Public Repository**

Definition: A public repository is accessible to anyone on the internet. Anyone can view, clone, and contribute to the repository.

**Advantages:**

1. Visibility and Reach: Public repositories allow you to showcase your work to a broader audience. This can be beneficial for personal branding, attracting potential collaborators, or demonstrating your skills to employers.

2. Community Contributions: Since anyone can access and contribute, public repositories can benefit from a diverse range of inputs, which can enhance the quality and features of the project.

3. Open Source Collaboration: They foster an open-source culture, encouraging transparency and shared learning within the developer community.

4. Ease of Sharing: Sharing links to a public repository is straightforward, facilitating easier collaboration and communication among developers.

**Disadvantages:**

1. Lack of Control: You may have less control over who contributes and how, which can lead to issues with quality or relevance of contributions.

2. Exposure to Criticism: Public repositories are visible to everyone, which can result in negative feedback or scrutiny from others.

3. Security Risks: Sensitive information should never be stored in public repositories, as it is accessible to anyone.

**Private Repository**

Definition: A private repository is restricted to selected users. Only those granted access can view or contribute to the repository.

**Advantages:**

1. Enhanced Security: Private repositories protect sensitive information and intellectual property since access is limited to trusted collaborators.

2. Control Over Contributions: You can manage who has access to the repository, allowing you to curate contributions and maintain quality standards more effectively.

3. Focus on Internal Collaboration: Ideal for teams working on proprietary projects, private repositories facilitate focused collaboration without outside interference.

4. No External Scrutiny: Developers can work on their projects without the fear of public criticism, allowing for more experimentation and innovation.

**Disadvantages:**

1. Limited Exposure: With a private repository, your work is not visible to the public, which can limit opportunities for recognition and collaboration from the broader community.
2. Reduced Community Contributions: Since access is restricted, you miss out on potential input from outside developers, which can enhance the project’s development.
3. Resource Constraints: Private repositories on GitHub may come with limitations on the number of collaborators, depending on the subscription plan.

**5**

Making your first commit to a GitHub repository is a fundamental step in using Git for version control. Here’s a detailed guide on how to do it, along with an explanation of what commits are and their importance in managing your project.

**Steps to Make Your First Commit**

1. **Set Up Git:**

Install Git on your computer if you haven’t already. You can download it from git-scm.com.

Configure your Git username and email, which will be associated with your commits

git config --global user.name "Your Name"

git config --global user.email [you@example.com](mailto:you@example.com)

1. **Create a Local Repository:**

Navigate to the folder where you want your project to reside

mkdir my-project

cd my-project

Initialize a new Git repository

git init

1. **Add Files:**

Create or add files to your project directory. For example, you can create a README file

echo "# My Project" > README.md

1. **Stage Changes:**

Stage the files you want to include in your commit using the git add command:

git add README.md

You can stage all files in the directory by using git add ..

1. **Make Your First Commit:**

Commit the staged changes with a descriptive message:

git commit -m "Initial commit"

1. **Push to GitHub:**

If you have a remote repository set up on GitHub, you can link your local repository to it:

git remote add origin <https://github.com/username/my-project.git>

Push your commit to GitHub:

git push -u origin master

**Understanding Commits**

What Are Commits? Commits are snapshots of your project at a specific point in time. Each commit records the current state of your files along with a message that describes what changes were made. This creates a historical record of your project’s development.

**How Do Commits Help?**

**Tracking Changes:**

Commits allow you to see a complete history of changes made to your project. You can view what was changed, who made the change, and when it was made.

**Version Management:**

With commits, you can easily revert to previous versions of your project if something goes wrong or if a new feature doesn’t work as expected.

**Collaboration:**

In a collaborative environment, commits enable multiple developers to work on the same project simultaneously. Each developer can make their own commits, and the history remains clear and organized.

**Debugging:**

If a bug is introduced, you can use the commit history to identify when the problematic change occurred, making it easier to troubleshoot issues.

**Documentation:**

The commit messages provide context for changes, which can serve as documentation for future reference. Good commit messages help collaborators understand the purpose behind changes.

**6**

Branching is a core feature of Git that allows developers to create independent lines of development within a repository. This is particularly important for collaborative development, as it enables teams to work on different features or fixes without interfering with each other’s work. Here’s a detailed look at how branching works, its significance, and the typical workflow involved in creating, using, and merging branches.

**How Branching Works in Git**

When you create a branch in Git, you essentially make a copy of the code at a specific point in time. This allows you to experiment, develop new features, or make fixes independently from the main codebase (often referred to as the main or master branch). Each branch maintains its own history of commits, making it easy to work in isolation.

**Importance of Branching for Collaborative Development**

1. **Isolation of Features:**
   * Branches allow developers to work on new features or bug fixes without affecting the stability of the main codebase.
2. **Parallel Development:**
   * Multiple team members can work on different branches simultaneously, enabling faster development and integration of new features.
3. **Easy Integration:**
   * Once a feature is complete and tested, it can be merged back into the main branch, ensuring that the main code remains stable.
4. **Simplified Experimentation:**
   * Developers can create branches to experiment with new ideas. If the experiment doesn’t work out, the branch can simply be deleted without impacting the main project.

**Typical Workflow for Branching**

**1. Creating a Branch**

Togit checkout -b feature-branch

This command creates a new branch called feature-branch and switches to it immediately.

**2. Using the Branch**

While on the new branch, you can make changes to your code. After making your changes, you will need to stage and commit them.

git add .

git commit -m "Add new feature"

You can continue to make more changes, stage, and commit them as needed.

**3. Merging the Branch**

Once you’ve completed your work on the feature branch, the next step is to merge it back into the main branch. First, switch back to the main branch:

git checkout main

Next, merge the feature branch:

git merge feature-branch

This command integrates the changes from feature-branch into the main branch. If there are any conflicts (i.e., changes in the same lines of code on both branches), Git will prompt you to resolve them.

**4. Deleting the Branch**

After merging, if you no longer need the feature branch, you can delete it:

git branch -d feature-branch

This keeps your branch list clean and organized.

create a new branch, you use the following command:

**7**

Pull requests (PRs) are a vital aspect of the GitHub workflow, serving as a bridge between collaborative coding and integrating changes into a project. They facilitate code review, discussion, and collaboration among team members. Here’s an exploration of their role, benefits, and the typical steps involved in creating and merging a pull request.

**Role of Pull Requests in the GitHub Workflow**

1. **Code Review:**
   * Pull requests allow team members to review changes before they are merged into the main codebase. This process helps identify potential issues, improve code quality, and ensure adherence to coding standards.
2. **Discussion Platform:**
   * PRs provide a dedicated space for discussion about the proposed changes. Team members can leave comments, ask questions, and suggest modifications, fostering collaborative communication.
3. **Integration Testing:**
   * Many teams set up automated tests to run when a pull request is created. This helps ensure that new changes do not introduce bugs or break existing functionality.
4. **Documentation of Changes:**
   * Each pull request includes a description of the changes made, providing context for reviewers and future reference. This documentation is valuable for understanding the evolution of the project.

**Typical Steps Involved in Creating and Merging a Pull Request**

**1. Create a Feature Branch**

Before making changes, create a feature branch off the main branch:

git checkout -b feature-branch

**2. Make Changes and Commit**

Work on your feature, then stage and commit your changes:

git add .

git commit -m "Implement new feature"

**3. Push the Branch to GitHub**

Once your changes are committed locally, push the branch to the remote repository.

git push origin feature-branch

**4. Create a Pull Request**

1. **Navigate to the Repository:**
   * Go to your repository on GitHub.
2. **Open the Pull Requests Tab:**
   * Click on the "Pull requests" tab.
3. **Create a New Pull Request:**
   * Click on the "New pull request" button. GitHub will show a comparison of the base branch (usually main) and your feature branch.
4. **Fill Out the PR Form:**
   * Provide a title and a detailed description of the changes you made. Include information about the purpose of the changes, any issues addressed, and any additional context for reviewers.
5. **Assign Reviewers and Labels (Optional):**
   * You can assign specific team members to review the PR and add labels to categorize it.
6. **Submit the Pull Request:**
   * Click the "Create pull request" button to submit.

**5. Review Process**

Once the PR is submitted, team members can review it:

* **Commenting:** Reviewers can leave comments and feedback directly on specific lines of code.
* **Approving Changes:** Reviewers can approve the changes if they find them satisfactory.
* **Requesting Changes:** If issues are found, reviewers can request changes, prompting the author to make adjustments.

**6. Merging the Pull Request**

After the review process is complete and any requested changes have been made:

1. **Merge the PR:**
   * If you have the necessary permissions, you can merge the pull request directly on GitHub by clicking the "Merge pull request" button.
2. **Choose the Merge Option:**
   * GitHub offers different merging strategies (e.g., merge commit, squash and merge, rebase and merge). Choose the one that best fits your project’s workflow.
3. **Delete the Branch (Optional):**
   * After merging, you can delete the feature branch to keep the repository tidy. GitHub typically provides an option to delete the branch right after merging.

**8**

Forking is a key feature on GitHub that allows users to create a personal copy of someone else’s repository. This enables you to experiment, make changes, and contribute back to the original project without affecting the original codebase. Here’s an overview of the concept of forking, how it differs from cloning, and scenarios where forking is particularly useful.

**What is Forking?**

When you fork a repository on GitHub, you create a copy of the repository under your own GitHub account. This forked repository is entirely separate from the original but maintains a connection to it, allowing you to propose changes back to the original repository via pull requests.

**How Forking Differs from Cloning**

* **Forking:**
  + Creates a copy of the repository under your GitHub account.
  + Allows you to make changes and propose them back to the original repository.
  + Maintains a connection to the original repository, enabling easier updates and collaboration.
* **Cloning:**
  + Creates a local copy of a repository on your machine.
  + Does not create a separate repository on GitHub; it simply allows you to work on the repository locally.
  + Changes made in a cloned repository must be pushed back to the original repository if you have the appropriate permissions, or you must create a new branch for a pull request.

**Scenarios Where Forking is Particularly Useful**

1. **Contributing to Open Source Projects:**
   * Forking is commonly used in open source development. You can fork a repository to make improvements, fix bugs, or add features without needing direct access to the original repository. After making changes, you can submit a pull request for the original maintainers to review and possibly merge your changes.
2. **Experimenting with New Ideas:**
   * If you want to try out new features or experiment with different approaches without affecting the original project, forking provides a safe environment to do so. You can modify your fork freely and even discard it if the changes don’t work out.
3. **Customizing a Repository:**
   * If a repository contains a project that you want to customize for your own use, forking allows you to maintain your own version while keeping track of the original project’s updates. This is especially useful for projects that are actively maintained.
4. **Learning and Practicing:**
   * Forking a repository can be a great way to learn from existing codebases. You can fork a project, explore the code, and make modifications to understand how it works. This is particularly beneficial for beginners looking to practice coding skills.
5. **Collaborative Development:**
   * In a team setting, if you want to work on a project without interfering with the main codebase, forking allows team members to work in their own environments. Each member can fork the repository, make changes, and then create pull requests to share their work.

**9**

GitHub issues and project boards are essential tools for tracking bugs, managing tasks, and organizing projects. They foster collaboration and streamline development processes by offering a structured way to manage tasks, discuss improvements, and prioritize work. Let’s break down their importance and how they can be used:

**1. GitHub Issues: Tracking Bugs and Managing Tasks**

GitHub issues provide a way to report bugs, propose features, or discuss ideas related to a project. They serve as a discussion thread where contributors and maintainers can communicate and work towards a resolution.

**Key Features of GitHub Issues:**

* **Labels:** Categorize issues based on type (e.g., bug, enhancement, question) or priority (e.g., high, low).
* **Assignees:** Assign issues to specific team members responsible for resolving them.
* **Milestones:** Group issues under a specific goal or feature that should be completed by a certain time.
* **Comments & Discussions:** Engage in real-time discussions to clarify the issue and propose solutions.
* **Issue Linking:** Link related issues to prevent duplication or connect bugs to specific feature requests.

**Example:**

A team working on an open-source library notices a bug in how the library handles API requests. A user opens an issue detailing the bug and provides steps to reproduce it. The maintainers label the issue as a "bug" and assign it to a developer. As the issue progresses, the assignee can comment on progress, link it to related issues, and close it once resolved.

**2. GitHub Project Boards: Improving Project Organization**

GitHub project boards function like Kanban boards, providing a visual representation of the workflow and progress of tasks. This helps teams organize tasks, visualize bottlenecks, and track the overall state of a project.

**Key Features of GitHub Project Boards:**

* **Columns:** Represent different stages in a workflow, such as "To Do," "In Progress," and "Done."
* **Cards:** Each GitHub issue can be turned into a card that moves through columns as the task progresses.
* **Automation:** Project boards can automate transitions, like moving issues from "In Progress" to "Done" once they are closed.
* **Filtering and Sorting:** Organize cards based on priority, deadlines, or assignees.
* **Multiple Project Views:** Allows the creation of different views for developers (focused on features), designers (focused on UI/UX tasks), etc.

**Example:**

An organization developing a web application uses a project board to manage tasks. They create columns like "Backlog," "Current Sprint," "In Review," and "Completed." Each feature request, bug, or task is represented as a card that moves across columns based on progress. Developers can focus on "Current Sprint" tasks while project managers keep an eye on what’s in the "Backlog."

**3. Enhancing Collaborative Efforts: Practical Use Cases**

These tools can significantly improve collaboration within teams, whether the project is open source or internal to a company. Here’s how:

* **Open Source Projects:**
  + **Community Involvement:** GitHub issues allow external contributors to report bugs and suggest improvements. This encourages active collaboration from the community.
  + **Transparency:** Project boards give visibility into the development process. Contributors can see the status of features and bugs in real-time, improving communication and reducing overlap in work.
* **Internal Team Projects:**
  + **Cross-functional Collaboration:** Designers, developers, and QA teams can use issues to manage their tasks. Designers might create issues for UI improvements, while developers work on coding tasks.
  + **Sprint Planning:** Teams can use project boards to organize tasks for sprints. Each sprint can be visualized as a project board, with features and bugs prioritized for completion during that time frame.

**Example:**

A distributed development team working on a mobile app uses a GitHub project board for sprint management. They assign tasks to different members of the team (UI designers, backend developers, and testers). As tasks move through stages from "In Progress" to "In Review" to "Done," each team member stays aligned on what is completed and what still needs attention.

**10**

Using GitHub for version control offers numerous advantages, but it also presents challenges, particularly for new users or teams unfamiliar with best practices. Below is a reflection on common challenges associated with using GitHub and strategies to overcome them to ensure smooth collaboration.

**Common Challenges**

**1. Merge Conflicts**

A frequent issue in collaborative projects occurs when multiple contributors work on the same codebase simultaneously, leading to merge conflicts. This happens when two or more people edit the same part of a file, causing Git to be unsure how to combine changes.

* **Pitfall:** New users may struggle to resolve merge conflicts, especially if they aren’t comfortable using Git commands or understanding the underlying code.
* **Best Practice:** Encourage contributors to pull changes frequently and work on distinct branches. Using feature branches and creating pull requests (PRs) can help isolate changes, reducing the likelihood of conflicts.

**2. Poor Commit Messages**

Writing clear, concise, and informative commit messages is crucial for keeping the project history understandable. However, new users often write vague or unhelpful commit messages like "fixed it" or "changed something."

* **Pitfall:** Inconsistent or unclear commit messages make it difficult to track the purpose of changes, leading to confusion during code reviews or when debugging issues later.
* **Best Practice:** Establish a **commit message convention**, such as the widely used [Conventional Commits](https://www.conventionalcommits.org/) standard. For example:
  + feat: for new features.
  + fix: for bug fixes.
  + refactor: for code changes that neither fix a bug nor add a feature.
  + docs: for documentation updates.

**3. Branching and Workflow Confusion**

Many new users are unfamiliar with the concept of branching and the purpose of different Git workflows (e.g., Git Flow, GitHub Flow). They may push directly to the main branch or fail to follow an organized process, which can lead to unstable code in production.

* **Pitfall:** Directly pushing to the main branch without review can introduce bugs or break existing functionality.
* **Best Practice:** Adopt a **branching strategy** that fits the team’s needs. For example, GitHub Flow advocates for:
  + Keeping the main branch stable.
  + Creating short-lived feature branches.
  + Using pull requests to propose, review, and merge changes.
  + Setting up branch protections to prevent direct pushes to the main branch.

**4. Lack of Code Reviews**

Collaborative projects benefit from code reviews to maintain code quality, spot bugs, and share knowledge among the team. However, new users or smaller teams may neglect this step, leading to unreviewed and potentially problematic code entering the project.

* **Pitfall:** Skipping code reviews can result in lower code quality, missed bugs, or security vulnerabilities.
* **Best Practice:** Encourage a **code review culture** where pull requests are reviewed by at least one other team member before being merged. GitHub’s pull request system allows for inline comments and discussions, making it a powerful tool for reviewing changes.

**5. Overwhelmed by Git Commands**

Git’s command-line interface can be intimidating for new users. Commands like rebase, cherry-pick, or even reset can seem complex, and mistakes in using them might lead to data loss or mismanaged repositories.

* **Pitfall:** New users might accidentally overwrite or lose important changes due to improper use of advanced Git commands.
* **Best Practice:** Educate new users on the basic Git commands like clone, pull, push, status, commit, and branch. Tools like **GitHub Desktop** or **GitKraken** provide user-friendly graphical interfaces that reduce the risk of making mistakes with advanced Git features.

**6. Unclear Project Structure and Conventions**

In large projects, inconsistent project structures and naming conventions can make it difficult for contributors to understand where to add or modify code, leading to errors or disorganization.

* **Pitfall:** Without a clear directory structure or naming conventions, contributors may introduce duplicate functionality or place files in the wrong directories.
* **Best Practice:** Establish a **clear project structure and coding conventions** from the start. Provide a CONTRIBUTING.md file with guidelines for contributors, including details on naming conventions, file organization, and how to submit code through pull requests.

**7. Excessive Large Binary Files**

Git is optimized for tracking code and text-based files, but large binary files (e.g., media assets, compiled files) can bloat a repository and slow down operations like cloning or pulling.

* **Pitfall:** Adding large files directly to the repository can slow down performance, making the project difficult to manage over time.
* **Best Practice:** Use **Git LFS (Large File Storage)** to handle large binary files. It stores the binary files outside the repository but keeps a reference to them within Git, improving performance.

**Strategies to Overcome Challenges and Ensure Smooth Collaboration**

1. **Use Pull Requests (PRs) Effectively:**
   * PRs provide a structured way to propose and discuss changes before merging them into the main codebase.
   * Include detailed descriptions in PRs to explain the purpose of the change, reference related issues, and highlight any areas that require specific attention.
2. **Enforce Branch Protections:**
   * Set up branch protection rules to prevent direct commits to the main branch, requiring PR reviews and automated tests to pass before merging.
3. **Continuous Integration (CI) Tools:**
   * Use CI tools like **GitHub Actions**, **Travis CI**, or **CircleCI** to automatically run tests whenever a pull request is opened. This ensures that the code is functional before merging it into the main branch.
4. **Automate Issue Tracking:**
   * Link pull requests to issues to automatically close them once the associated PR is merged. This maintains a clear connection between code changes and the bugs or features they address.
5. **Provide Git Training and Resources:**
   * Offer regular workshops or training sessions for new contributors on how to use Git and GitHub effectively. Provide resources like cheat sheets and encourage the use of visual Git clients for those uncomfortable with the command line.
6. **Encourage Small, Focused Commits:**
   * Large commits that encompass multiple changes can be difficult to review. Encourage developers to make small, incremental commits that address specific tasks or bugs. This improves the readability of the project’s history and simplifies code reviews.