

# 01 Course Introduction

GitHub Essentials



## Pete Behague

*Principal Technical Learning Specialist*



Peter.Behague@qa.com



[www.linkedin.com/in/pete-behague](https://www.linkedin.com/in/pete-behague)

**QA**

## Housekeeping

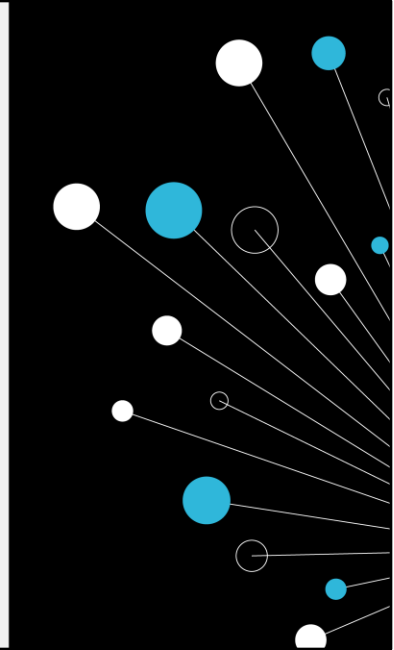


QA

## Learning objectives

- Gain the foundational skills and knowledge needed to effectively use GitHub for version control and collaboration
- Know how to:
  - Create repositories
  - Make commits
  - Carry out branching
  - Invoke pull requests
  - Merge and manage merge conflicts
- Configure and use Visual Studio Code to use Git and GitHub

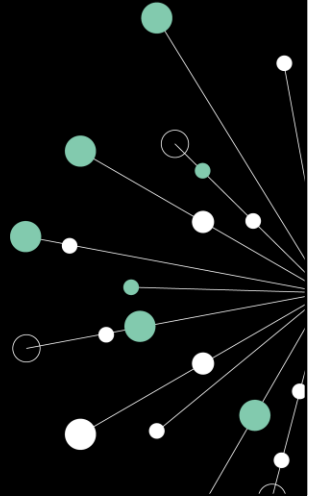
QA



## Course Outline

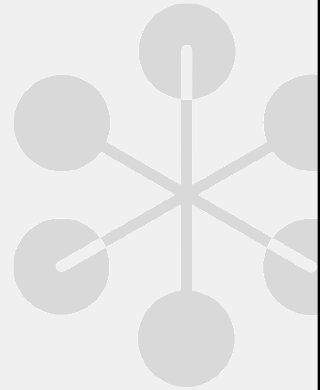
- Basic Git Commands
- Branching and Merging
- Introduction to GitHub
- Repository Creation and Setup
- Linking VSCode
- Pull Requests
- Managing Merge Conflicts
- Review and Q&A

**QA**



## Contents

- Course administration
- Pre-requisites
- Course objectives
- Course outline
- Introductions
- Questions
- Allocation of Virtual Machines



**QA**

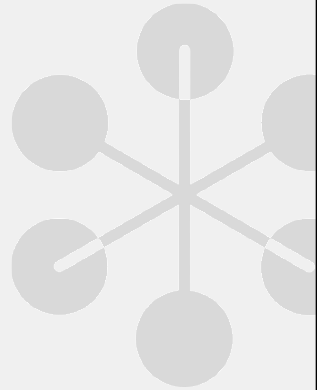
## Pre-requisites

### Essential skills:

- Basic Computer Skills:
  - Be able to navigate the files system (create, move, delete files and folders)
  - Exposure to text editors (Notepad++, Visual Studio Code)

### Beneficial skills

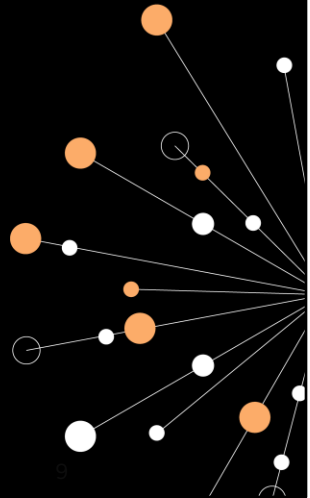
- Familiarity with basic command-line operations
- Programming awareness/Interest in software development



## Activity: Introductions

- Preferred name
- Organisation & role
- Experience of Programming, C# and OOP
- Key learning objective
- Hobby/Interest/Sport

QA





## Questions

Golden rule

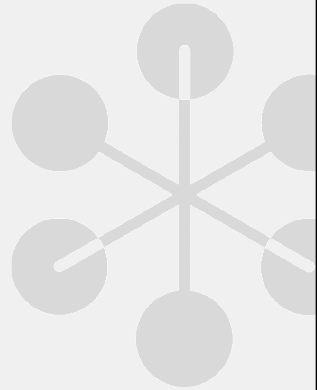
- 'There is no such thing as a stupid question'

First amendment to the golden rule

- '... even when asked by an instructor'
- Please have a go at answering questions

Corollary to the golden rule

- 'A question never resides in a single mind'
- By asking a question, you're helping everybody

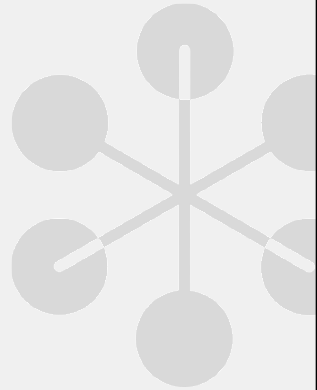


**QA**

## Allocation of Virtual Machines

Lab instructions assume you are using Microsoft Visual Studio (Community Edition). You are quite welcome to use a local installation.

However, your tutor can allocate you a virtual machine which has all the necessary software that you need for the course already installed on it.



## 02 Introduction to Git

GitHub Essentials

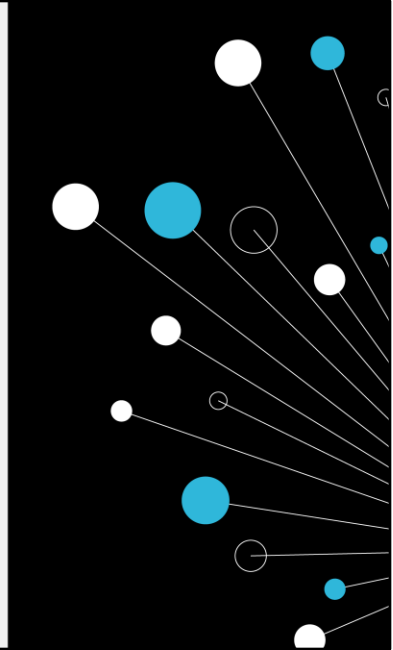


```
the text runs across the top
// persisted properties
<html> <p style="font-weight:bold;">HTML font code is displayed
<html> <body style="background-color:yellowgreen;">
<html> <div style="text-align:center;">
  // Non - text - :200px;"persisted properties
  <html> <errorMessage = ko , observable() ; ko
  <p style="color:orange;">HTML font code is displayed
  function todoitem(data) ;
    var self = this <html> <errorMessage = ko
    data = data || {} <html> <errorMessage = ko
  // Non - persisted properties
  <html> <errorMessage = text - :200px;"ko , observable() ; ko
  <p style="font-weight:bold;">HTML font code is displayed
  <body style="background-color:yellowgreen;">
    text - :200px;"> <todoitemid = data.todoitemid ;
    - text - :200px;"persisted properties
    <errorMessage = ko , observable() ; ko
```

## Learning objectives

- To understand the need for source control and how to use and configure basic Git commands

QA



## Session Content

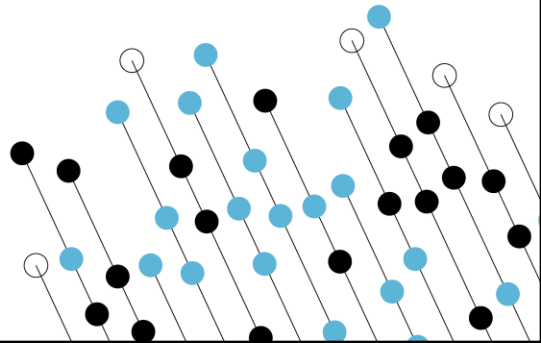
- Introduction to Version Control
- Installing and Configuring Git
- Basic Git Commands

**QA**



# Introduction to Version Control

Introduction to Git

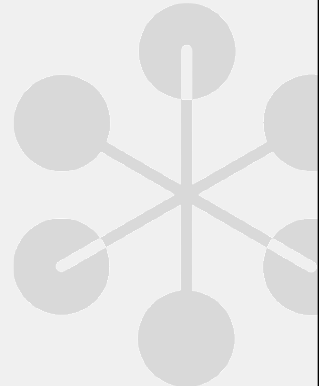


# Introduction to Source Control

Source Control is the practice of tracking and managing changes to code.

Known by a few different names:

- version control
- revision control



QA

Note: Whilst **Source Control** and **Version Control** are closely related terms, they are not entirely the same. There is a lot of overlap especially in purpose and function, but there are some subtle differences in scope and usage.

**Source Control** refers specifically to the management of source code files in a project.

- It ensures that code is organized, changes are tracked, and conflicts between team members are avoided.

- Example systems: Git, Subversion (SVN), Perforce.

**Version Control** is a broader concept that applies to tracking and managing changes to any kind of file, not just source code.

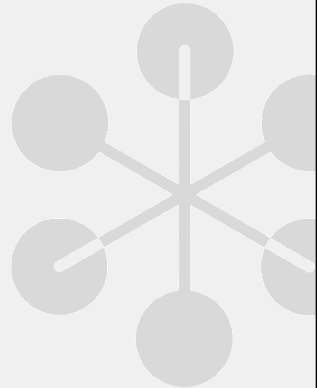
- It creates a history of revisions, making it possible to revert, branch, and merge changes for collaborative workflows.

- Example systems: Git (source code focus), Microsoft Word version history (general focus).

## Vital for managing software development projects

Being able to track changes to code allows developers to:

- Centralise all code changes and additions to one code repository
- Allow for simple and effective collaboration within development teams
- Control the integration of new code into the codebase
- Track changes from the entire team over the full lifetime of the project
- Revert code back to previous versions



**QA**



## Types of Version Control Systems

### Local Version Control:

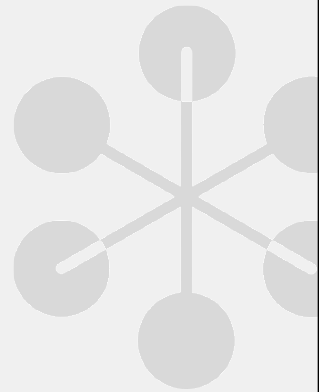
- All versions are stored on a single computer.
- E.G. Simple backup tools or systems like RCS (Revision Control System).

### Centralized Version Control (CVCS):

- A single server stores all files and version history.
- Clients check out, edit, and commit changes back to the server.
- E.G. SVN (Subversion), Perforce.
- Limitations: If the server goes down, the system is inaccessible.

### Distributed Version Control (DVCS):

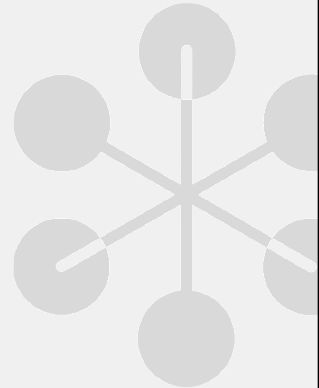
- Every user has a full copy of the repository
- Changes can be committed locally and pushed to a remote repository.
- E.G. Git, Mercurial.
- Advantages:
  - No single point of failure.
  - Offline access to the full repository.



## Source Control Management (SCM)

Used by developers to implement source-control practices giving the ability to:

- Store code in a central repository
- Track changes over time
- Create code branches so that additions are made in isolation from stable code
- Merge new code into a stable release branch, known as the main branch
- Integrate with CI/CD automation tools (e.g. Jenkins and CircleCI) such that code will be built and tested as it is generated and pushed to the repository

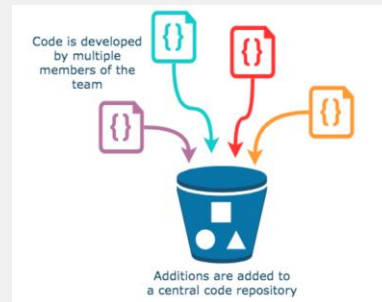


**QA**

## Repositories

Having one place where your code resides means that versioning stays consistent, regardless of who's working on it.

However, there is a danger developers could overwrite another developer's changes. So...

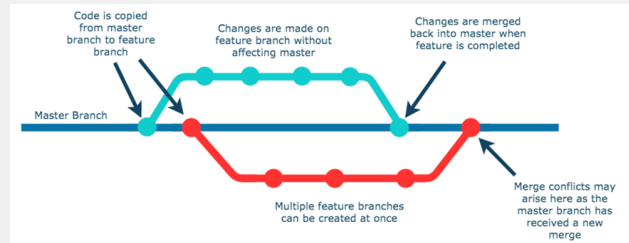


**QA**

## Branching

Allows for each developer to work on new features in isolation by creating a version of the source code that only they are working on

**Note: We will look into branching in a bit more detail later**



QA

Without source control, having multiple developers developing on the same code base simultaneously inevitably results in code conflicts and breakages to functionality, as two developers may need to work on the same parts of code to create their particular features.

Branching allows for each developer to work on new features in isolation by creating a version of the source code that only they are working on. This new version is called a **feature branch**, while the source code lives on the **main branch**.

Once changes have been made and the feature is implemented, the feature branch code is merged back into the source code on the main branch.

The SCM system will automatically detect the differences between the feature and main branches and alert the developer if there are any *conflicts*.

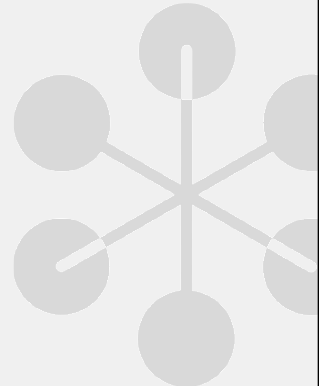
Conflicts may arise if changes have been made to the main branch **after** the feature branch has been created. The SCM will clearly show the changes that need to be made before merging can occur.

## Code Tracking

Code tracking allows development teams to keep track of all changes made to a project over time.

Problems easily solved as code contributors can be consulted directly regarding their code

If a feature causes a bug in the software after it is merged to main, code tracking makes it easy to revert the main branch back to a previous stable state



**QA**

Code tracking allows development teams to keep track of all changes made to a project over time. This allows for greater organisation, as all additions to code are fully documented and attributed to the developer who made them.

Problems can then be solved with ease as code contributors can be consulted directly regarding their code; if a contributor is no longer a part of the team and can't be consulted directly, their changes are still fully documented allowing the team to track their contributions despite their absence.

If a feature causes a bug in the software after it is merged to main, code tracking makes it easy to revert the main branch to a previous stable state until fixes can be made.

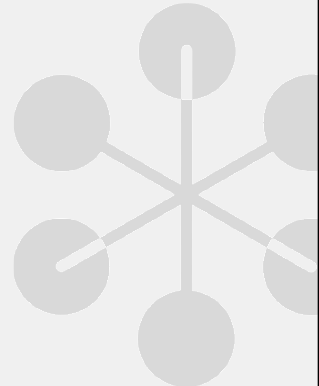
## Git

Git is a distributed version control system (VCS) used for tracking changes in files and coordinating work on those files among multiple people.

It was created by Linus Torvalds in 2005, primarily to manage the development of the Linux kernel.

Most widely used version control system in software development.

Versions are available for Windows and Mac operating systems.



**QA**

## Git Bash

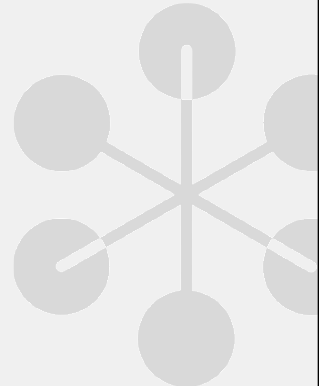
### Git Bash

Command prompt where Git instructions can be used to interact between GitHub and a local copy of the code.

Can use Bash CLI (Command Line Interface) in Windows or Mac.

Git Bash allows you to interact with the Windows environment using Linux commands and comes preloaded with Git for source control.

Can use Window's Powershell or command prompt as an alternative



QA

**Git Bash** is the command prompt used to interact between GitHub and a local copy of the code. It is a powerful and easy to use **Bash CLI** (Command Line Interface) in Windows or Mac, however, it is recommend to use the default terminal in Mac.

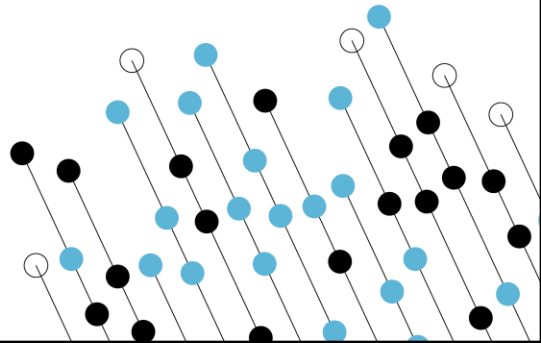
Git Bash allows you to interact with the Windows environment using Linux commands and comes preloaded with Git for source control.

The **pwd** command in Bash displays the current working directory—the directory in which the Bash session is currently operating. This is similar to running **cd** without any arguments in a Windows command prompt.

The **ls** command in Bash is used to list the contents of the current directory, functioning similarly to the **DIR** command in the Windows console. Both Bash and the Windows command prompt support the **cd** command, which stands for "Change Directory". When followed by a directory name, **cd** updates the session's current working directory to the specified location.

# Installing and Configuring Git

Introduction to Git





## Git Bash

In your browser of choice visit Git Bash download:

<https://git-scm.com/downloads>

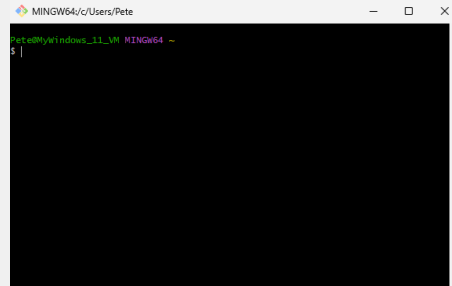
Download appropriate version for your operating system

Find the installer that was downloaded and run it  
Then take the default options

Git Bash app can be launched in the conventional way (Windows Start button...)



Git Bash  
App



**QA**

## Git Configurations (git config)

Need to have your username and email configured.

Can be done for single downloaded repository or globally for all cloned repositories

### Setting Config Globally

```
# git config --global user.name "[USERNAME]"
$ git config --global user.name "John Doe"
# git config --global user.email "[EMAIL]"
$ git config --global user.email "johndoe@example.com"
```

### Setting Config Locally:

```
# git config user.name "[USERNAME]"
$ git config user.name "John Doe"
# git config user.email "[EMAIL]"
$ git config user.email "johndoe@example.com"
```

QA

It is important to set a username and email address for Git when using a local installation. The settings are essential because Git uses them to associate your identity with the commits you make. Without this information, Git won't allow you to make commits effectively.

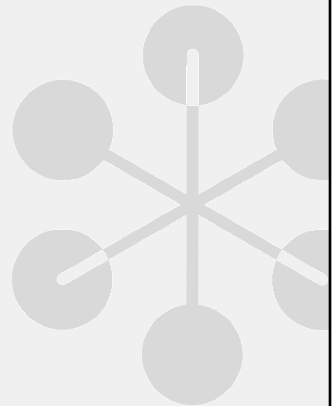
## Git Configuration – Accessing User Settings

The values can then be accessed using:

```
% git config user.xxx
```

You can list all the configurations with:

```
% git config --list
```

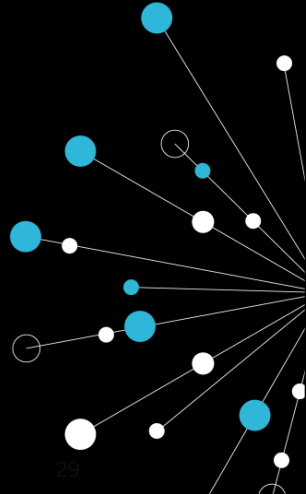


## Quick Lab - Git Installation and User Configuration

Download and install Git Bash and use the command line to configure Git with your user name and email

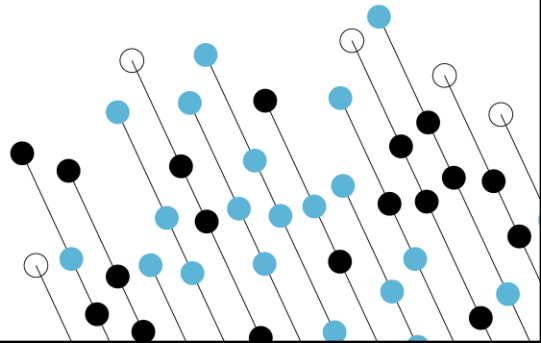
- Check the settings to ensure these are set

QA



# Basic Git Commands

Introduction to Git



## Git Commands – Entomology

Git commands all follow the same convention:

- The word 'git'
- Followed by an optional switch
- Followed by a Git command (mandatory)
- Followed by optional arguments

```
git [switches] <commands> [<args>]
```

```
git -p config -global user.name "Dave"
```

**QA**

The -p switch paginates the output if needed.

## Getting started with Git

When you first launch Git you will be at your home directory

- `~` or `$HOME`

Through the Git Bash you can then move through and modify the directory structure

Command	Explanation
<b>ls</b>	<b>Current files in the directory</b>
<b>mkdir</b>	<b>Make a directory at the current location</b>
<b>cd</b>	<b>Change to a specified directory</b>
<b>pwd</b>	<b>Print the current directory</b>
<b>rm</b>	<b>Remove a file (optional <code>-r</code> flag to remove a directory)</b>

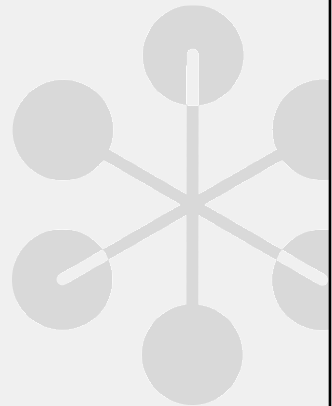


## Git Help

- Gives access to Git's built in documentation directly from the command line
- Syntax:

```
git help <command>
```
- Example usages:

```
git help add  
git help commit
```



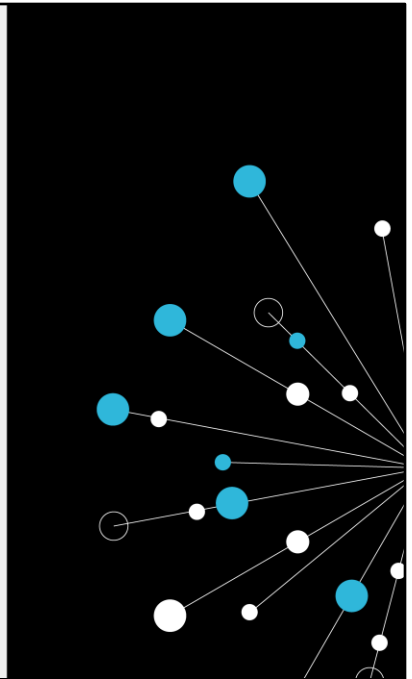


## Quick Lab - Git Help

Use your git bash command line to access git help to find out about:

- help
- glossary
- -a
- config
- -g

**QA**



To display the Git glossary:

```
git help glossary
```

To list all Git commands:

```
git help -a
```

To search git documentation for common concepts:

```
git help -g
```

## GIT Key Concepts - Repos

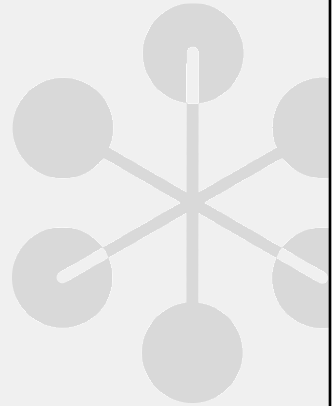
### GIT holds assets in a repository (repo)

- A repository is a storage area for your files
- This maps to a directory or folder on your file system
  - These can include subdirectories and associated files
  - Relevant Windows command prompt commands include:

```
mkdir firstRepo  
cd firstRepo  
git init
```

The repo requires no server but has created a series of hidden files

- Located in .git folder



**QA**

Make a new directory (folder):

```
mkdir firstRepo -
```

Change to new directory:

```
cd firstRepo
```

Initialise a get repository in the current directory:

```
git init
```

## Using git status

You can use the `git status` command to see detailed information about the status of your git repository

```
git status
On branch master

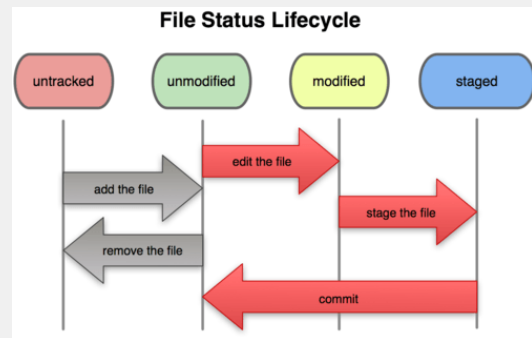
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
```

## Recording Changes to a Repository

Each file in a git directory can be tracked or untracked

- Tracked files are files that were in the last snapshot
  - They can be unmodified, modified or staged
- Untracked files are everything else
  - Not in your last snapshot or staging area



**QA**

The main tool you use to determine which files are in which state is the git status command.

## Staging

Staging is the step that you must take before committing a change

Enables you to choose what changes are actually going to get committed to the repository

```
# stage all files (whole working tree)
git add --all

# stage all files (from current dir and subdirs)
git add .

# stage selected files: git add [FILES]
git add file_1.txt file_2.txt
git add *.txt
```

**QA**

Staging is often bundled up with the commit

## Staging Files using the add Command

To add a new file to the repository, use the 'add' argument

```
# a single file
git add myFile.txt

# To add all the files (tracked and untracked files)
git add -A

# or
git add --all
```

`git status` will show the newly added file

```
git status
On branch master
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   myFile.txt
```

**QA**

Adding (`git add`) is a specific command that stages changes. Staging is the result of adding changes to the staging area. It's the step before committing.

Note to create a file using git bash you could do the following:

Navigate to the target directory:

```
cd path/toMyDirectory/
```

Either create the file using `touch` (which will create an empty file called `myFile.txt` in the current directory):

```
touch myFile.txt
```

Or use redirection to create a file (called `myFile.txt`) and add the text "Some Content" to it:

```
Echo "Some Content" > myFile.txt
```

Or use a text editor such as Notepad.exe:

```
Notepad myFile.txt
```

To verify the new file(s) exist use the `ls` (list files) command:

```
ls
```

## Committing files

To commit the changes, use the "commit" argument

- You can specify the author with the `-a` flag
- The `-m` flag is used to set a message

```
$ git commit -m "Add a new file called myFile.txt"
[master 93e8300] Add a new file called myFile.txt
1 file changed, 1 insertions(+)
```

This is now saved to the local version of your repository

`git status` will show everything is committed

```
$ git status
On branch master
nothing to commit, working tree clean
```

**QA**

If you issue a `git commit` with no message flag bash will launch a vi editor to which you must do the following:

- Switch to input mode (press `i`)
- Enter a commit message
- Hit `ESC` to complete the message
- Enter `ZZ` or `:wq` to exit the vi edit

## Viewing the commit log

- The `git log` command displays a detailed commit history of a Git repository including:
  - Commit Hash: A unique identifier for each commit
  - Author information
  - Date
  - The commit message

```
$ git log
commit 1ab2ce4d6a09e3ff436338cf178ab2d9e8eb327b (HEAD -> master)
Author: APerson <aperson@qa.com>
Date: Wed Jan 1 00:00:00 2025 +0000

    New folder and file

commit 3804b6a4549c32544ec6757ec7d2c24dede13d0e
Author: APerson <aperson@qa.com>
Date: Wed Jan 1 00:01:00 2025 +0000

    Add a new file called myFile.txt
```

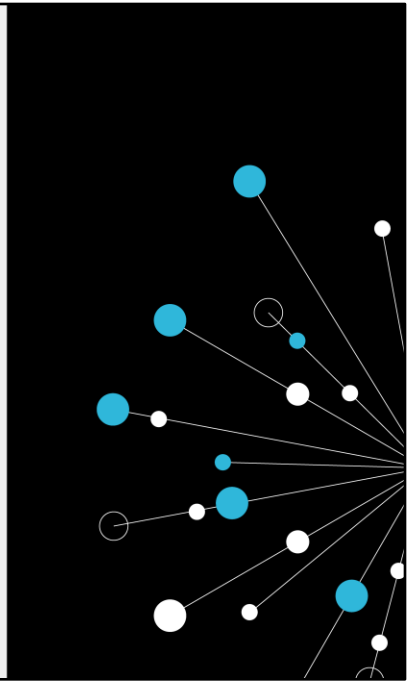
You can visualise the log history by running the `gitk` command in the folder of a git repository. Many IDE's will also provide you with a visual representation of the log.



## Quick Lab - Using Git Bash to create a first repository

- Launch your git command line (GitBash)
- Ensure you are at your home directory
- Create a directory called GitTest
- Change to the new directory
- Run the git init command
- Add a file called myFile.txt to the new directory
- Add the new file to the staging area by typing and run `git add -A`
- Enter and run `git status` and note that your new file is being tracked
- Enter `git commit` (don't forget the message)
- Enter `git status`
- Enter `git log`

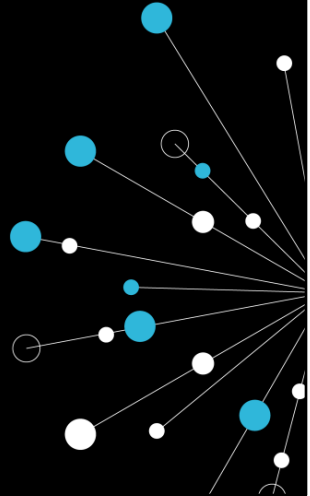
**QA**



# Appendix

SCM Tools and Repository Hosting Services

QA



## SCM Tools and RHS's

### Source Control Management

- Git (Most Common)
- Mercurial
- Subversion (often abbreviated to SVN)
- CVS
- Perforce

### Repository Hosting Services

- GitHub
- GitLab
- Bitbucket
- SourceForge
- Launchpad

**QA**

**Git** is by far the most common SCM system used in software development. **Git** is a *free* and *open-source* version control system. Its ubiquity is largely due to how easy it is to learn and its tiny footprint, which is a result of being written in low-level C code.

It is a decentralised SCM tool, meaning its operations are largely performed on your local computer and requires no communication with an external server, resulting in very fast performance.

Many cloud providers have their own code repository offerings which offer simple and powerful integration with other cloud services:

- AWS CodeCommit
- Azure Repos (as part of Azure DevOps)
- Google Cloud Source Repositories

## 03 Branching and Merging

GitHub Essentials

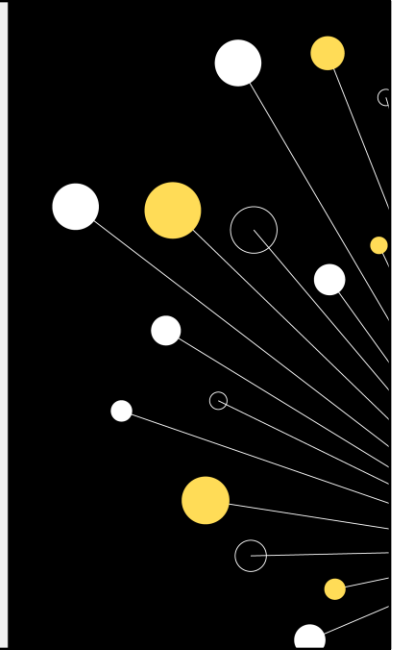


```
the text runs across the top
// persisted properties
<html> <p style="font-weight:bold;">HTML font code is displayed
<html> <body style="background-color:yellowgreen;">
<html> <div style="background-color:yellowgreen;">
// Non - text - :200px;" persisted properties
<html> <errorMessage = ko , observable() ; ko
<p style="color:orange;">HTML font code is displayed
function todoitem(data) ;
var self = this
data = data || {}
// Non - persisted properties
<html> <errorMessage = text - :200px;"
<p style="font-weight:bold;">HTML font code is displayed
<body style="background-color:yellowgreen;">
text - :200px;" <todoId = data.todoId
- text - :200px;" persisted properties
errorMessage = ko , observable() ;
```

## Learning objectives

- Understanding why branching is such a useful feature
- Learn how to create branches, switch between them, and merge them back into the main branch.

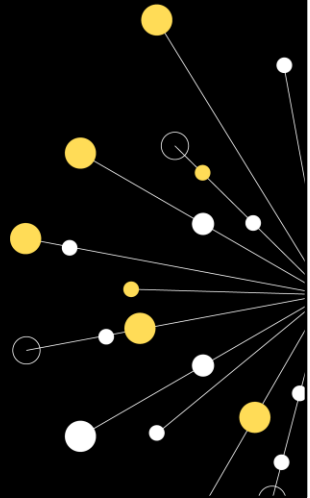
QA



## Session Content

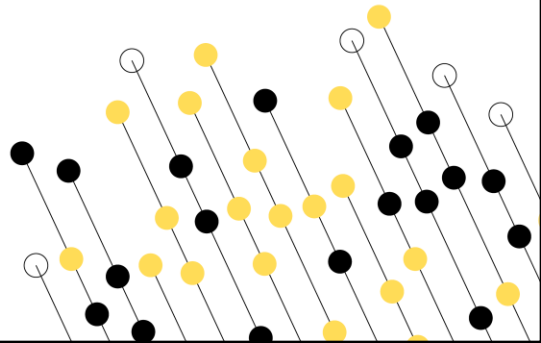
- Understanding branches
- Creating a new branch
- Switching branches
- Merging branches
- Deleting branches

QA



# Understanding Branches

Branching and Merging

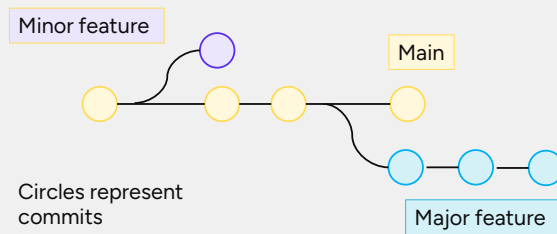


## What is Branching?

Branches enable you to work on separate parts of a project independently, without affecting the main branch.

Once your work is finished, the branch can be merged into the main project seamlessly.

Additionally, you can switch between branches to work on different tasks or features without any interference between them.



QA

Branches are other “paths” or “lines” of development. Imagine that you’re making a website for a client, and it consists of an HTML file and a CSS file. You work on these two files in a default repository. This default repository is also conveniently associated with a default branch, which is often called “main”, or “trunk”.

Branching allows you to diverge from the main line of development without doing accidental damage to the main line. Git branches are very lightweight compared to other VCS. It encourages a workflow that allows you to branch and merge

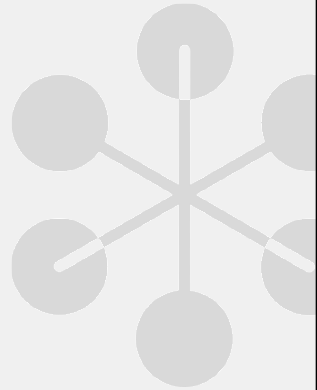
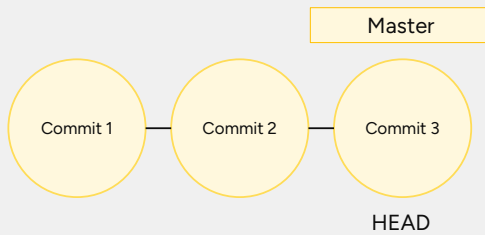
Branches build on core Git features. When you commit you have a snapshot of current content along with zero or more pointers to the current commits based on this repository or parents



## How Branch History Works 1

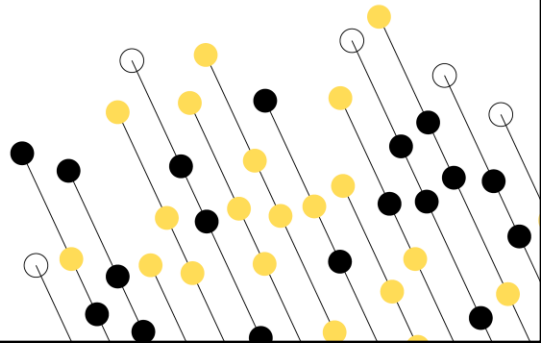
Commits occur in a line created when we first initialise the repo

- Git also creates a default Branch called Master
- Master is a pointer to the last commit in the branch
- The HEAD represents our current location
- Normally in the same place as Master



## Creating a new branch

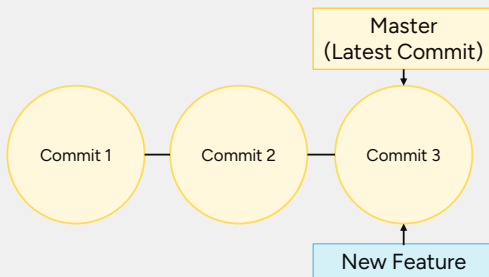
Branching and Merging



## How Branch History Works 2 – Branch Creation

Use `git branch <new-feature>` to create a branch

- Branches are just pointers to commits.
- When you create a branch, a new pointer is created
- The repository remains unchanged

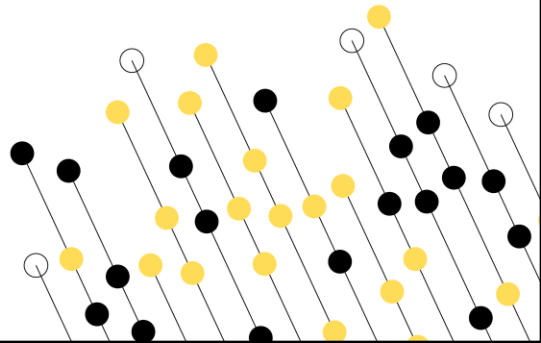


QA

52

# Merging Branches

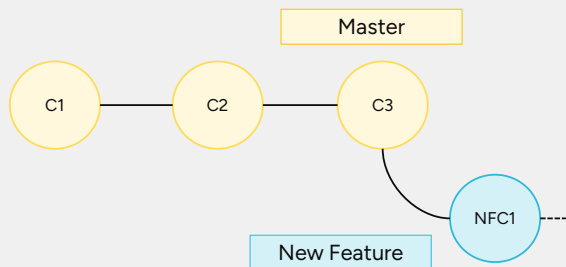
Branching and Merging



## How Branch History Works 3 – Branch Switching (Checkout)

To start adding commits to a new branch you need to

- Select it using `git checkout <branch-name>`
- Or use `git checkout -b <branch-name>` to create and checkout in a single operation
  - Head is moved to checked-out branch
- The use `git add` and `git commit` to work on branch



You can navigate to any prior commit and check it out. From that point create a new branch and then work on the fix and commit when ready.

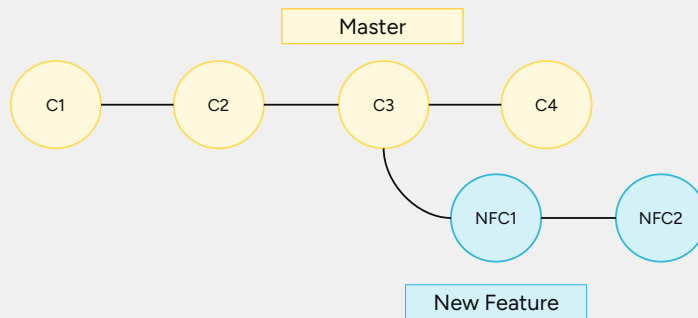
```
$ git checkout -b hotfix thepast
```

The above code creates a new branch and sets from a previously defined start point (thepast). The start point can be a branch, tag or SHA1 ID

## How Branch History Works 4 – Work on Chosen "Branch"

Work can continue on the original "Master" path as well as the branch

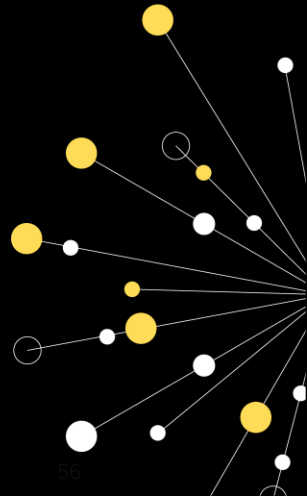
Branches can be further branched



## Quick Lab – Create and Work on a Branch

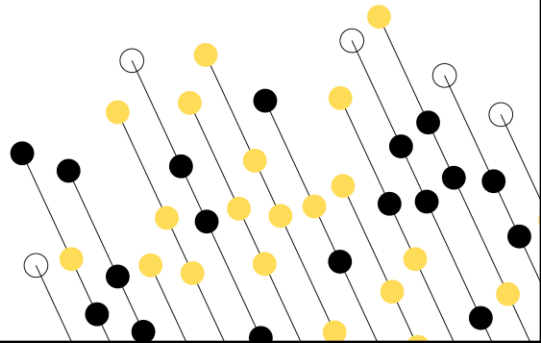
- In most recent repo
- Create and Checkout a new branch
- Use touch to create a new file
- Add it to the repo
- Commit the changes
- Use Checkout to return to the master
- Check the contents of your directory and note what has happened to the new file
- Check the log history

QA



# Merging Branches

Branching and Merging

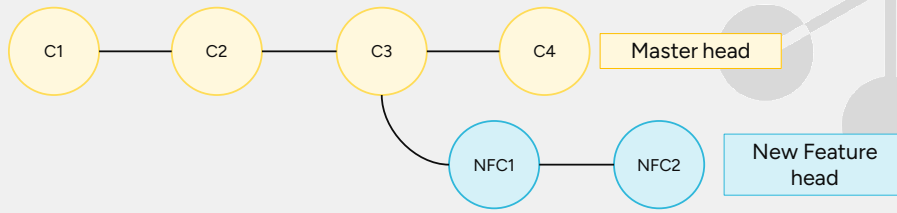




## Merging Branches - 1

Branches diverge code bases – ultimately want to merge branches

- Branches merge into the **current** branch
- Git takes two branch heads (C4 & NFC2) and finds a common base commit (C3 below)

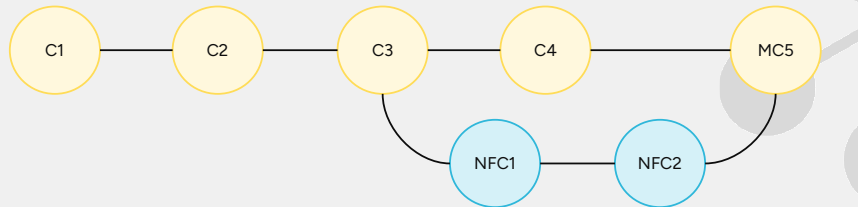


QA

## Merging Branches - 2

Use `git merge <branch-name>` to combine a specified branch (NF) with the **current** branch.

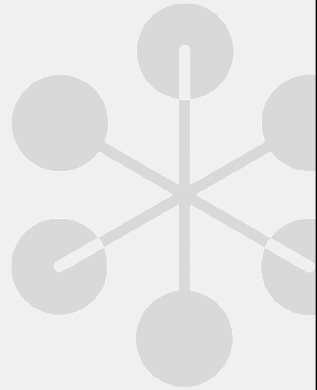
- From the common base (C3) git creates a new "merge commit" that combines the changes of each of the branch's commit sequences.



QA

## Before Merging

- Use `git status` to ensure the HEAD is pointing at the correct merge receiving branch.
  - Use `git checkout` if necessary
- Make sure both branches are up to date with their latest changes (i.e. committed)
- You can use `git diff <master> <branch>` to explore the differences between branches.
  - The order of the output is significant – the master is shown first

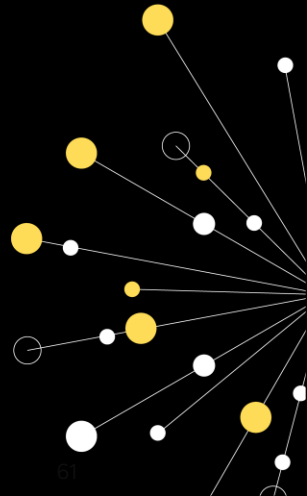


## Quick Lab – Merging Branches

- Setup
  - Initialize a new repository
- Create and modify a second branch
  - Create and switch to a new branch called new-feature
  - Add a new file called myNewFeaturesFile.txt to the new-feature, add a line of text to it and commit
  - Switch back to main

Continued on next slide

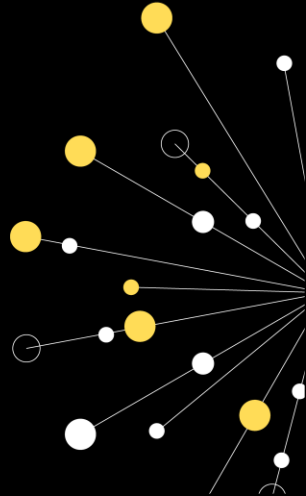
QA



## Quick Lab – Merging Branches

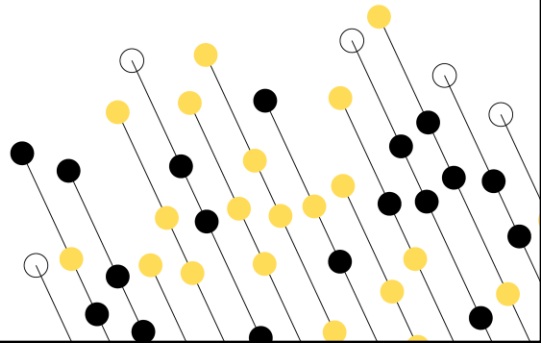
- Attempt to merge
  - Attempt to merge the new-feature branch into main.  
The merge should be successful
- Verify the Merge
  - Check the log
  - Inspect the master.txt and feature.txt files

QA



## Deleting Branches

Branching and Merging



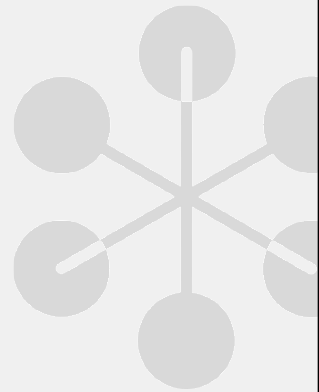
## Deleting Branches

If the work on a branch is no longer needed it can be deleted

```
$ git branch -d newbranch
```

You need to be very careful – git really deletes the branch

- Git does protect us from deleting the branch we are currently on
- If there are uncommitted changes in the branch you may need to use the `-D` flag



QA

64

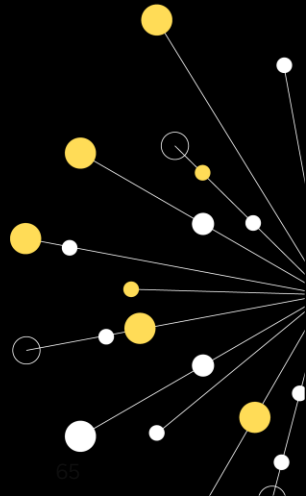
**Note:** merging a branch into the master (or main) branch in Git does **not automatically delete** the merged branch. After the merge, the branch still exists in your local and remote repositories unless you delete it manually.

## Quick Lab – Deleting Branches

### Making and Deleting a Branch

- Create a new directory called DevOps on your home directory.
- Initialise it as a Git repository.
- Add a file called bar and commit it.
- Use the branch command to see the current branches.
- Create a new branch called ops and check it out.
- Delete the Master branch.

QA



65





```

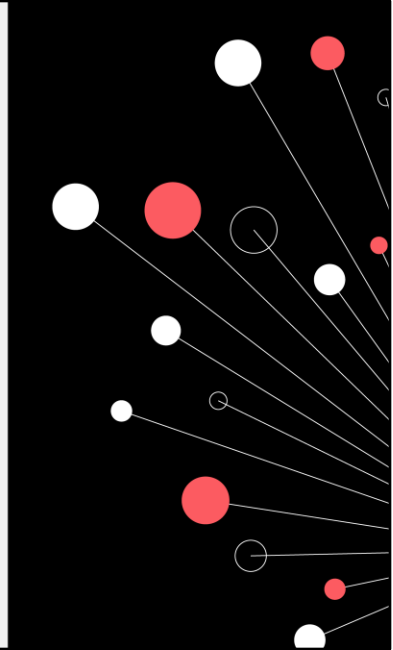
// persisted properties
<html> <p style="font-weight:bold;">HTML font code is done
<html> <body style="background-color:yellowgreen;">
<html>text - :200px;" <todoId = data.todoId - :200px;"
// Non - text - :200px;" persisted properties
<html> <errorMessage = ko , observable()
<p style="color:orange;">HTML font code is done
function todoItem(data) ; <html> <errorMessage =
var self = this <html> <errorMessage =
data = data || {} <html> <errorMessage =
// Non - persisted property function, todoItem
<html> <errorMessage = text - :200px;" ko , observable()
<p style="font-weight:bold;">HTML font code is done
<body style="background-color:yellowgreen;">
text - :200px;" <todoId = data.todoId - :200px;"
- text - :200px;" persisted properties
<errorMessage = ko , observable()

```

## Learning objectives

- To understand what GitHub is and why it's an essential tool for developers
- Know how to set up an account
- Be able to navigate the GitHub interface

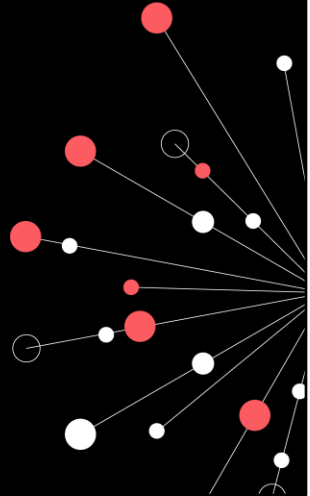
QA



## Session Content

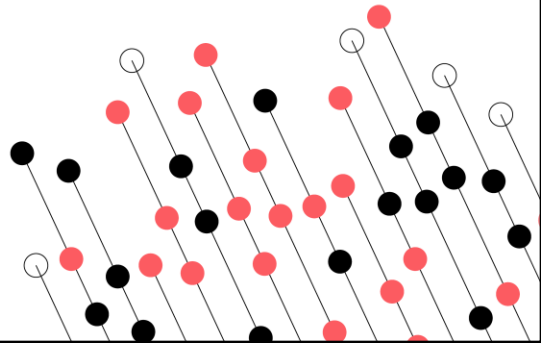
- Overview of Git and GitHub
- Setting up a GitHub account
- Exploring the GitHub interface

**QA**



# Overview of Git and GitHub

Introduction to GitHub



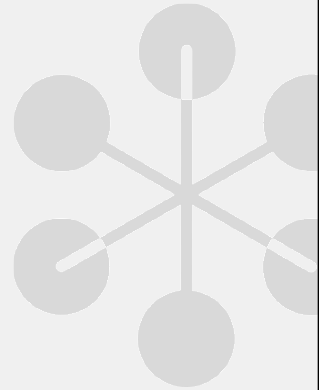
## What is Git? A Summary

Git is a distributed version control system that lets you manage and keep track of your source code history.

- Tracks history of files and changes.
- Allows branching and merging for parallel development.
- Distributed nature ensures every developer has the full repository.

Why Use Git?

- Enables collaboration.
- Facilitates experimentation without affecting the main code.
- Provides a safety net with the ability to revert changes.
- Enforces data integrity.
- Allows and manages local and remote development.



**QA**

## What is GitHub?

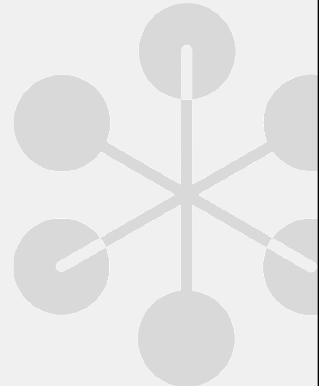
Cloud-based code hosting service used to help developers store, manage, track and control changes to their Git repositories

- Repository hosting for version control.
- Hosts collaboration tools: Pull requests, code reviews, and issues.
- Integrated CI/CD pipelines and automation tools\*.
- Community-driven with public and private repositories.

Why Use GitHub?

- Simplifies collaboration and sharing.
- Provides a backup for repositories.
- Connects developers through open-source projects.
- Can be and is used by developers to showcase projects.

**QA**



**GitHub** is a cloud-based code hosting platform to help developers store, manage, track and control changes to their code. It is a platform used for collaboration and version control.

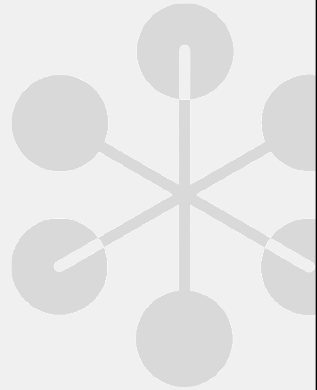
\* Beyond the scope of this course

## Git vs. GitHub

Feature	Git	GitHub
<b>Purpose</b>	Version control system	Hosting and collaboration
<b>Storage</b>	Local and distributed	Cloud-based
<b>Collaboration</b>	Requires manual setup	Built-in tools
<b>Usage</b>	Works offline	Requires internet access

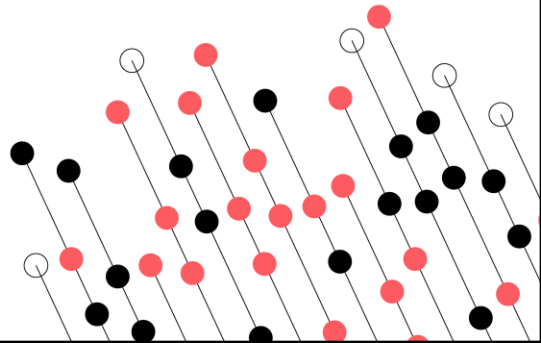
GitHub allows you to fork personal copies of repositories to experiment or contribute

**Note:** Many Git commands but **not** all, can be run from within GitHub



# Setting up a GitHub account

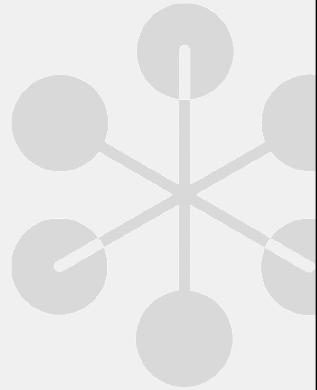
Introduction to GitHub





## Setting Up a GitHub Account

- Visit GitHub (<https://github.com>)
- Sign Up (username, email address, password)
  - Verify your email address
  - Choose a plan (free or paid)
  - Complete Onboarding (Optional)
  - Configure Profile
- Consider configuring 2-Factor Authentication



**QA**

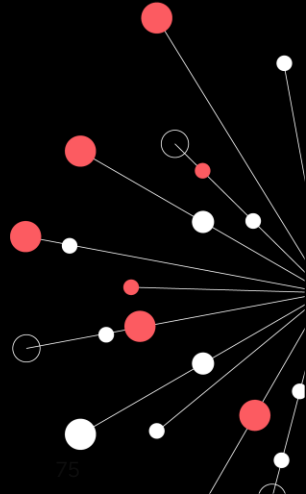
Create your GitHub account

<https://docs.github.com/en/get-started/start-your-journey/creating-an-account-on-github>

## Quick Lab – Set up a GitHub Account

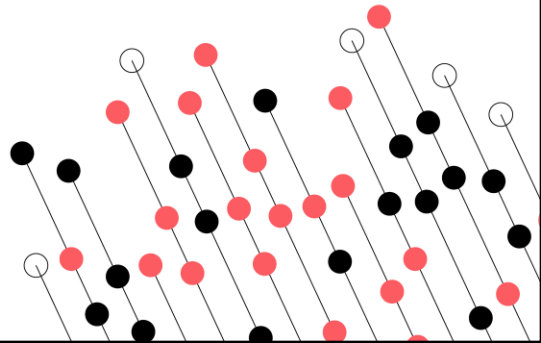
If you've not already done so set up a GitHub account by following the steps on the previous slide

QA

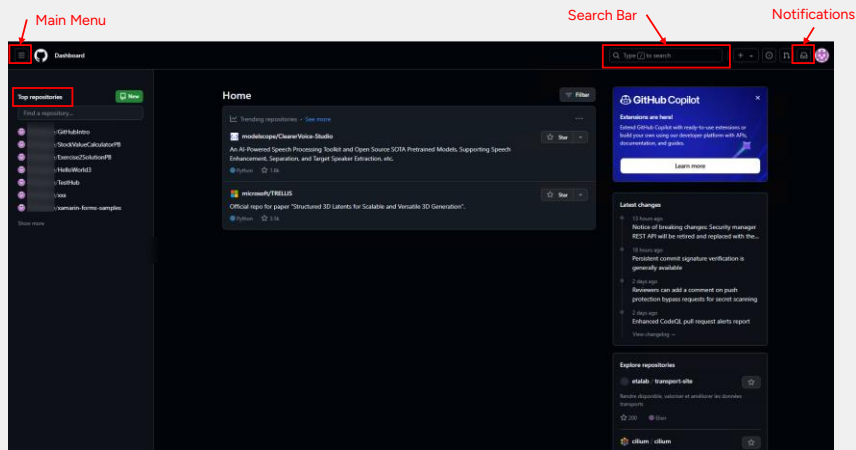


# Exploring the GitHub interface

Introduction to GitHub



## GitHub Home Page



QA

77

The Main menu takes you to:

- Issues: Allow you to plan, discuss and track work for specified repositories
- Pull Requests: Proposals to merge a set of changes from one branch into another which can be reviewed by others before integration takes place
- Projects: Adaptable tool for planning and tracking work on GitHub.
- Discussions: Collaborative community forum around projects.
- Codespaces: Cloud-based development environment
- Explore: Discover public repositories and trending projects
- Marketplace: Links you to developers who want to extend and improve their workflows

Search Bar: To search for repositories, users, or topics.

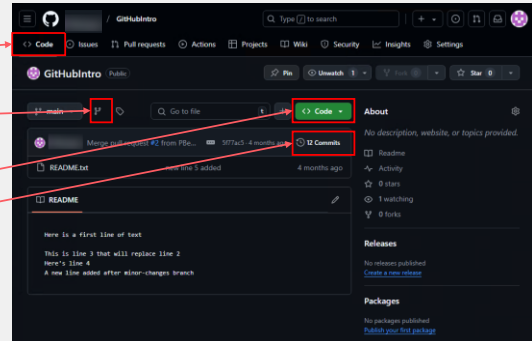
Notifications: Updates on issues, pull requests, and mentions.

Repositories: Places where code, files and each file's revision history are stored

## GitHub Repository Page

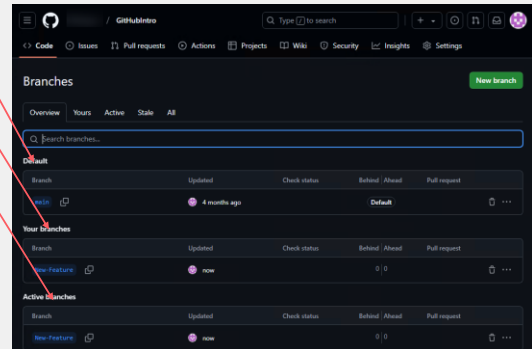
A repository (or "repo") is where code and project files are stored and managed. Features include:

- Code Tab: Shows the main files and folder structure.
- Branches: Switch between or create branches for parallel development.
- Code Dropdown: Allows cloning of the repository
- Commits: Tracks changes and who made them.



## GitHub Branches

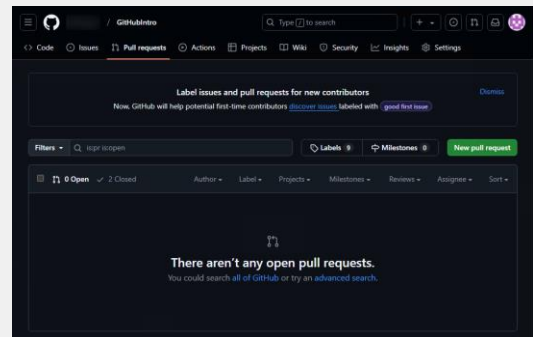
- Main Branch: Represents the stable version of the project.
- Feature Branches: For working on new features or fixes.
- Active Branches: Shows which branches are active



## GitHub Pull Requests (PRs)

Propose, review, and merge changes from one branch to another.

- Discussion and Review: Add comments or request changes.
- Merge Process: Show how changes are merged into the main branch.



## 05 Repository Creation and Setup

GitHub Essentials



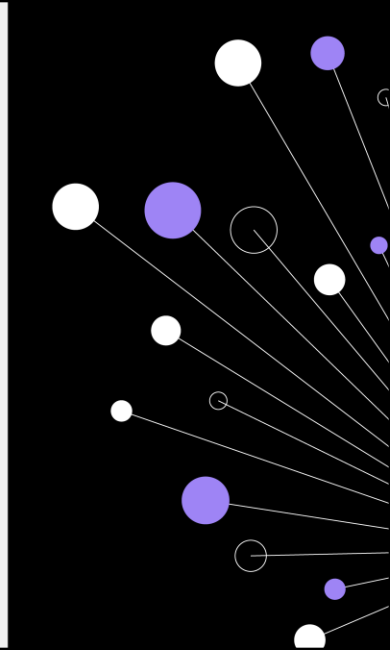
```
the text runs across the top
// persisted properties
<html> <p style="font-weight:bold;">HTML font code is displayed
<html> <body style="background-color:yellowgreen;">
<html> <div style="text-align:center;">
  // Non - text - :200px;"persisted properties
  <html> <errorMessage = ko , observable() ; ko
  <p style="color:orange;">HTML font code is displayed
  function todoitem(data) ;
    var self = this <html> <errorMessage = ko
    data = data || {} <html> <errorMessage = ko
  // Non - persisted properties function
  <html> <errorMessage = text - :200px;"ko , observable() ; ko
  <p style="font-weight:bold;">HTML font code is displayed
  <body style="background-color:yellowgreen;">
    text - :200px;"> <todoitemid = data.todoitemid ;
    - text - :200px;"persisted properties
    <errorMessage = ko , observable() ; ko
```



## Learning objectives

- Know how to create a new repository in GitHub, clone it locally, and set up a project structure.

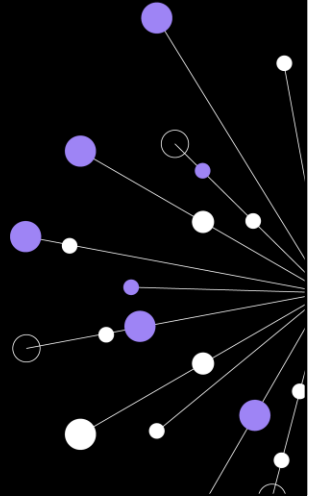
QA



## Session Content

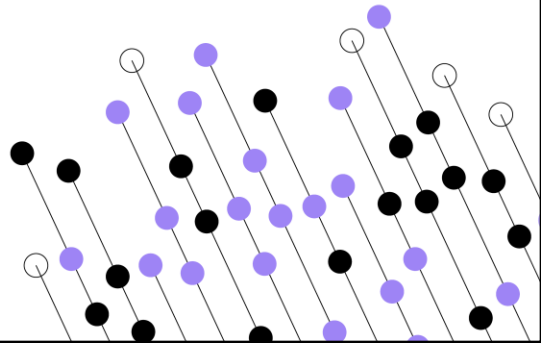
- Creating a new repository on GitHub
  - Adding a README file
  - Initializing a .gitignore file
- Cloning the repository locally
- Working with Remote Repositories
- Setting up your project structure

QA



# Creating a new repository on GitHub

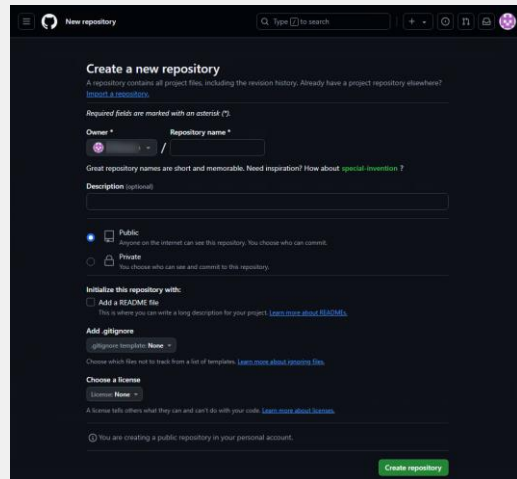
Repository Creation and Setup



# Steps to Create a New Repository in GitHub - 1

- Log In to GitHub
- Navigate to the "New Repository" Page
  - Click on your profile picture in the top-right corner of the page.
  - From the dropdown menu, select "Your repositories".
  - Click the green "New" button on the right side of the repositories page.
  - Alternatively, click the "+" icon in the top-right menu and select "New repository".

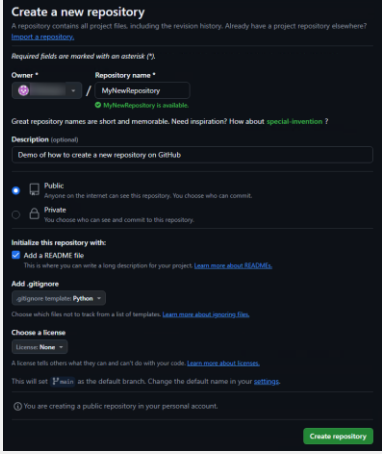
QA



The screenshot shows the GitHub 'Create a new repository' page. At the top, there's a header with the GitHub logo and a search bar. Below the header, the page title is 'Create a new repository'. A sub-header explains that a repository contains all project files, including the revision history, and provides a link to 'Browse all repositories'. The form includes fields for 'Owner' (with a dropdown menu), 'Repository name' (with an asterisk indicating it's required), and 'Description' (optional). There are radio buttons for 'Public' (selected) and 'Private'. Below this, there's a section 'Initialize this repository with:' with a checkbox for 'Add a README file'. There's also a section 'Add .gitignore' with a dropdown menu. At the bottom, there's a 'Choose a license' section with a dropdown menu. A green 'Create repository' button is at the bottom right. A note at the bottom left states 'You are creating a public repository in your personal account.'

## Steps to Create a New Repository in GitHub - 2

- Fill in Repository Details
  - Repository Name
  - Description (optional)
  - Visibility
    - Public (to anyone)
    - Private (to you and collaborators)
- Initialize Repository Settings
  - Initialize with a README
  - Add .gitignore (optional but **very** important)
  - Choose a licence (optional)
- Create the Repository
  - Click the green button



The screenshot shows the 'Create a new repository' page on GitHub. It includes fields for 'Owner' (selected as 'MyNewRepository'), 'Repository name' (with a green checkmark indicating availability), and a 'Description' field. Under 'Initialize this repository with:', the 'Add a README file' checkbox is checked. Below that, the 'Add .gitignore' section shows a dropdown menu with 'Python' selected. The 'Choose a license' section has a dropdown menu with 'None' selected. A green 'Create repository' button is at the bottom right.

QA

### Repository Details

- **Repository Name:**
  - Enter a unique and descriptive name for your repository.
- **Description (Optional):**
  - Add a short description of what the repository is about.
- **Visibility:**
  - **Public** so anyone can view the repository.
  - **Private** so only you and collaborators can see it.

### Repository Settings

- **Initialize with a README:**
  - Check this box to include a basic README file, which is typically the first file people see when visiting your repository. You will need to add your own text to this file.
- **Add .gitignore (Optional):**
  - Select a .gitignore template to exclude specific files from version control (e.g., environment variables, build files). **Very important** to ensure only source code (rather than built code and any dependencies) is pushed to the repository.

- **Choose a License** (Optional):
  - Select an open-source license for your project (e.g., MIT, Apache).

## Ignoring Files

Git categorises all files in your local repository into:

**Tracked** - files which have been staged or committed

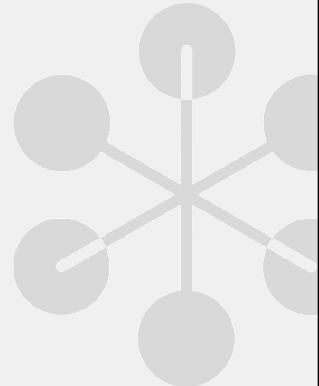
**Untracked** - files which have not been staged

**Ignored** - files which will be ignored

We often do not want some files in our remote repository. e.g. compiled code, build output directories, dependency caches.

In order to make sure that Git does not track them, we need to create a **.gitignore** file in our repository.

There are sets of standard prepopulated .gitignore files for different programming languages.



**QA**

Git categorises all files in your local repository into "**tracked**" (files which have been staged or committed), "**untracked**" (files which have not been staged), or "**ignored**" (files which will be ignored).

We often do not want some files in our remote repository. This could be because they are large or unnecessary (e.g. compiled code, build output directories, dependency caches). In order to make sure that Git does not track them, we need to create a **.gitignore** file in our repository.

This file is a list of **intentionally untracked** files that Git should ignore. We can use an asterisk "\*" to specify which type of files should be ignored (an asterisk matches anything except a slash).

For example, if we use "\*.pyc", Git will ignore all files that end in ".pyc".

A leading "\*\*/" means match in all directories. For example, if we use "\*\*/venv", Git will ignore a file or directory "venv" anywhere in our repository.

Our **.gitignore** file may look like one of these examples:

```
# gitignore file example 1
```

```
**/venv
```

```
*.pyc
```

```
__pycache__
```

```
# gitignore file example 2
```

```
.settings/
```

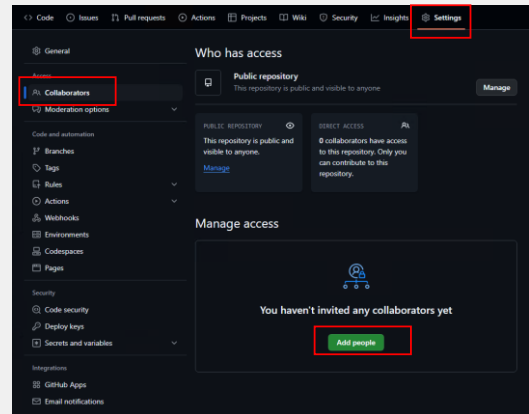
```
target/
```

```
bin/
```

## Adding Collaborators

You can optionally add collaborators by:

- Navigate to the repository's **Settings** > **Collaborators**



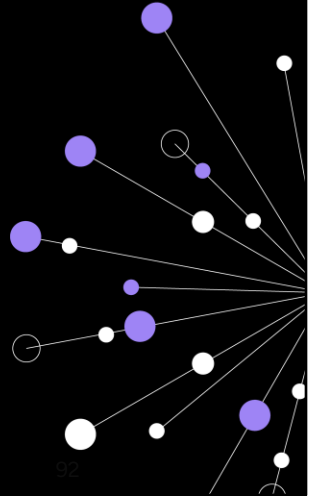
QA



## Quick Lab – Create a GitHub Repository

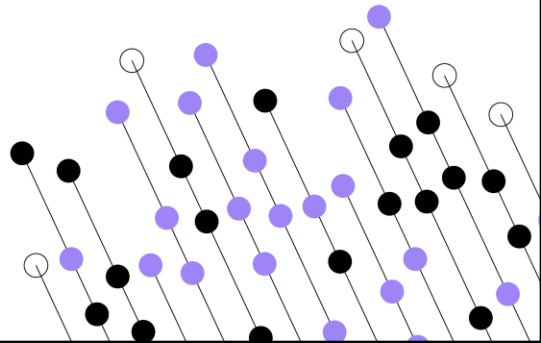
- Navigate to GitHub.com
- Create a new remote repository called git-practice
  - Ensure the repository has a README file
  - Ensure the repository has a .gitignore file for Python

QA



## Cloning the repository locally

Repository Creation and Setup



## Attack of the Clones!

Cloning makes a physical copy of a Git repository

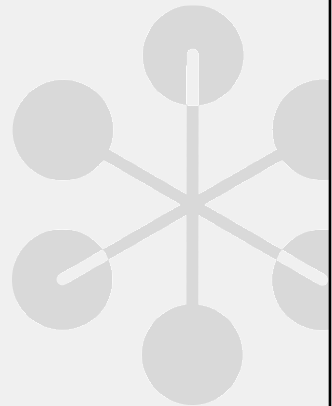
- It can be done locally or via GitHub
- You can push and pull updates from the repository

The benefit of cloning repos is that the commit history is maintained

- Changes can be sent back between the original and the clone

Cloning is achieved:

- From GitHub by Using the Code dropdown button on the repository page to copy the HTTPS, SSH, or GitHub CLI link.
- And then using Git's clone command



## Cloning a repository

Cloning copies the entire repository to your hard drive

- The full commit history is maintained

To clone a remote repository:

```
$ git clone [repository]
```

```
$ git clone https://github.com/username/repositoryname
```

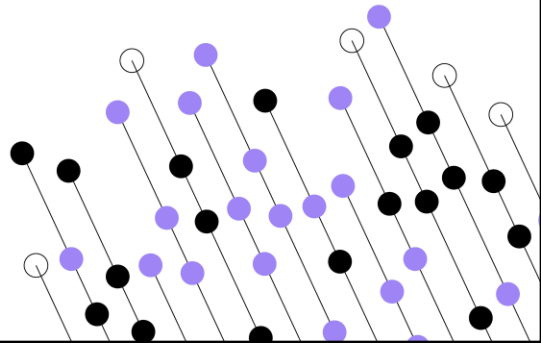
To clone a specific branch:

```
$ git clone -b branchName repositoryAddress
```

**QA**

## Working with Remote Repositories

Repository Creation and Setup



## Working with remote repositories

To see configured remote repositories on GitHub from a local Git installation run the `git remote` command

- If you have cloned a repository, you should see the origin.

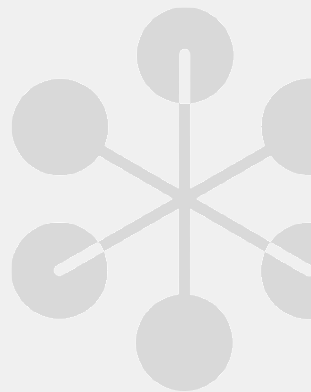
To see the actual url being pointed to use:

```
git remote -v
```

To add a remote repository use:

```
git remote add <shortname> <url>
```

- Where shortname is an alias for the url



QA

97

You can also specify `-v`, which shows you the URL that Git has stored for the short name to be expanded to.

If you want to rename a reference, in newer versions of Git you can run `git remote rename` to change a remote's shortname.

```
$ git remote rename pb dave
$ git remote
origin
dave
```

It's worth mentioning that this changes your remote branch names, too. What used to be referenced at `pb/master` is now at `dave/master`.

If you want to remove a reference for some reason — you've moved the server or are no longer using a particular mirror, or perhaps a contributor isn't contributing anymore — you can use `git remote rm`:

```
$ git remote rm dave
$ git remote
```

origin

## Pushing to the repository

To update local changes to the remote GitHub repository, use the `push` command:

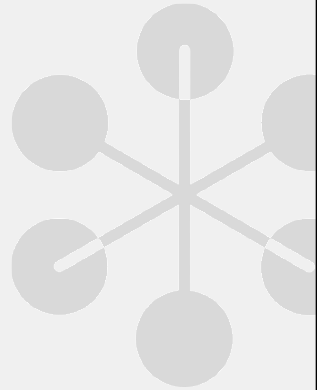
```
git push <remote> <branch>
```

To set the branch to automatically push you can use:

```
git push -set-upstream <remote> <branch>
```

If the remote path has already been configured you can simply invoke:

```
git push
```





## Pulling the repository

To pull all the changes to the local repository you use the `pull` command

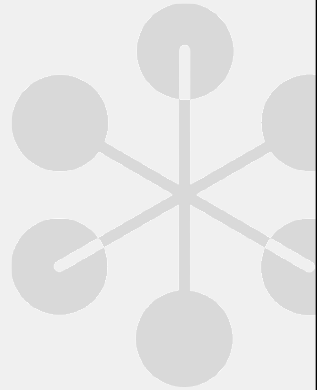
```
git pull <remote> <branch>
```

It is good practice to pull the repository before pushing changes

- You can get an up-to-date copy of the repo to push to
- You can see any conflicts before they are pushed
- You can stash your changes locally before pulling the remote branch

If the remote path has already been configured you can simply invoke:

```
git pull
```



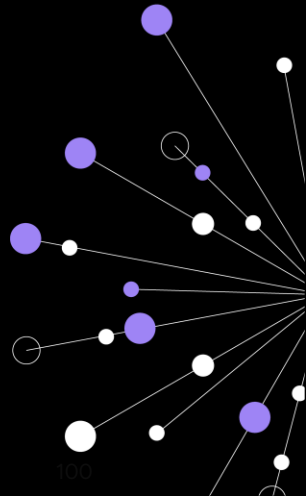
## Quick Lab – Clone the Repository

Clone the GitHub git-practice repository you created earlier and create a local copy of it

- In GitHub locate the git-practice repository and click on the Green "Code" button
- Copy the HTTPS URL
- In Git Bash on your local machine enter and run the following:  

```
git clone <url>
```
- Confirm the README and .gitignore files have been retrieved by running `ls -a`

QA



## O6 Linking to VSCode

GitHub Essentials

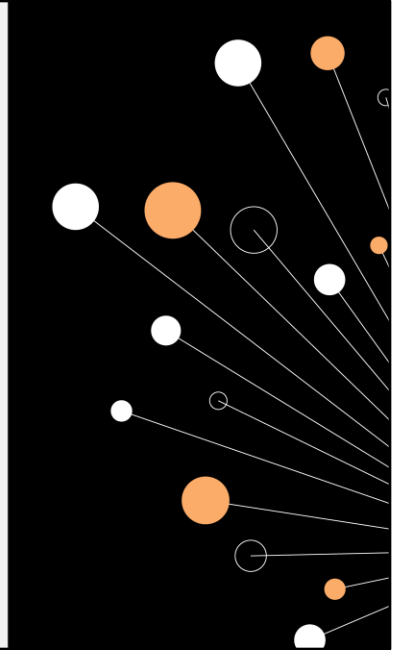


```
the text runs across the top
// persisted properties
<html> <p style="font-weight:bold;">HTML font code is displayed
<html> <body style="background-color:yellowgreen;color:white
<html> <text - :200px;"> <todoId = data.todoId; todoId
// Non - text - :200px;" persisted properties
<html> <errorMessage = ko , observable() ; ko
<p style="color:orange;">HTML font code is displayed
function todoItem(data) ;
var self = this <html> <errorMessage = ko
data = data || {} <html> <errorMessage = ko
// Non - persisted properties function
<html> <errorMessage = text - :200px;">
<p style="font-weight:bold;">HTML font code is displayed
<body style="background-color:yellowgreen;color:white
text - :200px;"> <todoId = data.todoId; todoId
- text - :200px;" persisted properties
<errorMessage = ko , observable() ; ko
```

## Learning objectives

- Understand how IDEs (using Visual Studio Code (VSC) as an example) have built in functionality to expedite the Git process
- Know how to log in to GitHub and use tools to create and update repositories

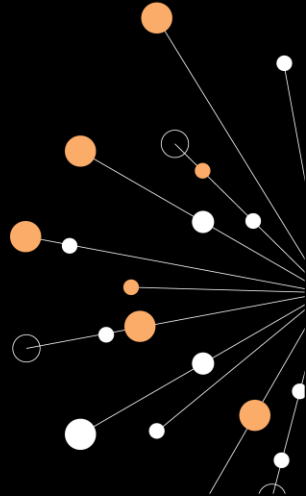
QA



## Session Content

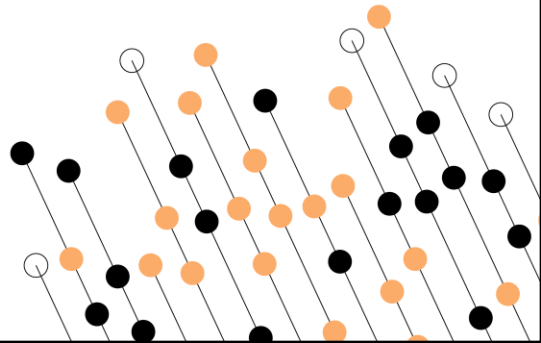
- Linking VSCode to GitHub
- Use tools to create and update repositories
- Install extensions to assist in the process
- Configure VSCode for git operations

**QA**



## Linking VSCode to GitHub

Linking to VSCode



## Git and Visual Studio Code(VSC)

VSC has Git support built in.

Features include:

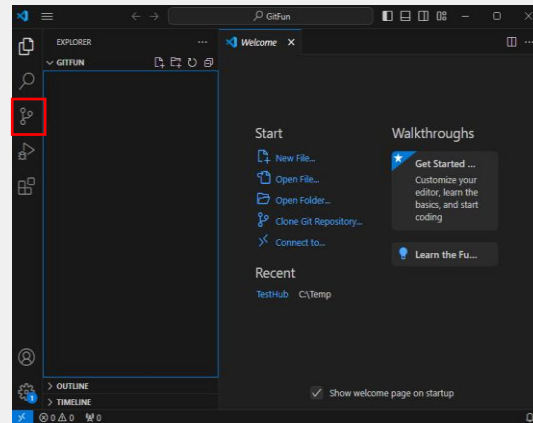
A Git Status Bar shows, branch, dirty indicators, incoming and outgoing commits

Can do most common Get operations from within the editor:

- Initialize a repository
- Clone
- Stage and Commit
- Push and Pull
- Create branches
- Resolve conflicts
- View diffs

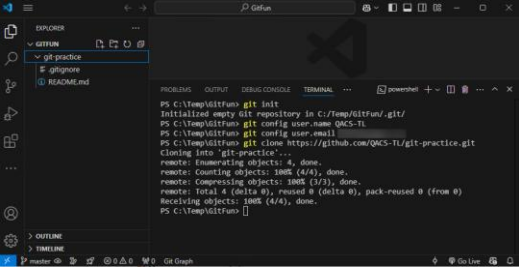
Extensions can be added (see later)

**QA**



## Log onto Git repository using VSC

- Open VSC
- Open a Folder where code is to live
- Open a Terminal Window (via View menu)
- Type in the following commands:  
`git init`  
`git config user.name <username>`  
`git config user.email <email>`



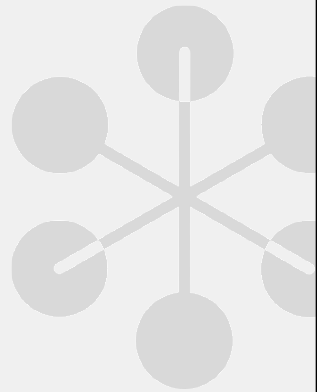
The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar shows a folder named 'GITFUNK' containing 'git practice', 'gitignore', and 'README.md'. The main editor area is open to 'git practice'. A terminal window is open at the bottom, displaying the following commands and output:

```
PS C:\Temp\GitFun> git init
Initialized empty Git repository in C:\Temp\GitFun\.git\
PS C:\Temp\GitFun> git config user.name QACS-TL
PS C:\Temp\GitFun> git config user.email
PS C:\Temp\GitFun> git clone https://github.com/QACS-TL/git-practice.git
Cloning into 'git-practice'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (4/4), done.
PS C:\Temp\GitFun>
```



## Cloning Repos using VSC

- Browse to GitHub repository
- Click on green "Code" button
- Copy HTTPS clone url
- Goto VSCode terminal window and enter:  
`git clone <clone url>`

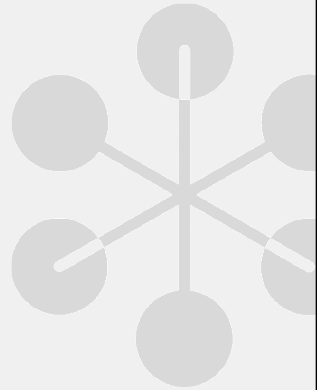


## Creating New Repos from Local Code using VSC

- Open Your (Python) Project in VS Code
- Open the Terminal in VS Code
- Initialize a Git Repository using `git init`
- Add Your Files to the Staging Area using `git add .`
- Make the Initial Commit using `git commit -m "Initial Commit"`
- Create a New Repository on GitHub
- Link Your Local Repository to GitHub
  - Copy the Repository URL
  - In VSC's terminal add the Remote Repository  
`git remote add origin <repository-url>`
- Push Your Code to GitHub



Confirm on GitHub



112

When pushing the code to GitHub you may need to use the following:  
`git push --set-upstream origin main`

## Staging and Committing Changes using VSC

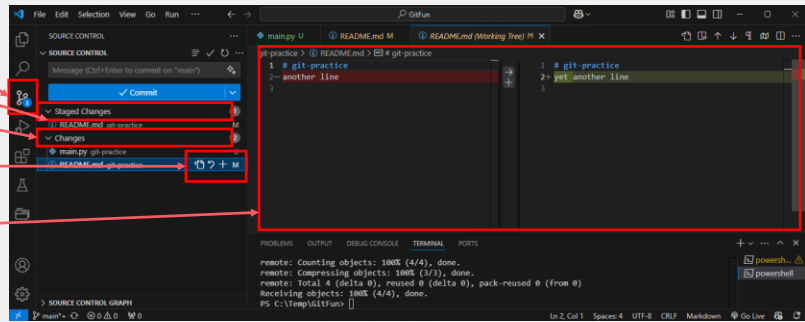
Source Control icon

Staged items

Changed items

Open File, Discard  
Changes, Stage, View  
Modifications

Original vs Modified



QA

113

### Open the Source Control Panel:

1. Click on the **Source Control** icon in the Activity Bar on the left-hand side of the screen
2. If your folder is not yet a Git repository, you will see an option to initialize it:
  1. Click **"Initialize Repository"** to run `git init` in your project folder.

### Staging:

#### 1. View Changes:

1. In the Source Control panel, you'll see a list of files with changes.
2. Modified files will appear under **Changes**.

#### 2. Stage Changes:

1. Hover over the file you want to stage.
2. Click the + icon that appears next to the file name to stage it. Or right-click the file and select **"Stage Changes"**.
3. To stage all changes at once, click the + icon next to the **Changes** heading.

### Committing:

#### 1. Write a Commit Message:

1. Locate and select the "Message" text box at the top of the Source Control panel

2. Type a meaningful commit message describing the changes.
- 1.Commit the Changes:
1. Press the Commit button to commit the staged changes. Or click the checkmark icon ✓ at the top of the Source Control panel.

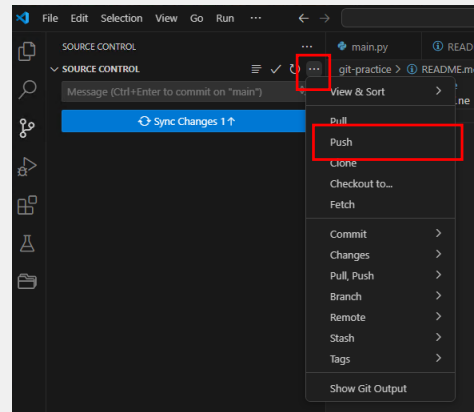
## Using VSC to Push (and Pull) from GitHub

Once all changes have been committed and your repository is connected to GitHub. Push the changes by:

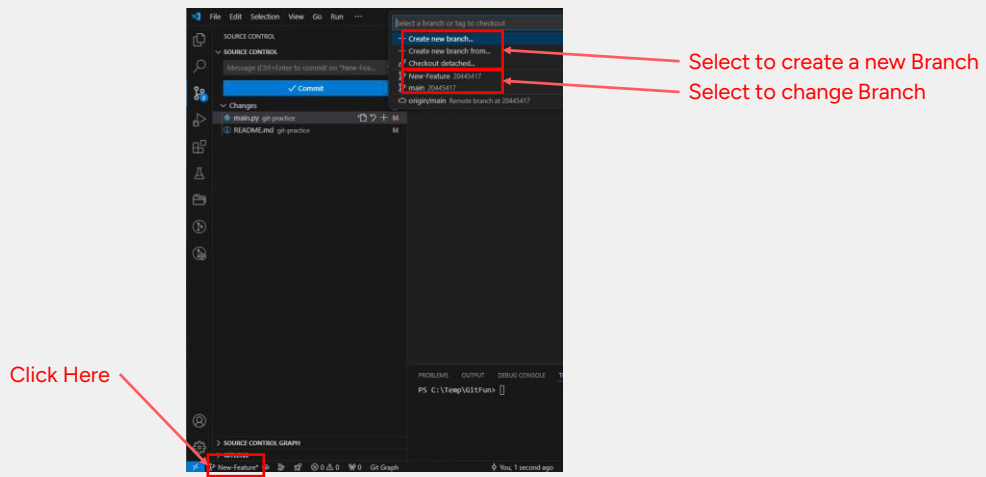
- Clicking the ... menu in the Source Control panel and selecting "Push".

OR

- Press the Sync Changes button
  - Note this will carry out a Pull followed by a Push to ensure no issues occur when the merge takes place.



## Creating and Changing Branches in VSC



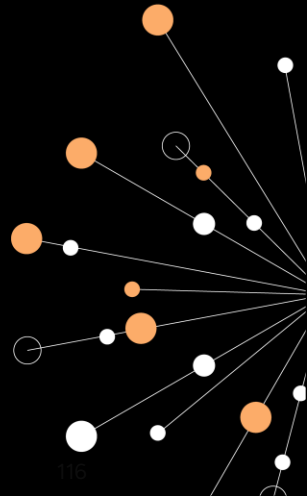
QA

115

## Quick Lab – Create and Manage Repos from VSC

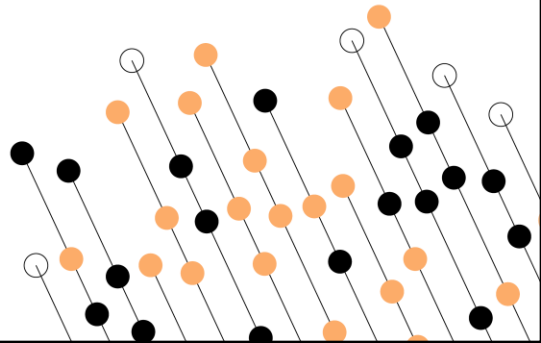
- Log into VSCode
- Clone and open a repo
- Create a new repo from local code in VSCode
- Push and pull from remote sources

QA



## Useful VSC Extensions

Linking to VSCode





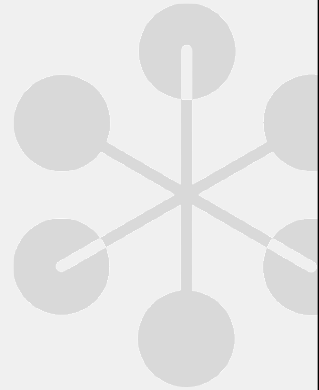
## Useful Visual Studio Code Extensions

**Git Blame:** See Git Blame information in the status bar and on the last selected line in your editor.

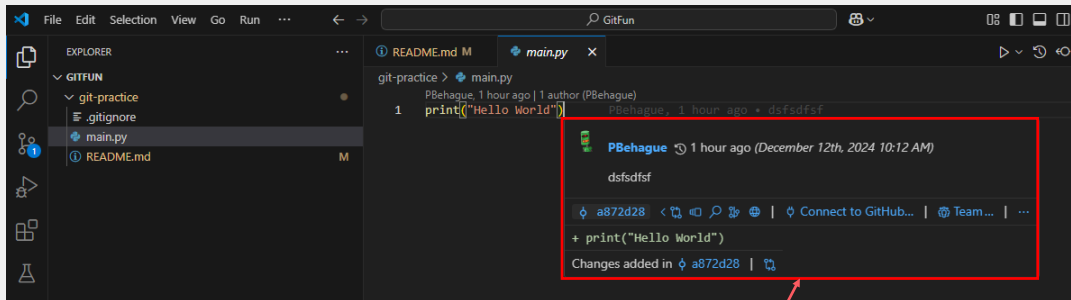
**Git Extension Pack:** Packages some of the most popular Git extensions.

- **Open in GitHub, Bitbucket, GitLab:** Use to jump to a source code line in Github, Bitbucket, Visualstudio.com and GitLab
- **GitLens:** Helps you to visualize code authorship at a glance
- **Git History:** View git log, file history, compare branches or commits

**Git Graph:** See a Git Graph of your repository and perform Git Actions (commands) by right clicking on commit, branch or tag

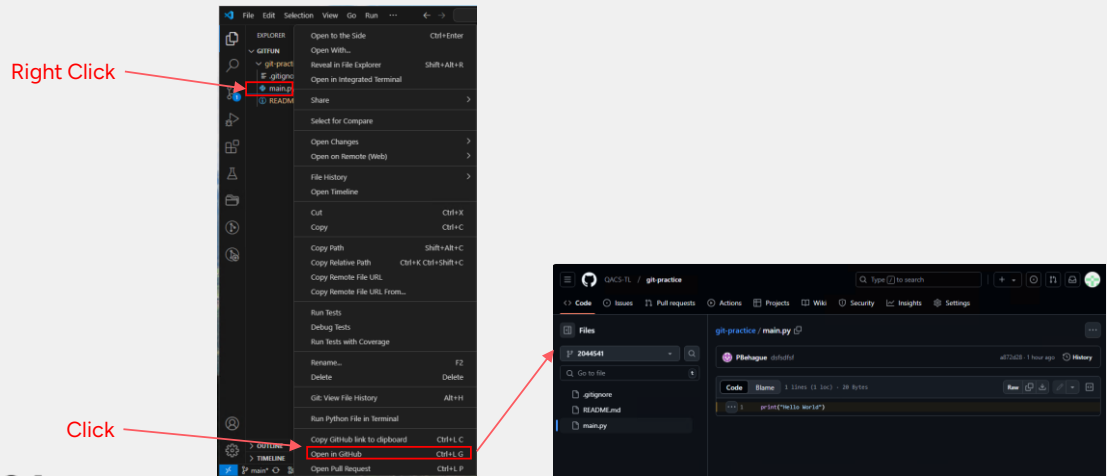


## Git Blame



Shows information about the last selected line in your editor, including the author, date and time change was made, etc.

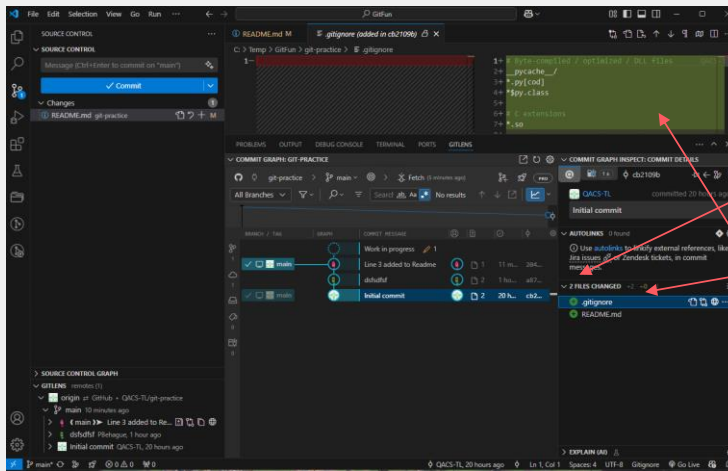
## Open in GitHub, BitBucket, GitLab



QA

120

## Git Lens – Commit Graph



Shows all commits

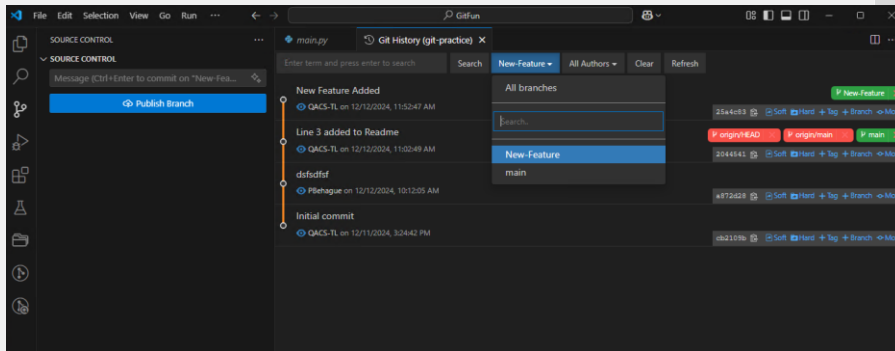
Shows what changed

Shows the changes

QA

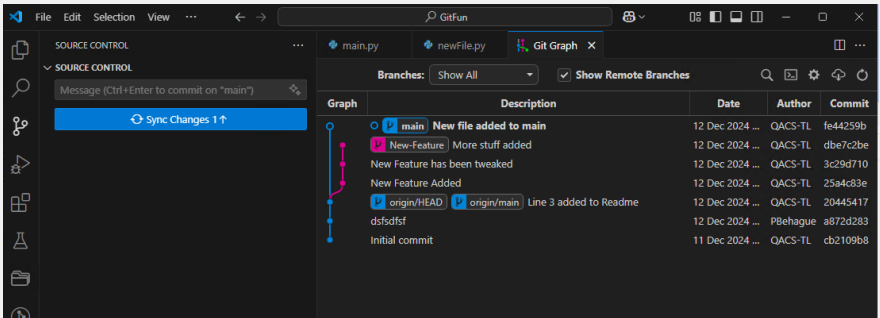
# Git History

Shows who did what on which branch



# Git Graph

Visually shows commits and branches, dates and authors

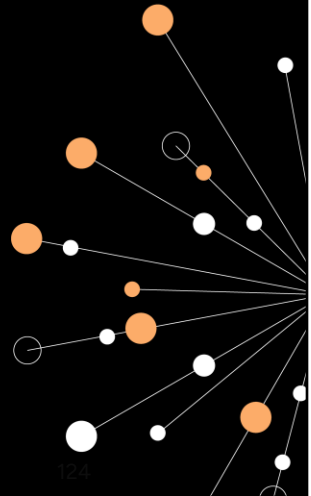


QA

## Quick Lab – Add extensions to VSC

Add and use some of the extensions mentioned above and try them out.

QA



## 07 Pull Requests

GitHub Essentials



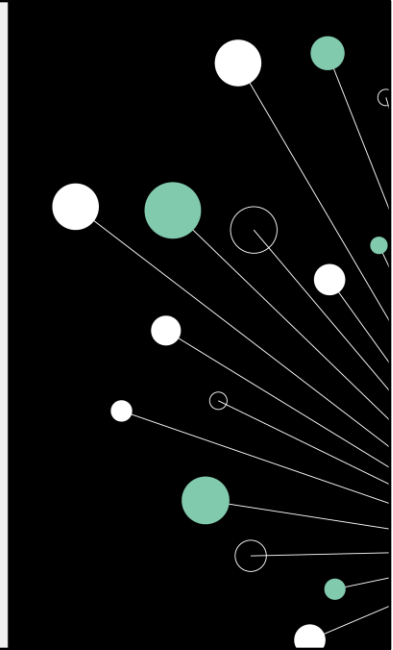
```
the text runs across the top
// persisted properties
<html> <p style="font-weight:bold;">HTML font code is displayed
<html> <body style="background-color:yellowgreen;">
<html> <div style="background-color:yellowgreen;">
// Non - text - :200px;" persisted properties
<html> <errorMessage = ko , observable() ; ko
<p style="color:orange;">HTML font code is displayed
function todoitem(data) ;
var self = this
data = data || {}
// Non - persisted properties
<html> <errorMessage = text - :200px;"
<p style="font-weight:bold;">HTML font code is displayed
<body style="background-color:yellowgreen;">
text - :200px;" <todoId = data.todoId
- text - :200px;" persisted properties
errorMessage = ko , observable() ; ko
```



## Learning objectives

Understand the concept of pull requests and know how to use them for code review and collaboration.

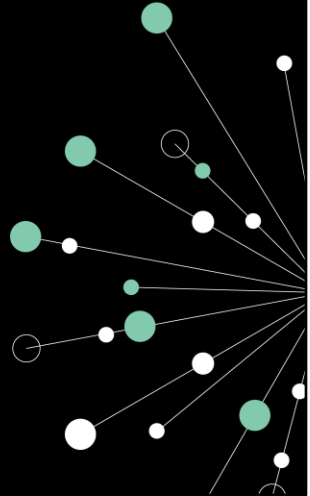
QA



## Session Content

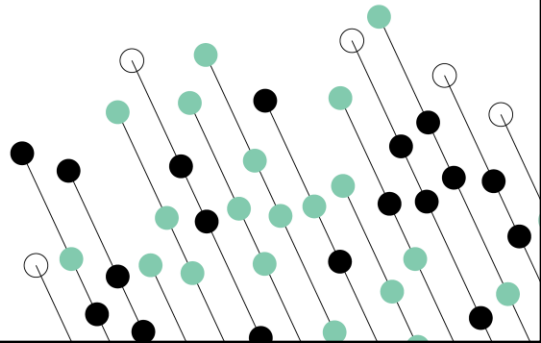
- What is a pull request?
- Creating a pull request
- Reviewing and discussing code changes
- Merging pull requests
- Closing pull requests

**QA**



## What is a pull request?

Pull Requests

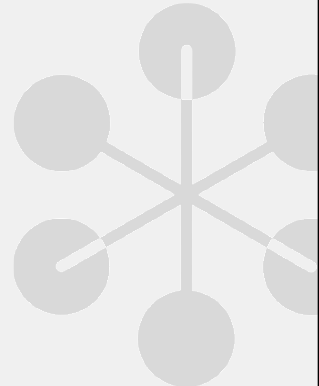


## Pull Requests

A pull request (PR) is a feature used in Git-based platforms such as GitHub.

- Allow developers to propose changes they've made in a repository branch and request that those changes be reviewed and merged into another branch (usually the main or master branch).
- The term "pull request" comes from the idea of asking the repository owner or collaborators to pull your changes into the target branch.

Despite the name, you're not pulling anything — you're actually requesting that your changes be pulled into another branch, usually the main or develop branch.

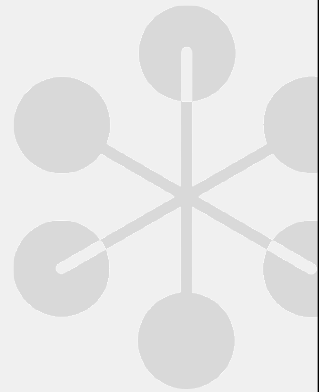


**QA**

## Why Use Pull Requests?

PR's are essential for collaborative development and are used for:

- Code Reviews
- Collaboration
- Testing and Validation
- Audit Trail
- Safe Integration
- Feedback and Learning



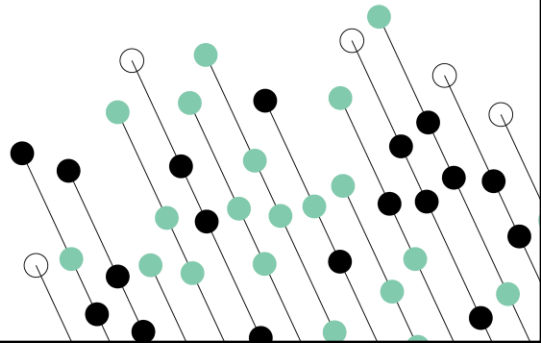
QA

132

- **Code Review:** PRs allow other team members or collaborators to review your code and suggest improvements/find bugs to ensure code quality before it's merged.
- **Collaboration:** Multiple people can collaborate on a feature branch by adding comments to the pull request.
- **Testing and Validation:** A pull request can be used to trigger automated tests or Continuous Integration (CI) workflows to ensure new code doesn't break the project.
- **Audit Trail:** PRs are useful for accountability and understanding why specific changes were made because they provide a history of changes and the context behind them.
- **Safe Integration:** By working in separate branches and merging changes through PRs, the main branch remains stable. Thus protecting projects from being incomplete or containing buggy code.
- **Feedback and Learning:** Junior developers or new team members can learn from experienced developers through pull request reviews and feedback.

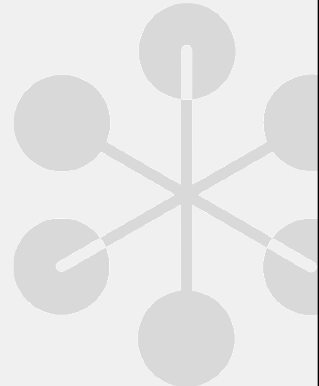
## Creating a pull request

Pull Requests



## Pull Request Walkthrough – Branch Creation and Push

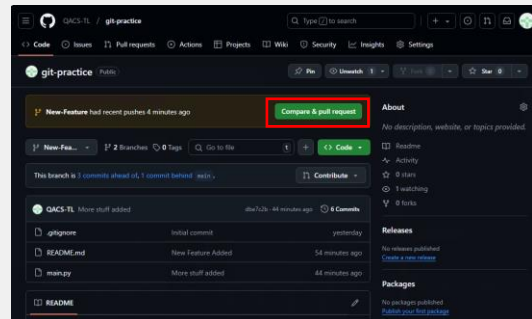
- Create a branch for a new feature or bug fix  
`git checkout -b feature-branch`
- Make changes and commit code
- Publish the new branch by pushing it to GitHub  
`git push origin feature-branch`



**QA**

## Pull Request Walkthrough –Triggering a Compare and Pull Request

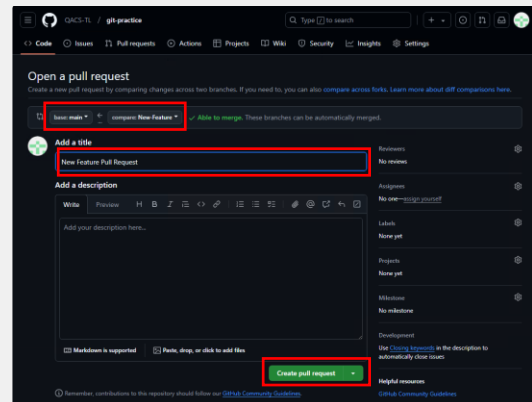
On GitHub open a Pull Request by navigating to the **new** branch and clicking the Compare & Pull Request button





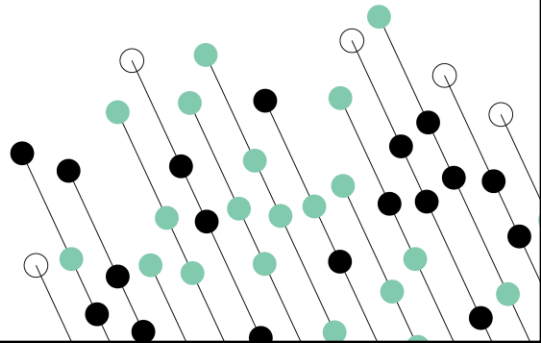
## Pull Request Walkthrough – Opening a Request

- Ensure branch you want to merge to (typically the repository's main branch) is correctly selected
- Give request a title and optionally a description
- Click Create Pull Request button



## Reviewing and Discussing Code Changes

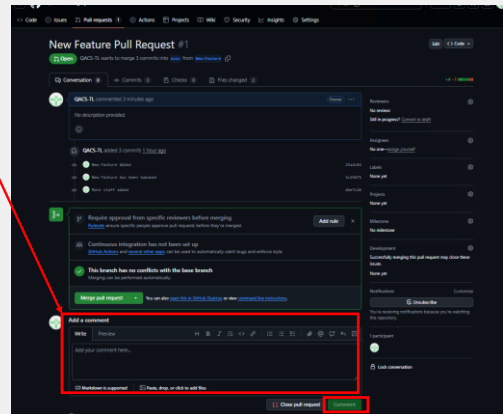
Pull Requests



## Pull Request Walkthrough – The Review Page

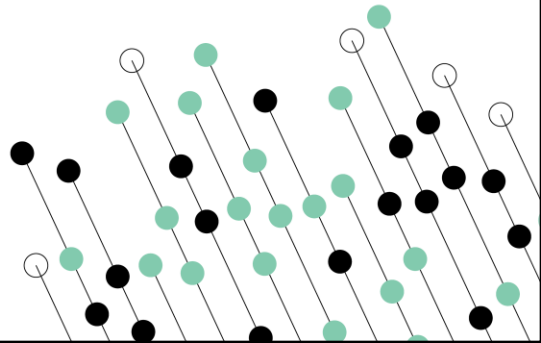
**The Review Page** shows high-level overview of the changes between your branch and the main (master) branch

**Review Process:** Team members or maintainers review the pull request, leave comments, and suggest changes.



## Merging Pull Requests

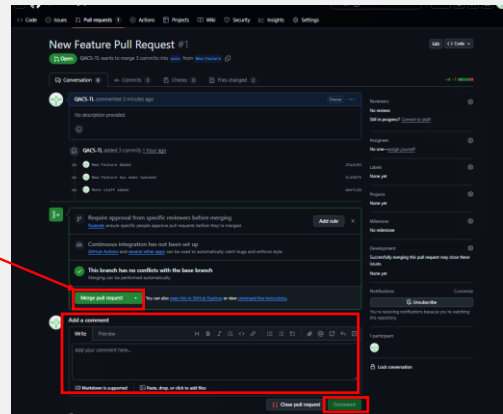
Pull Requests



## Pull Request Walkthrough – Merging Branches

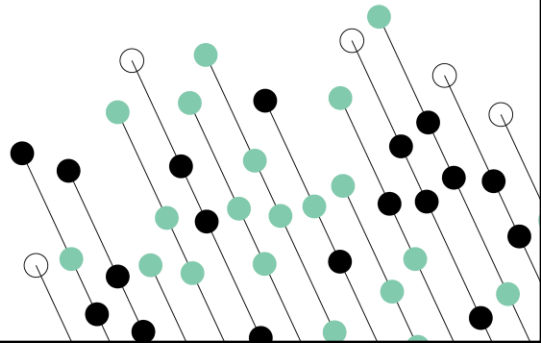
### Merge the Pull Request:

- Once approved, the pull request is merged into the target branch.
- You can delete the feature branch after merging to keep the repository clean.



## Closing Pull Requests

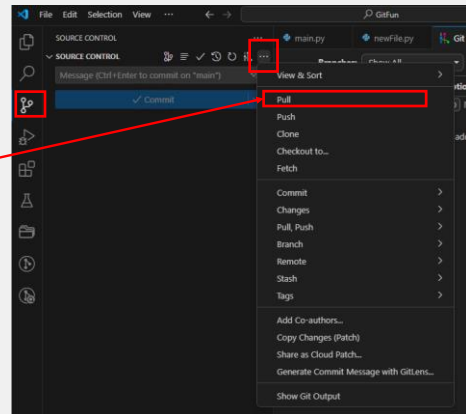
Pull Requests



## Pull Request Walkthrough – Tidying up the Local Git Repo

If the pull request was successful and the branch merged and deleted on GitHub you will need to get the local Git repository back in step:

- In VSC make sure you are working with the main branch (i.e. the one your branch was merged into)
- In the Source Control panel click on the period ellipses and select the Pull menu option
- Make sure the branch you want to delete is **not** active (select main in the bar bottom left of VS's status bar)
- Press F1 to open VSC's command palette and type Delete Branch and select Git: Delete Branch...
- Select the branch you want to delete and press enter



QA

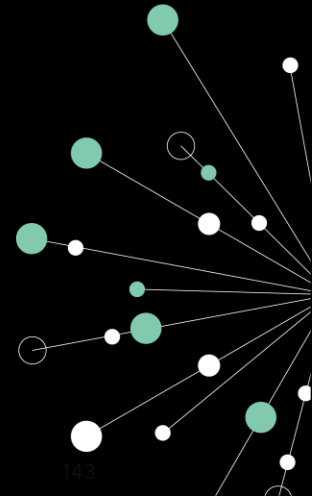
142

You can also use the **Source Control View** to delete a branch. From the Source Control panel click on the period ellipses and select Branch. Then Delete Branch... Then select the branch you want to delete from the list that appears at the top of VSC.

## Quick Lab – Carry out a Pull Request

- Clone the Repository in VSC
- Create a New Branch
- Make Changes and Add a Feature
- Stage and Commit the Changes
- Push the Branch to GitHub
- Create a Pull Request on GitHub
- Review and Merge the Pull Request
- Delete the Feature Branch (on GitHub and locally)
- Pull the Latest Changes to Main

QA





## 08 Managing Merge Conflicts

GitHub Essentials

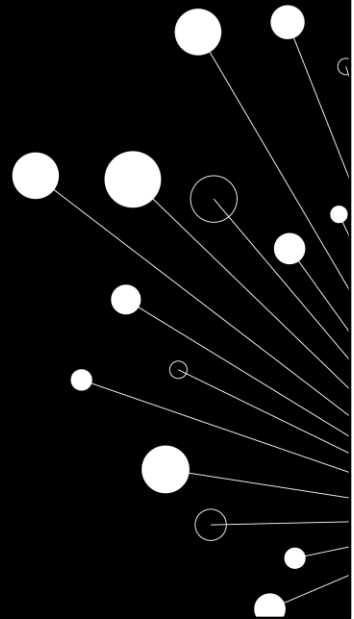


```
the text runs across the top
// persisted properties
<html> <p style="font-weight:bold;">HTML font code is displayed
<html> <body style="background-color:yellowgreen;">
<html> <div style="background-color:yellowgreen;">
// Non - text - :200px;" persisted properties
<html> <errorMessage = ko , observable() ; ko
<p style="color:orange;">HTML font code is displayed
function todoitem(data) ;
var self = this
data = data || {}
// Non - persisted properties
<html> <errorMessage = text - :200px;"
<p style="font-weight:bold;">HTML font code is displayed
<body style="background-color:yellowgreen;">
text - :200px;" <todoId = data.todoId
- text - :200px;" persisted properties
errorMessage = ko , observable() ; ko
```

## Learning objectives

- Understand common merge conflicts and know how to resolve them effectively.

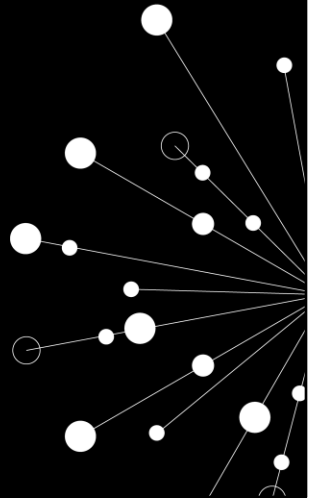
**QA**



## Session Content

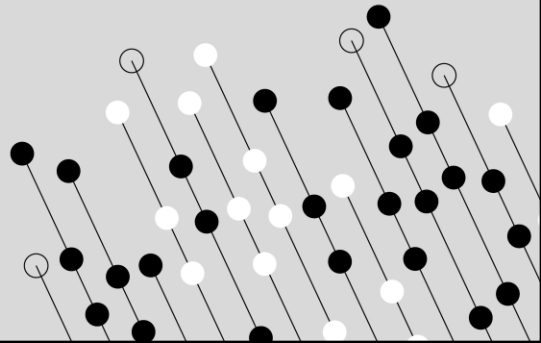
- Understanding merge conflicts
- Identifying conflicts in your code
- Using Git tools to resolve conflicts
- Committing resolved conflicts
- Best practices for avoiding merge conflicts

**QA**



## Understanding merge conflicts

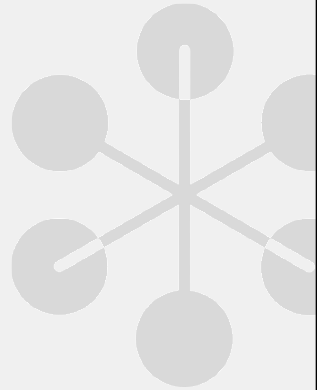
Managing Merge Conflicts



## Failed Merges

Merges can fail because

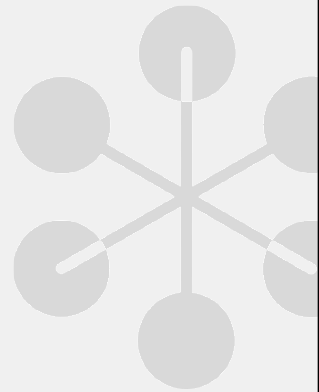
- There are uncommitted changes in either branch
- There is a conflict with another developer's code



## Merge Conflicts

When multiple developers try to edit the same content (using different branches) there is a possibility a conflict may occur. E.G.

- Same lines altered in a file
- A file is deleted that another developer is editing
- When many branches are being merged, and changes are scattered across various files and lines
- Git's `merge` cannot automatically decide which of the changes is correct.
- Conflicts impact only the developer performing the merge
- Git flags the file as conflicted and stops the merge process, leaving it up to the developer to resolve the conflict manually.



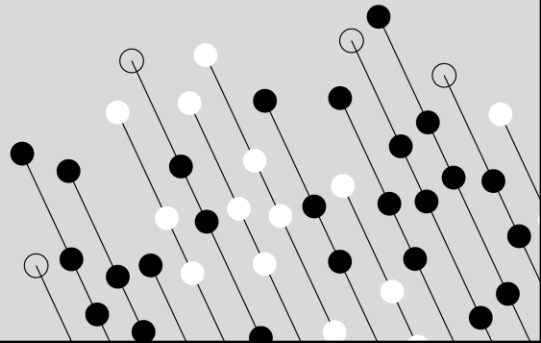
QA

151

Conflicts typically occur when two individuals modify the same lines in a file or when one person deletes a file that another is editing. In such situations, Git cannot automatically decide the correct changes. Conflicts impact only the developer performing the merge, while the rest of the team remains unaffected. Git flags the file as conflicted and pauses the merge process, leaving it up to the developer to resolve the conflict manually.

## Identifying conflicts in your code

Managing Merge Conflicts



## Identifying Merge Conflicts

On attempting to merge a branch called New-Feature with main where there are conflicts you will see an error message like the following:

```
PS C:\Temp\GitFun\git-practice> git merge New-Feature
Auto-merging newFile.py
CONFLICT (content): Merge conflict in newFile.py
Automatic merge failed; fix conflicts and then commit the result.
PS C:\Temp\GitFun\git-practice>
```

See what `git status` has to say

```
PS C:\Temp\GitFun\git-practice> git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
(use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   newFile.py

no changes added to commit (use "git add" and/or "git commit -a")
PS C:\Temp\GitFun\git-practice>
```

QA





## What Happened ?

```
$ git status
# On branch contact-form
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified:   contact.html
#       deleted by them: error.html
```

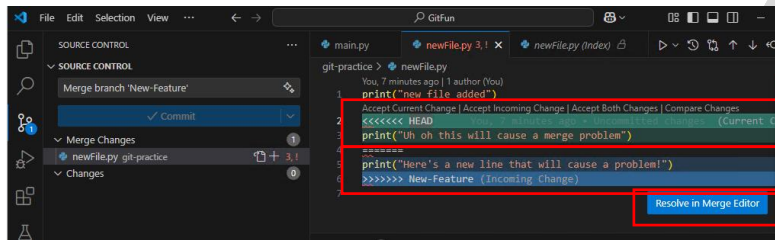
Two developers modified the same file on the same line(s).

The 95% case...

Other scenarios happen less frequently (here: one developer modified a file, while the other deleted it).

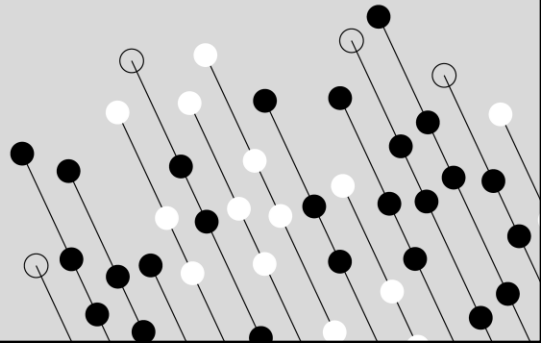
## VSCode Shows Both Versions of Modification

Once conflict is understood click on the Resolve in Merger Editor button

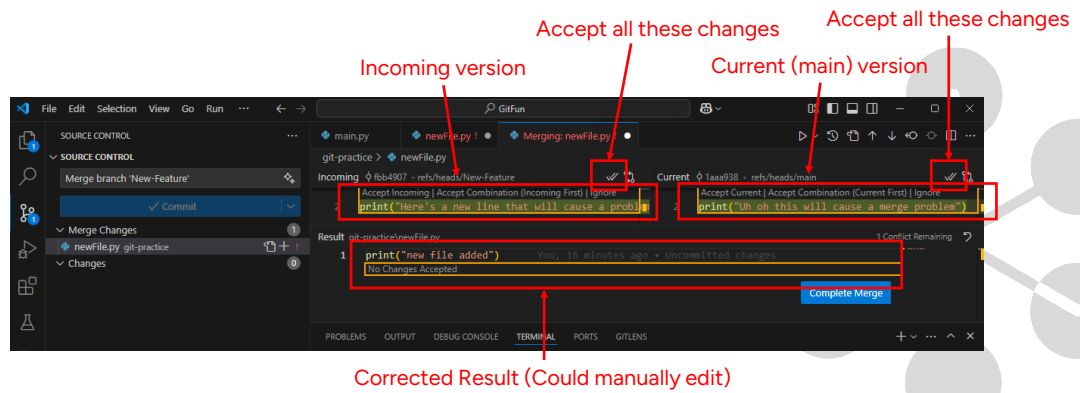


## Using Git tools to resolve conflicts

Managing Merge Conflicts

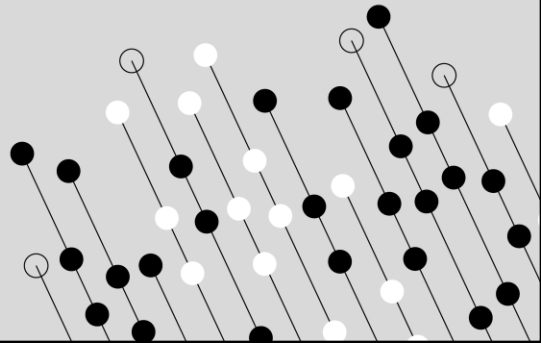


## The Resolve in Merger Editor Tool



## Committing resolved conflicts

Managing Merge Conflicts



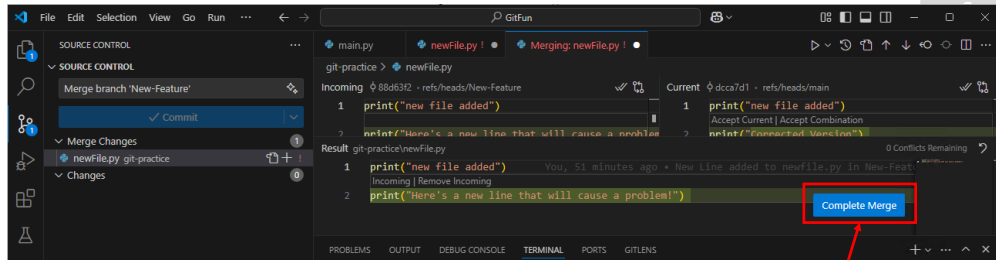
## Solve it

1. Save the resolved file.
2. Stage (git add) this change.
3. Commit like any other change.



= marks as resolved  
(might already be  
done by external tool)

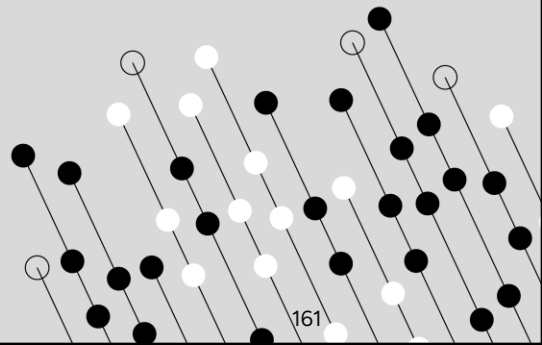
## Committing Resolved Conflicts



Click when result is finalised

QA

## Best Practice for avoiding Merge Conflicts



161



## Git Best Practice

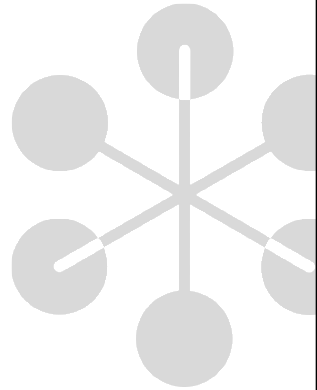
Commit often with clear messages.

Use branches for features or fixes.

Regularly pull changes to stay updated with the team.

Resolve conflicts promptly.

Always review code changes before merging (pull before you push)

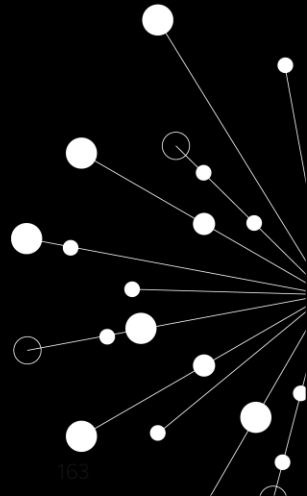


## Quick Lab – Managing Merge Conflicts 1

- Setup
  - Initialize a new repository
  - Create a file called data.txt with one line of text and make the first commit
- Create and modify two branches
  - Create and switch to a new branch called feature-branch
  - Make a change to data.txt in the feature-branch by adding a second line and commit
  - Switch back to main
  - Make a conflicting change to data.txt in the main branch by adding a second line and commit

Continued on next slide

**QA**



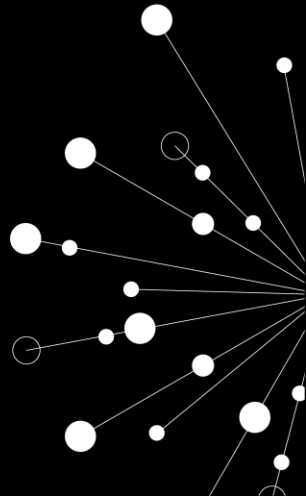
163

## Quick Lab – Managing Merge Conflicts 2

- Attempt to merge
  - Attempt to merge feature-branch into main
  - View the conflict message
- Resolve the conflict
  - Open the conflicted file
  - Examine the conflict markers
    - HEAD: Represents changes from the main branch.
    - Below =====: Changes from the feature-branch.
  - Resolve the conflict manually
    - Edit line 2 to be amalgamation of the alternatives
  - Save and close the file

Continued on next slide

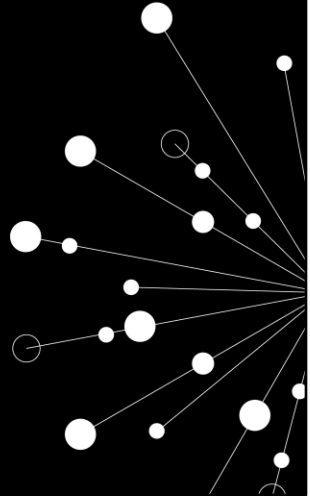
**QA**



## Quick Lab – Managing Merge Conflicts 3

- Complete the Merge
  - Add the resolved file
  - Commit the merge
- Verify the Merge
  - Check the log
  - Inspect the data.txt file

**QA**



## 09 Review and Q&A

GitHub Essentials

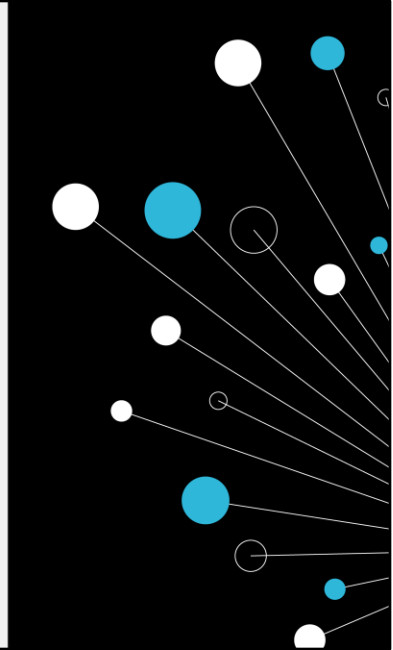


```
the text runs across the top
// persisted properties
<html> <p style="font-weight:bold;">HTML font code is
<html> <body style="background-color:yellowgreen;">
<html> <div style="background-color:yellowgreen;">
// Non - text - :200px;" persisted properties
<html> <errorMessage = ko , observable() ; ko
<p style="color:orange;">HTML font code is
function todoitem(data) ;
var self = this
data = data || {}
// Non - persisted properties
<html> <errorMessage = text - :200px;"
<p style="font-weight:bold;">HTML font code is
<body style="background-color:yellowgreen;">
text - :200px;" <todoitemid = data.todoitemid
- text - :200px;" persisted properties
errorMessage = ko , observable() ; ko
```

## Learning objectives

- Be reminded of key concepts of the day
- Q&A session to address any remaining questions.

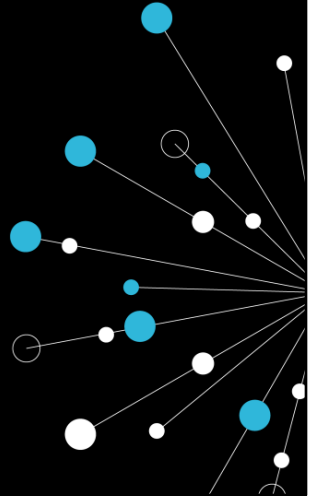
**QA**



## Session Content

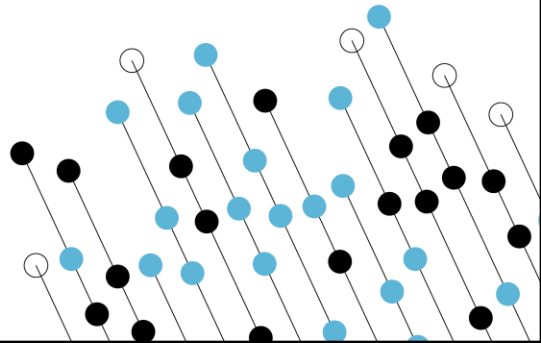
- Recap of key concepts
- Q&A session
- Next steps and additional resources

**QA**



## Recap of Key Concepts

Review and Q&A





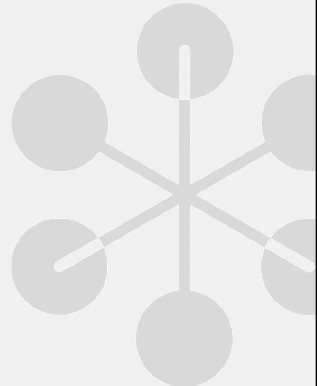
## Version Control Basics

### •What is Git?

- A distributed version control system for tracking changes in files and allow multiple developers to collaborate on projects.

### •Why use Git?

- Keeps a detailed history of changes.
- Enables multiple developers to work on the same project without conflicts.
- Provides tools for branching, merging, and resolving conflicts.



**QA**

## Key Git Concepts

Basic Git Commands

Branching and Merging

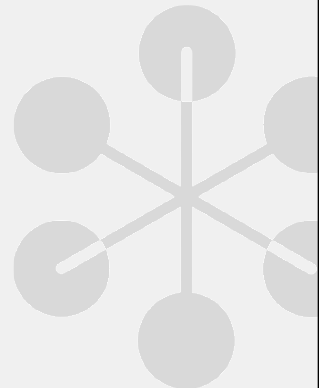
Introduction to GitHub

Repository Creation and Setup

Linking VSCode

Pull Requests

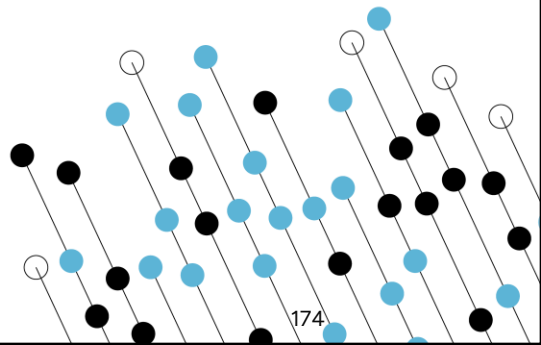
Managing Merge Conflicts



**QA**

## Q&A

Review and Q&A



## Any Final Questions?

Last opportunity to ask any questions you may still have:

Still confused?



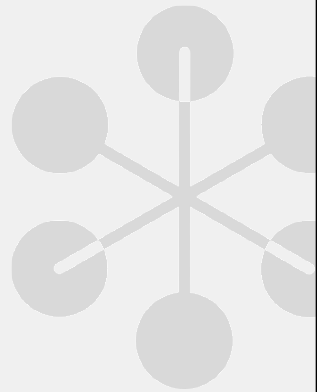
Last pieces of the puzzle?



If not you then help someone else by asking that question you've been itching to ask



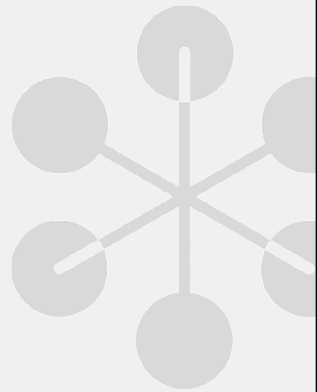
**QA**



## End of Course Evaluations

Don't forget to complete the end of course evaluation!

You will have been sent a link to the questionnaire in an email today



# Thank you

QA

