



# TOP - Preparation

Shymmy W. Garcia

# Contents

---

## The Terminal

Introduction and basic commands

01

## Git - GitHub

Introduction and basic commands

02

## Node Js

What is node js

03

## Javascript

Introduction to Javascript

04



# The Terminal

# The terminal



## What is Terminal?

Terminal is an application that gives us a command line interface (or CLI) to interact with the computer.

Everything you can do in Finder you can do in Terminal.

Developers use Terminal because, more often than not, it is much faster to use Terminal than a graphical user interface (GUI) such as Finder.

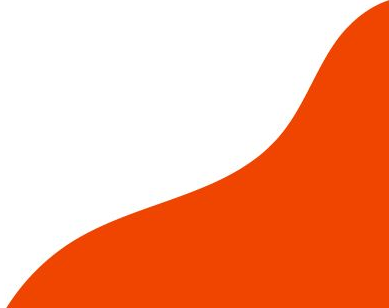


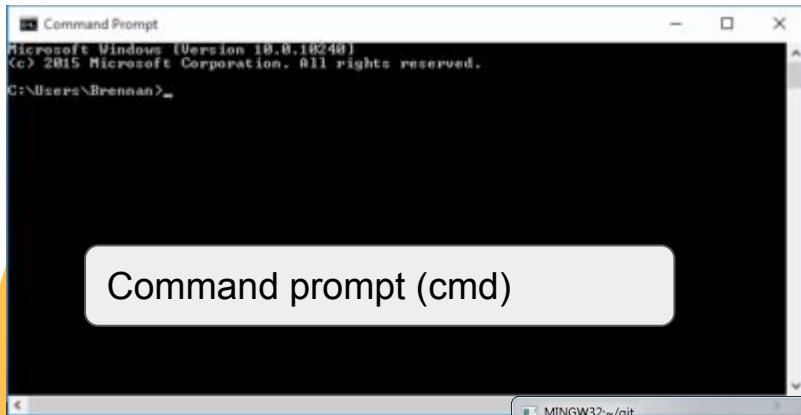
# The terminal

## What is a shell? Bash/ZSH

The **shell** is the program which actually processes commands and returns output. Most shells also manage foreground and background processes, command history and command line editing. These features (and many more) are standard in bash, the most common shell in modern linux systems. (We are using zsh).

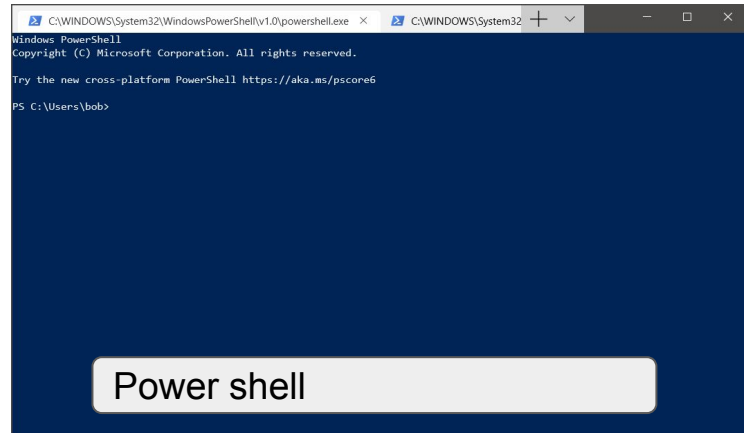
A **terminal** refers to a wrapper program which runs a shell. Decades ago, this was a physical device consisting of little more than a monitor and keyboard. As unix/linux systems added better multiprocessing and windowing systems, this terminal concept was abstracted into software.

A large, solid orange shape in the bottom right corner of the slide, resembling a stylized wave or a corner graphic.



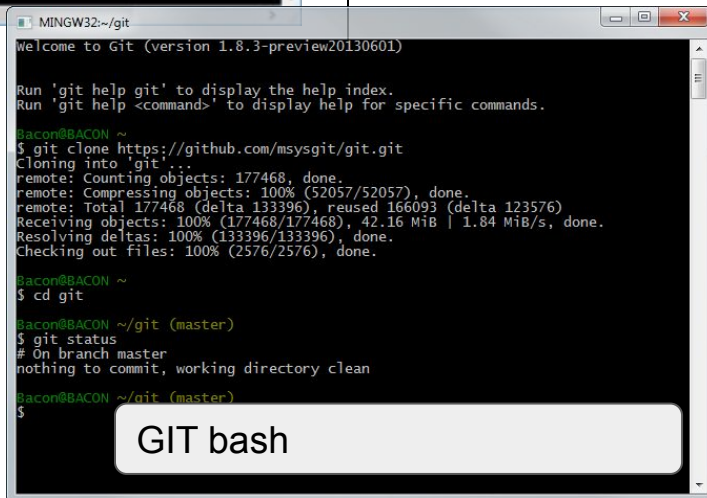
A screenshot of the Windows Command Prompt (cmd) window. The title bar reads "Command Prompt". The window content shows the Microsoft Windows version (10.0.18240) and copyright information (© 2015 Microsoft Corporation). The prompt is at "C:\Users\Brennan>".

Command prompt (cmd)



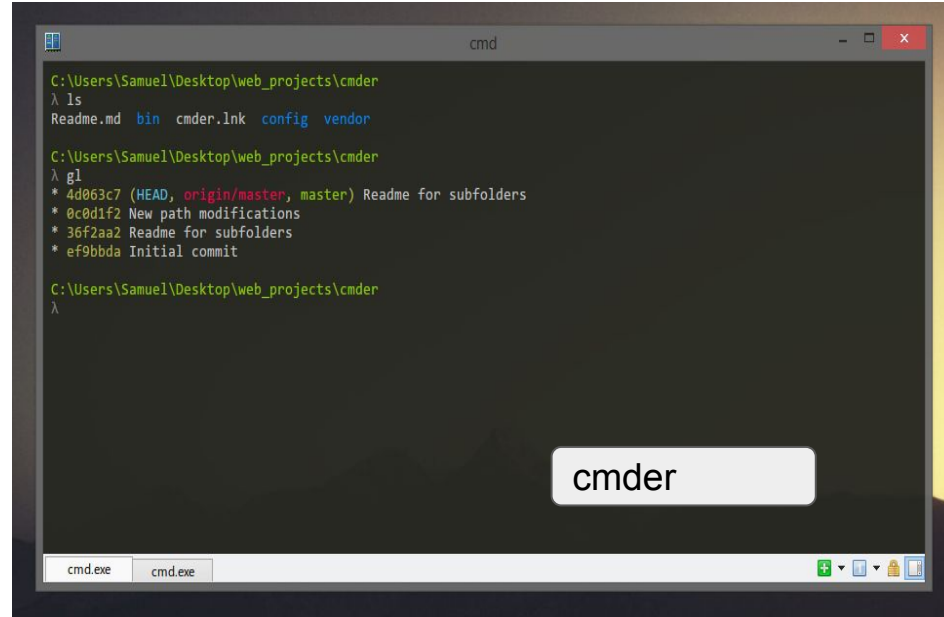
A screenshot of the Windows PowerShell window. The title bar shows the path "C:\WINDOWS\System32\WindowsPowerShell\v1.0\powershell.exe". The window content displays the PowerShell version (5.0.10586.16) and copyright information (© Microsoft Corporation). The prompt is at "PS C:\Users\Bob>".

Power shell



A screenshot of the Git Bash terminal window. The title bar shows the path "MINGW32~/git". The window content displays the Git version (1.8.3-preview20130601) and copyright information (© 2013-2015 Git project). The prompt is at "Bacon@BACON ~". The user has cloned a repository from GitHub and checked out the master branch. The prompt is now at "Bacon@BACON ~/git (master)".

GIT bash



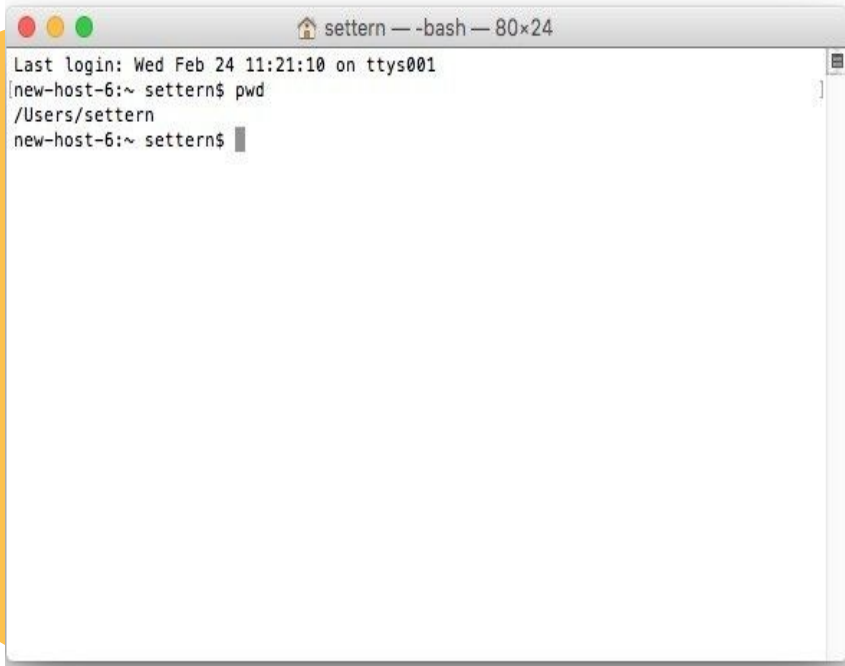
```
C:\Users\Samuel\Desktop\web_projects\cmdr
λ ls
Readme.md  bin  cmdr.lnk  config  vendor

C:\Users\Samuel\Desktop\web_projects\cmdr
λ git log
* 4d063c7 (HEAD, origin/master, master) Readme for subfolders
* 0c0d1f2 New path modifications
* 36f2aa2 Readme for subfolders
* ef9bbda Initial commit

C:\Users\Samuel\Desktop\web_projects\cmdr
λ
```

cmdr

cmd.exe cmd.exe



A screenshot of a macOS Terminal window. The title bar shows a home icon, the text 'settern — -bash — 80x24', and standard window control buttons. The terminal content shows a login session: 'Last login: Wed Feb 24 11:21:10 on ttys001', followed by a prompt 'new-host-6:~ settern\$' where the user enters 'pwd', resulting in the output '/Users/settern'. The prompt returns to 'new-host-6:~ settern\$' with a cursor.

```
settern — -bash — 80x24
Last login: Wed Feb 24 11:21:10 on ttys001
new-host-6:~ settern$ pwd
/Users/settern
new-host-6:~ settern$
```

Bash

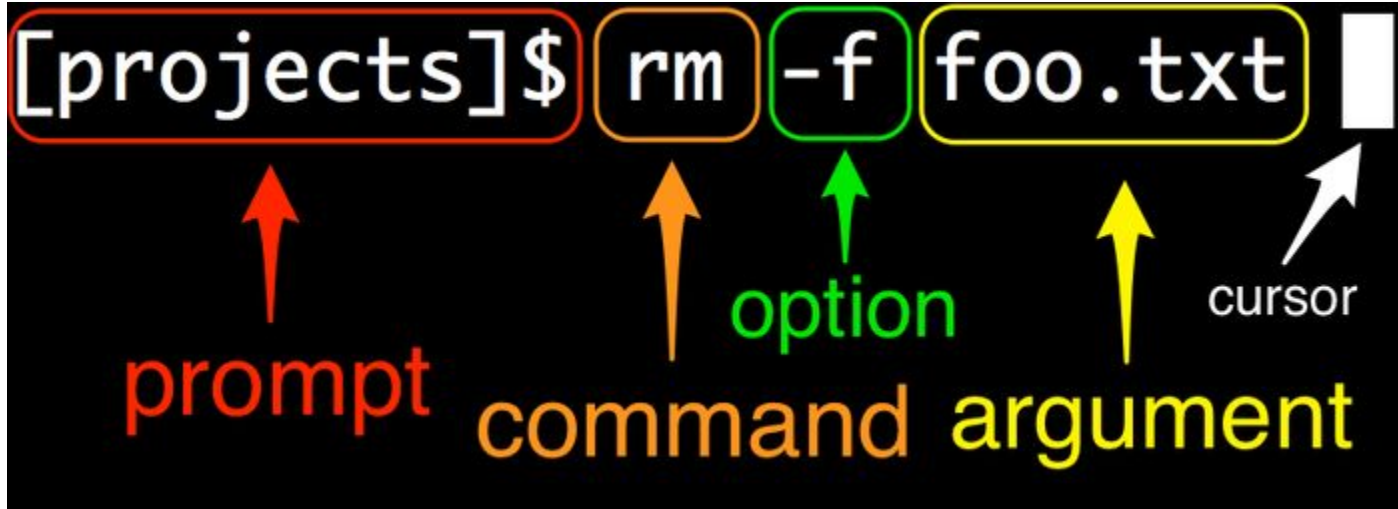
Zsh → Z-shell



# The terminal

The typical elements of a command, are:

- The prompt (to “prompt” the user to do something)
- a command (as in “give the computer a command”),
- an option (as in “choose a different option”),3 and
- an argument (as in the “argument of a function” in mathematics).



# Mac: Terminal (z-Shell)

## Basic Commands

### ▼ Basic commands

echo

pwd  
(print working directory)

ls  
(list items)

### ▼ Navigation

cd (..)  
(change directory)

cd /  
(root directory)

cd /Users  
(users directory)

cd or cd ~  
(home directory)

### ▼ Files / folders

touch  
(create/ "touch" file)

mkdir  
(create directory)

rm  
(delete file)

rmdir  
(delete empty folder)

cp  
(copy file/folder)



**Let's practice commands**

**MAKE IT REAL**  
CAMP

**GIT**

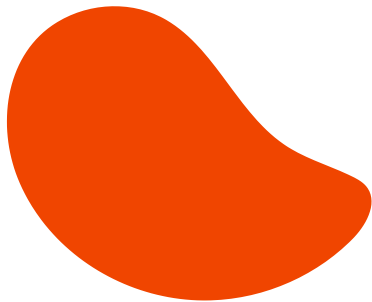


**git**

## What is GIT?



Git is an Open Source Distributed (Decentralized) Version Control System (VCS) designed to make it easier to have multiple versions of a code base, sometimes across multiple developers or teams.



# Version Control System (VCS)

## What is Version Control?

Version control, also known as source control, is the practice of tracking and managing changes to software code.

## What is a Version Control System?

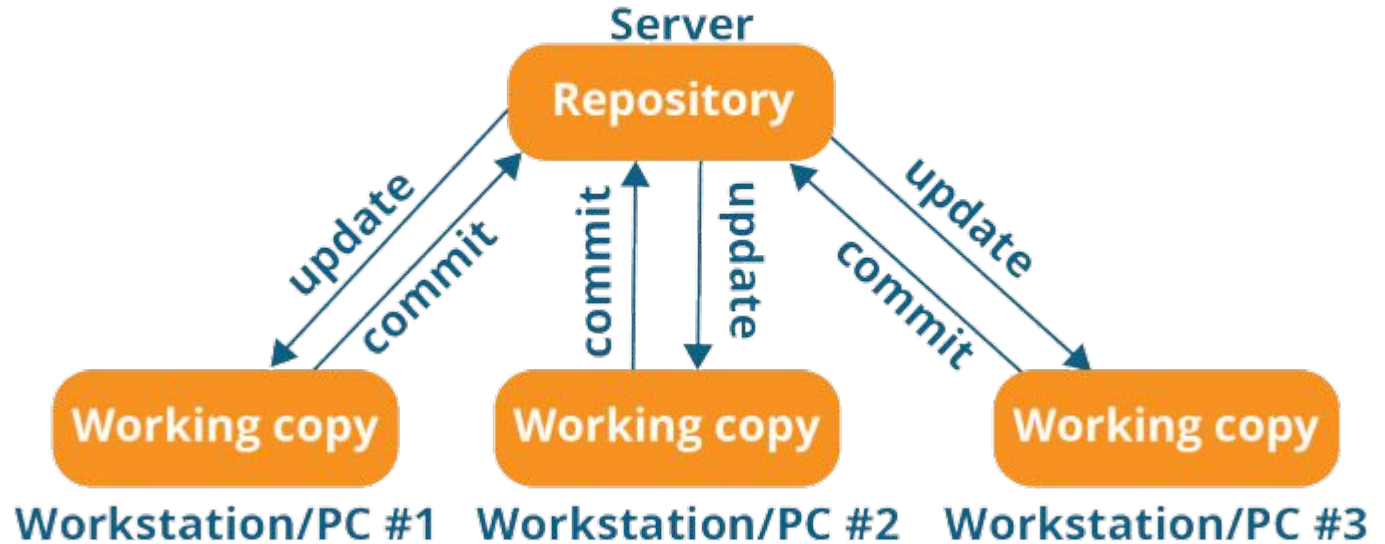
Version control systems are software tools that help software teams manage changes to source code over time.

Version control software **keeps track of every modification to the code** in a special kind of database. If a mistake is made, developers can turn back the clock and compare earlier versions of the code to help fix the mistake while minimizing disruption to all team members.



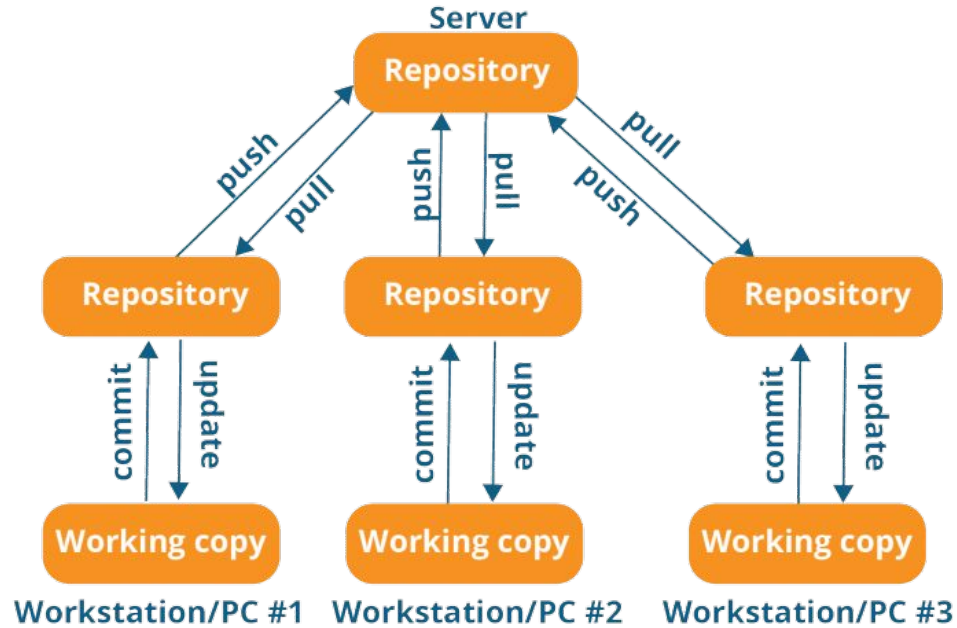
## What is Distributed?

Centralized version control system



# What is Distributed?

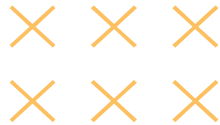
Distributed version control system





## Be Aware!

**GIT IS NOT EQUAL TO GITHUB!**





# GIT: Setup and basics

- **Download and Install GIT:**

- **Verify GIT version:**

`git --version.`

- **Configure GIT:**

`git config --global user.name <name> :`  
Define the author name to be used for all commits by the current user.

`git config --global user.email <email>:`  
Define the author email to be used for all commits by the current user.it.

## Installation and setup



# GIT

## ▼ Basic commands

git init

git add

git status

git commit

git diff

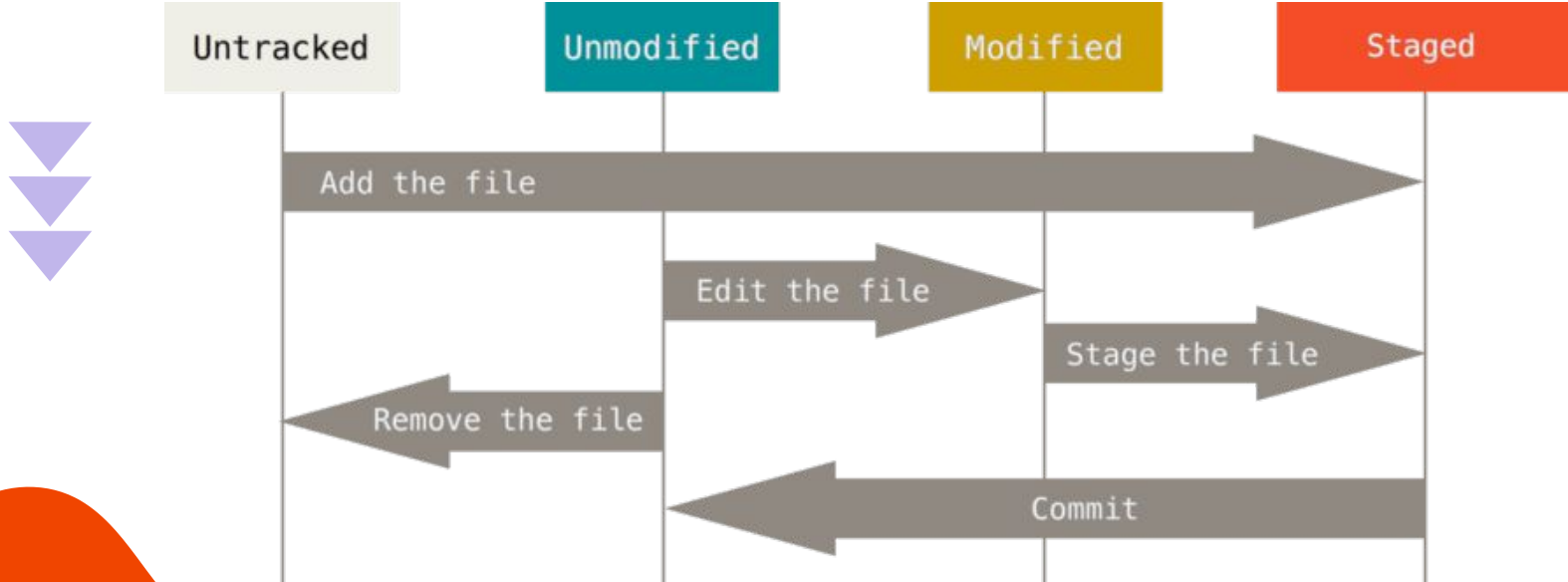
git rm --cached

git log



# How Does Git Works?

# The lifecycle of the status of your files



# GIT File Status



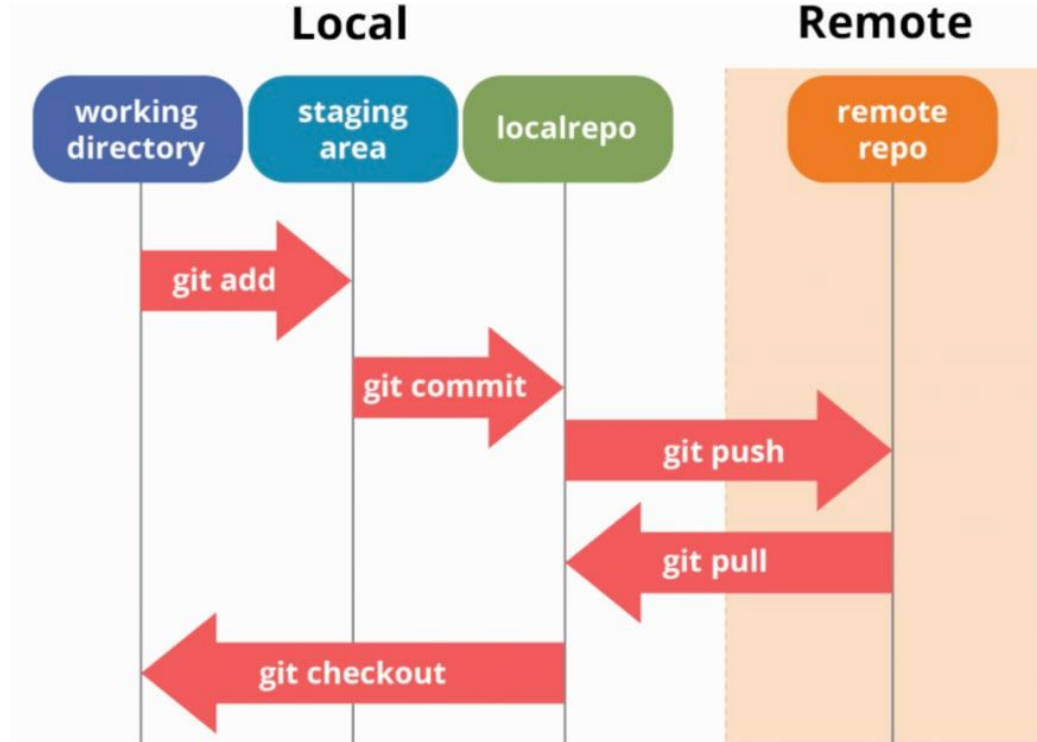
Each file in your working directory can be in one of the following two states: **Tracked or Untracked**

File status in Life Cycle :

- Untracked
- Unmodified
- Modified
- Staged



# Basic GIT WorkFlow





# Basic GIT WorkFlow

**Git directory** - Stores the metadata and object database for your project (while cloning the repository) .

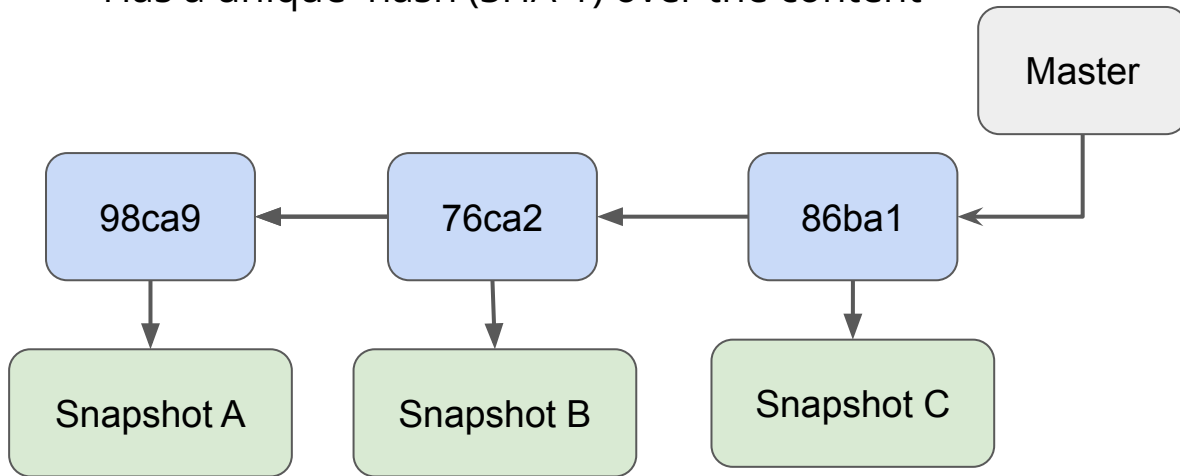
**Working directory** - Single checkout of one version of the project. The files in the directory are pulled out of the compressed database in the Git directory and placed on disk for you to edit and use.

**Staging area** - It is a simple file, generally present in your Git directory, that stores information about what will go into your next commit.



# What is a commit, anyway?

- In Git, a commit is a snapshot of your repo at a specific point in time.
- Keeps references to its parents
- Has a unique hash (SHA-1) over the content





**Let's practice GIT  
commands**

## Let's do it

- Create a new repo
- Create a file
- Check the Status of the file
- Tracking New Files
- Staging Modified Files
- Short Status -s
- Commit
- Commit no Staging area -a -m
- Untrack Files





**Let's practice GIT  
commands**



# GIT: Branches - Branching

## Recap

- Git -> software - tool

## Recap

- Git -> software-tool
- Repository -> folder Project/.git

```
$ git init
```

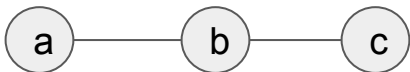


## Recap

- Git -> software-tool
- Repository -> folder `Project/.git`
- Commit -> `Project/.git/objects`  
object (changes, author, ID ...)

# Recap

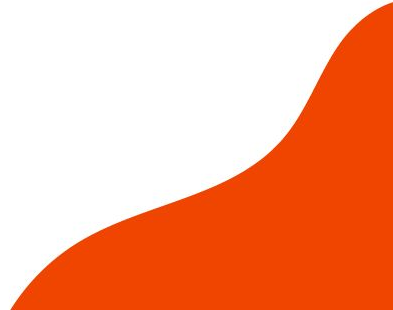
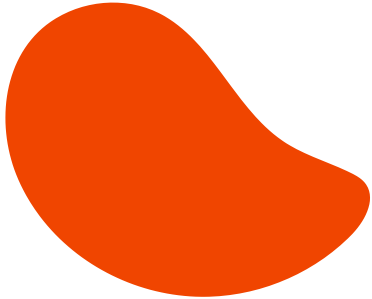
**Project/.git**

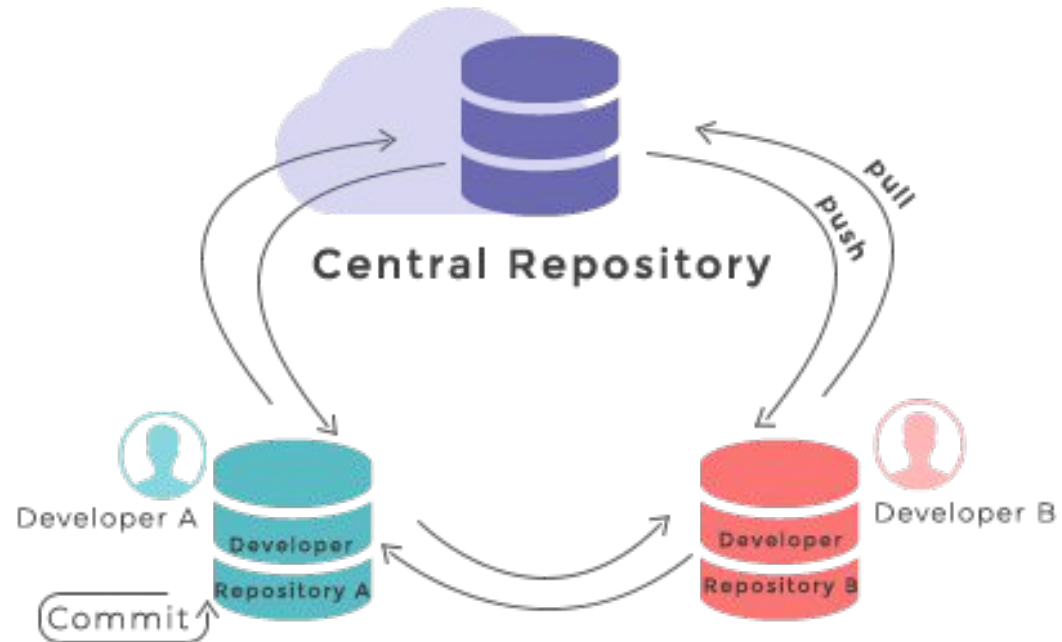


# What if ...

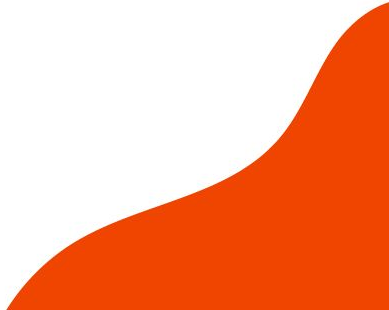
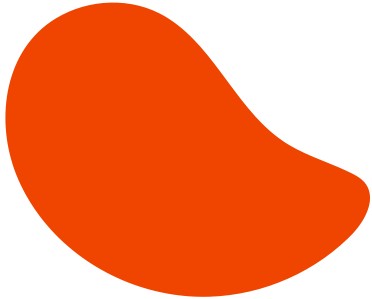
**Experiment**

**Collaborate**



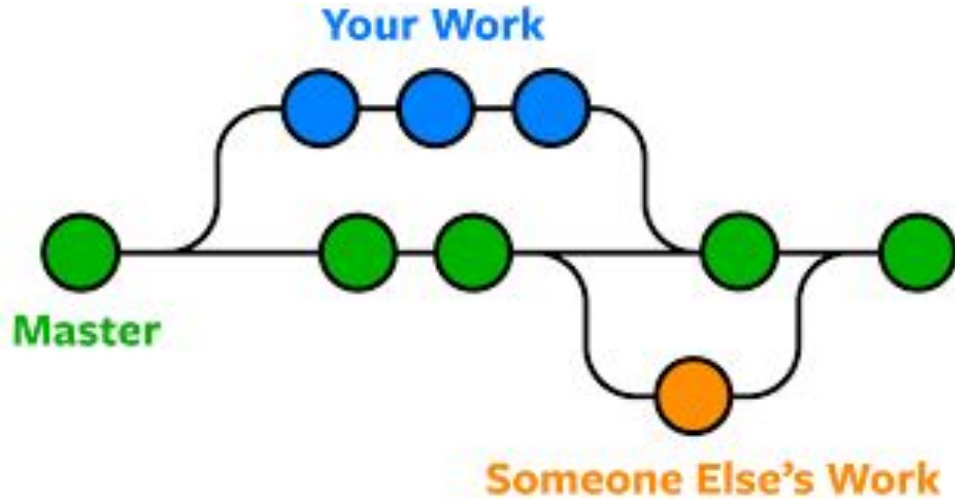


**How could you experiment/collaborate  
with your code safely**



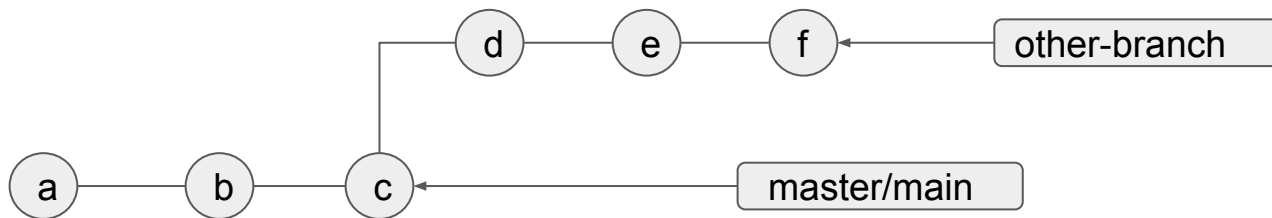
# Branching

Branching means you diverge from the main line of development and continue to do work without messing with that main line.



# What is a branch

Technically it is a reference to a commit.



# Git branch Commands

```
$ git branch <your branch name> // creates
```

```
$ git checkout <your branch name> // switches
```

```
$ git checkout -b <your branch name> // crates and switches
```

```
$ git branch // lists
```

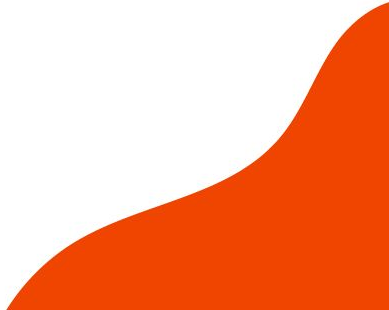
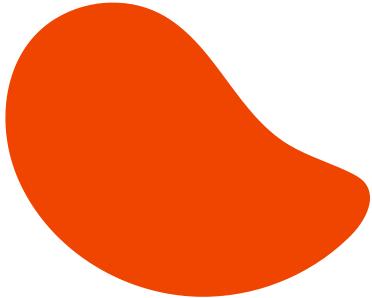
```
$ git branch -d <your branch name> // deletes
```

```
$ git branch -m <your branch name> // renames
```



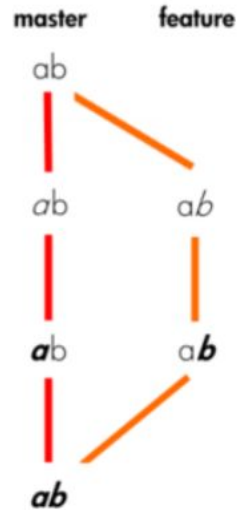


How could we **integrate** changes from one branch to another

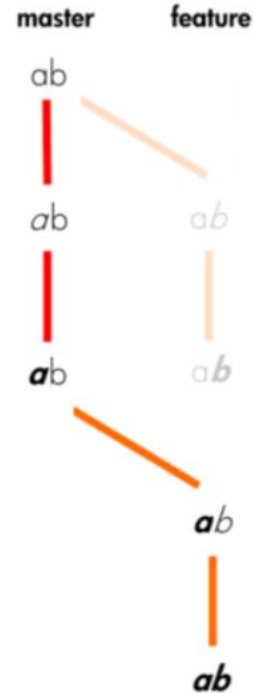


# Merging vs Rebasing

## Merge



## Rebase



## git merge:

```
git checkout master  
git merge my-branch
```

### Fastforward

When there are no new  
commits in master

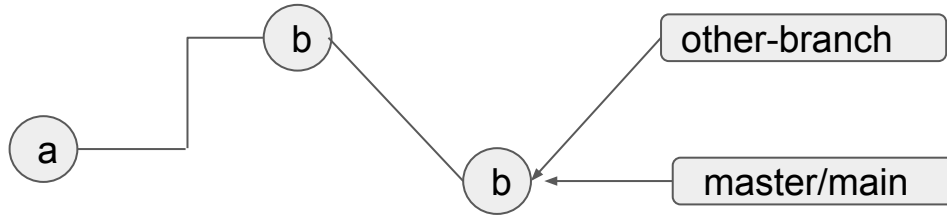
### Merge commit

When there are also new commits in  
master as well as in my-branch

### Solving conflicts

When code was erase or modify in the  
same lines

## git merge:

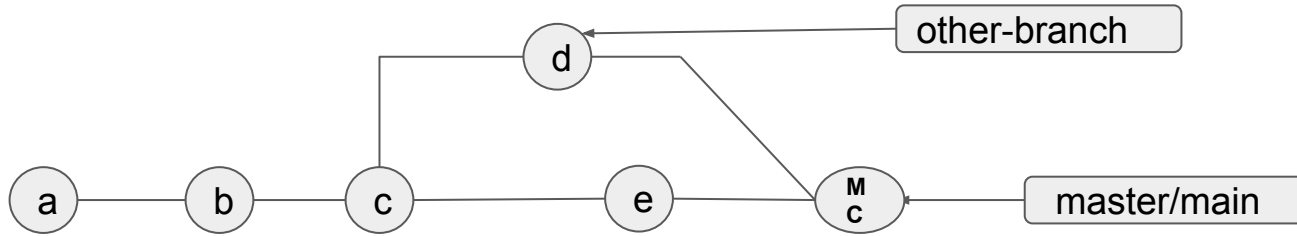


Create a new branch and one new commit from it.  
Merge new branch into master

## Fastforward

When there are no new  
commits in master

# git merge:

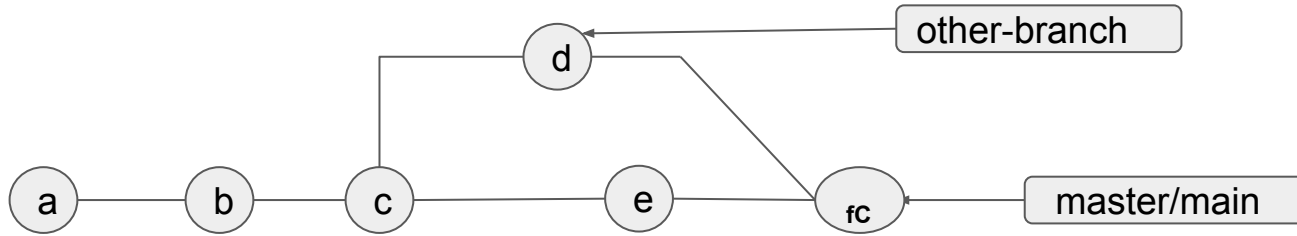


Create a new file in branch 2  
Update file in master  
Merge branch-2 to master

## Merge commit

When there are also new commits in master as well as in ny-branch

# git merge:



Update a file in branch 2 and commit  
 Update same file in the same lines in master and commit  
 Merge branch-2 into master  
 Solve conflicts  
 git add .  
 git commit

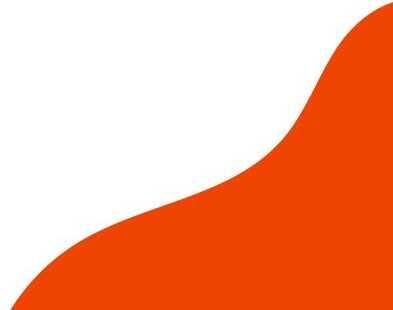
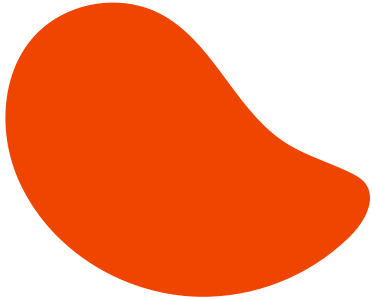
## Solving Conflicts

When there are also new commits in master as well as in my-branch and they modified the same lines in a file or created files with the same name

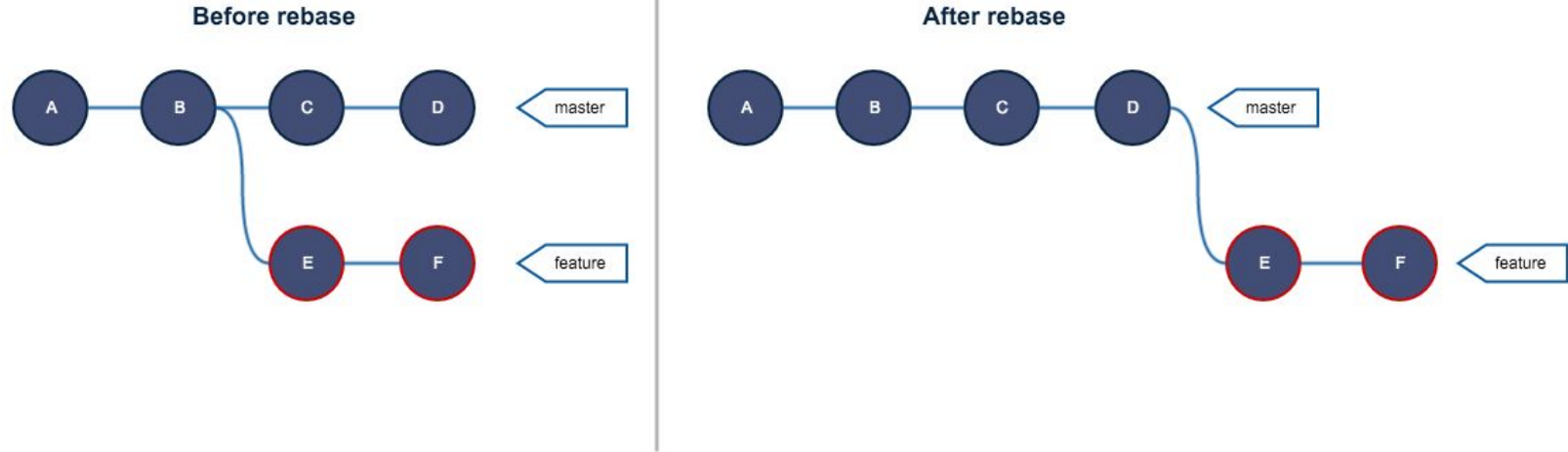
# Git rebase



Imagine you need to bring some new commits from master to your branch but you do not one to create a merge commit



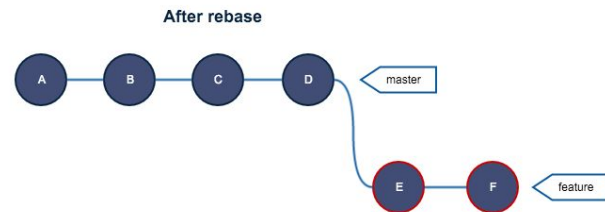
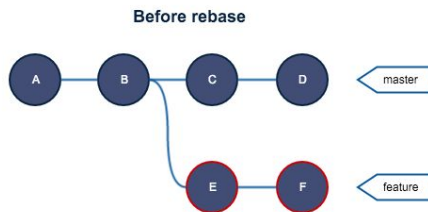
# git rebase: workflow





# git rebase: workflow

```
git checkout my-branch
git rebase master
```



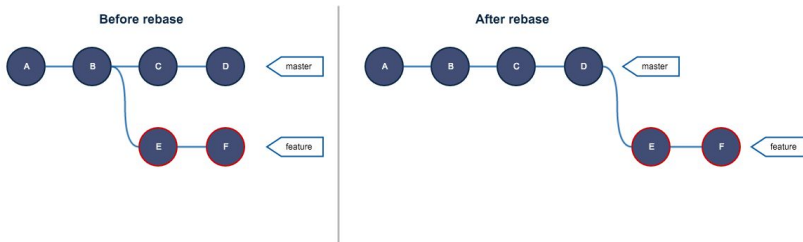
rewinding head to replay your  
work on top of it

# git rebase: workflow

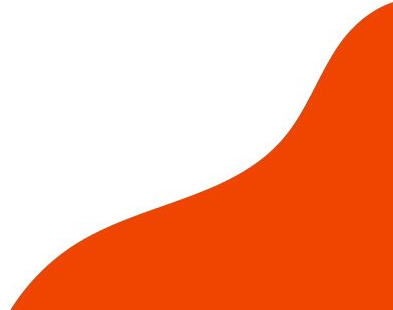
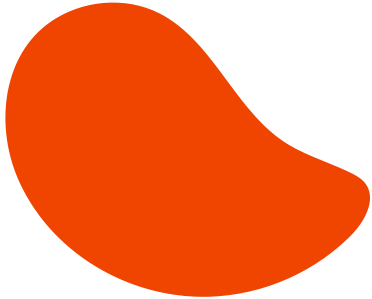
Example of rebasing master into my-branch

1. Move the commits you have made to my-branch to a temporary place.
2. Update my-branch with the new commits that have been created in master.
3. Apply the new commits you have made to my-branch, which had now been updated with respect to master.

The result is that my-branch will now have the same commits as master, in addition to the new commits you had in my-branch.



## Local vs Remote repos





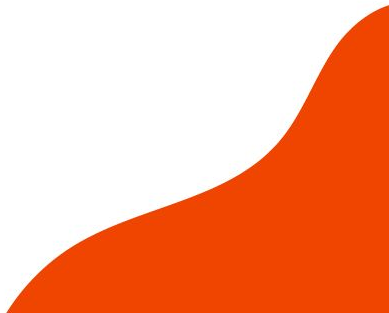
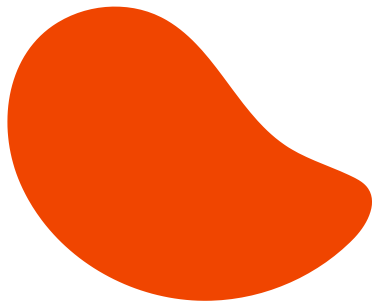
**Let's practice GIT  
branching**

# Merging vs Rebasing



From a conceptual standpoint, git merge and git rebase are used to achieve the same ultimate goal: to integrate changes from one branch into another branch.

There are, however, distinct mechanics to both methods.



# Git merge



- Git merge will combine multiple sequences of commits into one unified history.
- In the most frequent use cases, git merge is used to combine two branches.
- git merge takes two commit pointers, usually the branch tips, and will find a common base commit between them.
- Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence.



## git merge: pros

- Generally the easiest option to merge your master branch into your current working feature branch.
- You can ``git checkout feature`` and then ``git merge master`` or you could just do it with one command: `git merge feature master`
- This create a new 'merge commit' in your feature branch, which is a non-destructive operation that ties the histories of both branches.
- This preserves the exact history of your project



## git merge: cons

- The branch that you merge will always have an extraneous merge commit that will be tracked every time you need to incorporate upstream states.
- In other words, it essentially creates a forked history at the point where you merge.
- This can lead to muddling the history of your branch, thereby making it more difficult for yourself or other developers to track the history of changes using ``git log`` and/or roll back to previous states





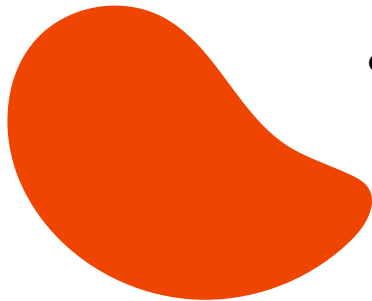
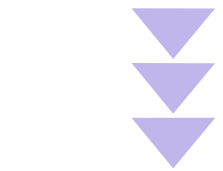
## git rebase: pros

- To rebase, you would `git checkout feature` and then `git rebase master`.
- Instead of creating a merge commit, rebase will move the entire feature branch to start from the tip of the master branch by rewriting the project history and creating brand new commits for each commit in the original branch.
- The result is a singular history with no forking of the commit history.



## git rebase: cons

- Because rebase rewrites project history, you lose the context provided by a merge commit, i.e. you won't be able to see when upstream changes were actually integrated into the feature branch.
- More importantly, you could potentially cause extreme difficulty by rebasing master to the tip of your feature branch, leading git to think that your master branch's history has diverged from the rest
- In doing so, everyone else would still be working from the original master branch, and both masters would need to be merged together.





# GIT: Remote Repos

# Remote repositories

- A remote repository in Git, also called a remote, is a Git repository that's hosted on the Internet or another network.

## GitHub

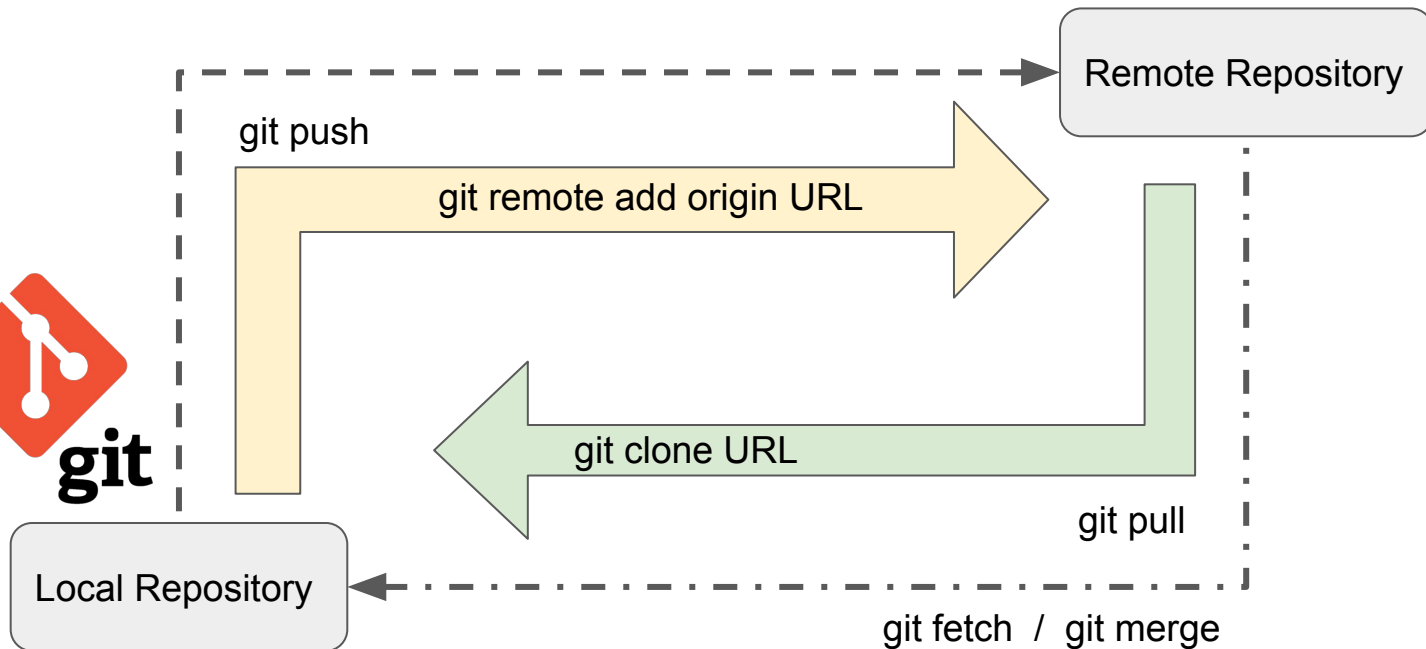
- GitHub is a **Git repository hosting service**
- GitHub provides a Web-based graphical interface.
- It also provides access control and several collaboration features, such as a wikis and basic task management tools for every project.





**Let's go to github**

# Git & GitHub



# **GIT**

## **Remote Commands**

### **Commands**

`git remote add origin URL`

`git fetch -> git merge remote/Branch`

`git pull origin branch`

`git push origin branch`

`git clone URL`

### **Remote Branches**

`git branch -a`

`git branch -r`

`git remote show origin`

`git branch -vv`

**MAKE** **IT** **REAL**  
CAMP

**NODEJS**





# Node Js

- Node.js is a JavaScript runtime built on Chrome's V8 JavaScript engine.
- Node allows developers to write JavaScript code that runs directly in a computer process itself instead of in a browser.
- Node can, therefore, be used to write server-side applications with access to the operating system, file system, and everything else required to build fully-functional applications.



**Let's go to NODE JS**

- **Sintax:**

Variables

Data Types

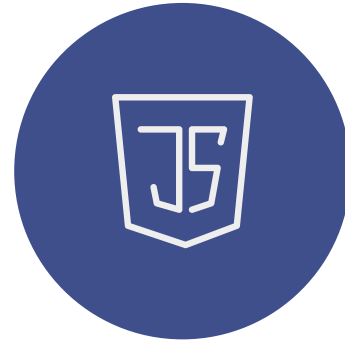
- **Flow control**

Conditional

Iterative

- **Functions**

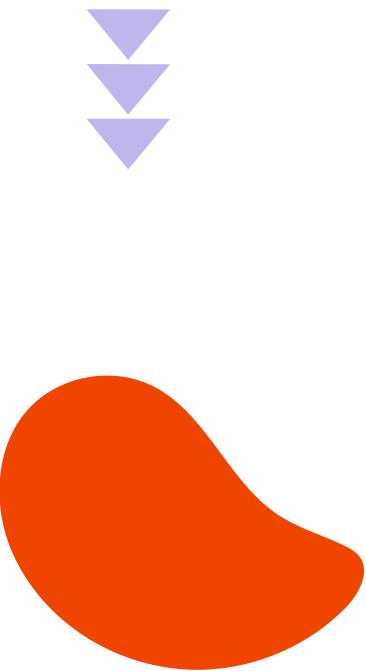
**NodeJS**






# JS: Fundamentals

# Variables



keyword	const	let	var
global scope	NO	NO	YES
function scope	YES	YES	YES
block scope	YES	YES	NO
can be reassigned	NO	YES	YES



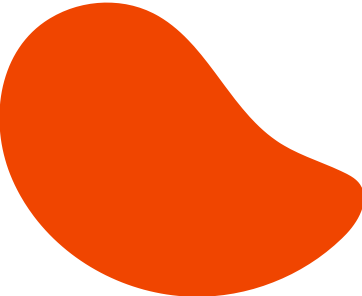

# Data Types

Primitive values (immutable datum represented directly at the lowest level of the language)

- Boolean type
- Null type
- Undefined type
- Number type
- String type
  
- BigInt type\*
- Symbol type\*

# Conditionals

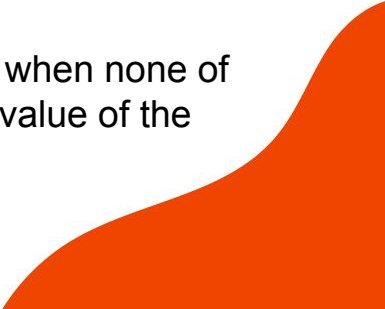

## IF/ELSE



```
if (condition) {  
    statements1  
} else {  
    statements2  
}
```

## switch-case

```
switch (expression) {  
    case value1:  
        //Statements executed when the  
        //result of expression matches value1  
        [break;]  
    case value2:  
        //Statements executed when the  
        //result of expression matches value2  
        [break;]  
    ...  
    case valueN:  
        //Statements executed when the  
        //result of expression matches valueN  
        [break;]  
    [default:  
        //Statements executed when none of  
        //the values match the value of the  
        expression  
        [break;]]  
}
```



# Iterations

## for

for ([initialization]; [condition]; [final-expression]){  
    Statement  
}

## while

```
while (condition){  
    statement  
}
```

## do -while

```
do{  
    statement}  
while (condition);
```





# JS: Functions

# Functions

## Function Declarations

```
function name([param[, param[,..., param]]]) {  
  [statements]  
}
```

## Function Expressions

```
const variableName = function(param, param) {  
  statements  
};
```

## Arrow Functions

```
(param1, paramN) => expression
```



# JS: Node Modules

## Node Js - package.json

As a general rule, any project that's using Node.js will need to have a package.json file.



At its simplest, a package.json file can be described as a manifest of your project that includes the packages and applications it depends on, information about its unique source control, and specific metadata like the project's name, description, and author.

A package.json file is always structured in the JSON format, which allows it to be easily read as metadata and parsed by machines.



# NodeJs - package.json



```
{  
  "name": "myproject", // The name of your project  
  "version": "0.92.12", // The version of your project  
  "description": "my project description.", // The description of your project  
  "main": "index.js"  
  "license": "MIT" // The license of your project  
}
```



# **NodeJs - package.json - dependencies** and devDependencies



Dependencies are the modules that the project relies on to function properly.

it allows the separation of dependencies that are needed for production and dependencies that are needed for development



# Node Js - package.json



```
{  
  "name": "myproject", // The name of your project  
  "version": "0.92.12", // The version of your project  
  "description": "my project description.", // The description of your project  
  "main": "index.js"  
  "license": "MIT" // The license of your project,  
  "devDependencies": {  
    "mocha": "~3.1",  
    "native-hello-world": "^1.0.0",  
  },  
  "dependencies": {  
    "fill-keys": "^1.0.2",  
    "resolve": "~1.1.7"  
  }  
}
```

# Node Js -Essential npm Commands

- **npm init** -> to Initialize a Project
- **npm init --yes** -> to Instantly Initialize a Project

## Install Modules with npm install

- **npm install** <module> # Where <module> is the name of the module you want to install
- **npm i** <module> # Where <module> is the name of the module you want to install - using the i alias for installation
- **npm install -D** <module> # Where <module> is the name of the module you want to install, using -D flag for instal dev dependencies

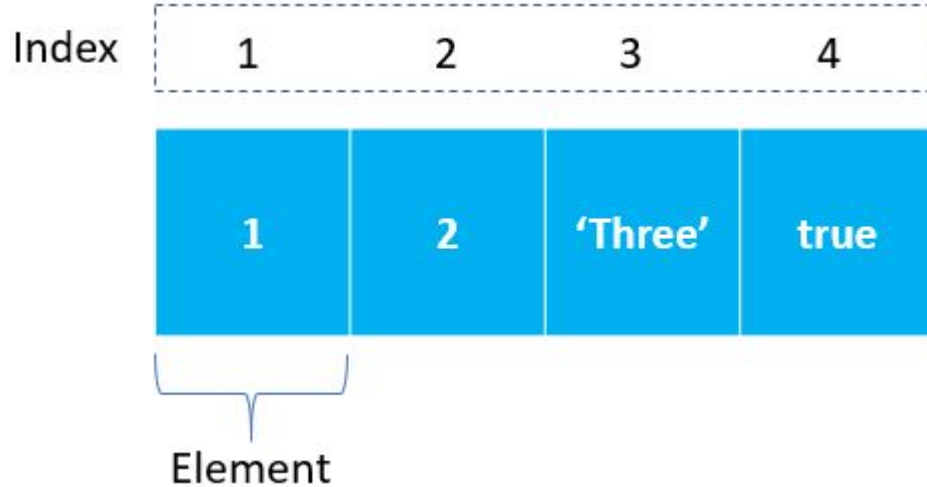




# JS: Array

# Arrays

an array is an ordered list of values. Each value is called an element specified by an index.



## Arrays Creation

- Using []

```
const cars = ['Saab', 'Volvo', 'BMW'];
```

- new Array()

```
const cars = new Array('Saab', 'Volvo', 'BMW');
```

## Changing Elements

- array[position]=value

## Array Properties

- length property provides an easy way to append a new element to an array

# Arrays Methods


- **pop()** method removes the last element from an array
- **push()** method adds a new element to an array (at the end)
- **shift()** method removes the first array element and "shifts" all other elements to a lower index.
- **unshift()** method adds a new element to an array (at the beginning), and "unshifts" older elements
- **The slice( )** method copies a given part of an array and returns that copied part as a new array. It doesn't change the original array.
- **The splice( )** method changes an array, by adding or removing elements from it.

# Iterate Arrays

- **for - of**
  - for (variable of iterable) {  
statement }
- **forEach**  
forEach(callbackFn)



# Arrays - map - filter -reduce

- 
- **.map():** Creates a new array with the results of calling a function for every element in array.
  - **.filter():** Creates a new array with all elements that pass the test from the provided function.
  - **.reducer():** Combines each element of an array, using a reducer function you specify, and returns a single value.



```
array.reduce((accumulator, currentValue, index, array) => {...}, initialValue)
```





# JS: Objects

# Objects

- **Object:** is used to store various keyed collections and more complex entities. Objects can be created using the `Object()` constructor or the object initializer / literal syntax..
- an object can be created with figure brackets `{...}` with an optional list of properties. A property is a “key: value” pair, where key is a string (also called a “property name”), and value can be any data type.
- **Create an object**
  - `const user = {};` // "object literal" syntax
  - `const user = new Object();` // "object constructor" syntax



# Objects



```
const person = {  
  firstName: 'John',  
  lastName: 'Doe',  
  age: 22,  
  eyeColor: 'Hazel',  
};
```



## Accessing Object Properties




- Object properties can be accessed using the dot notation:
  - `objectName.propertyName`
- Or using square brackets:
  - `objectName["propertyName"]`

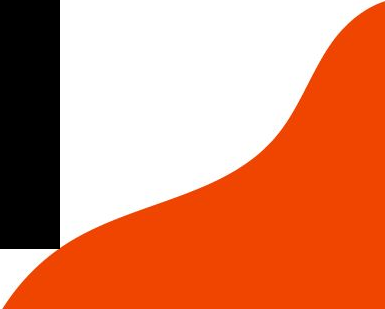
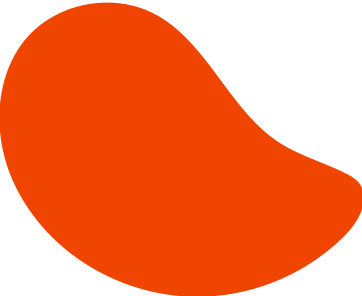


## Object Methods

- Object methods are actions that can be performed on objects.
- Methods are stored in properties as function definitions.



```
const person = {  
  firstName: 'John',  
  lastName: 'Doe',  
  age: 22,  
  eyeColor: 'Hazel',  
  greeting: function () {  
    return `Hi, I am ${this.firstName}  
    ${this.lastName}.`;   
  },  
};
```



## Constructors and object instances


- JavaScript uses special functions called constructor functions to define and initialize objects and their features.
- They are useful because you'll often come across situations in which you don't know how many objects you will be creating.
- Constructors provide the means to create as many objects as you need in an effective way, attaching data and functions to them as required.

## Constructors and object instances



```
function Person(name) {  
  this.name = name;  
  this.greeting = function() {  
    console.log('Hi! I\'m ' + this.name +  
      '.');  
  };  
}  
  
let person1 = new Person('Bob');  
let person2 = new Person('Sarah');
```

## The Object() constructor



```
let person1 = new Object({  
  name: 'Chris',  
  age: 38,  
  greeting: function() {  
    alert('Hi! I\'m ' + this.name + '.');  
  }  
});
```

