# Fundamentals 1

## IntelliJ

To demonstrate the features in IntelliJ I'll be using a sample project of mine.

https://github.com/astb01/claimants-service-api

### Features in IntelliJ

#### Keymap

This is how IntelliJ maps the keys in relation to which device you're on:

> **Preferences > Keymap**: Mac OSX. Windows: Default

#### Navigating Around the Idea with shortcuts

- To find files anywhere: **Double shift**. You also use capitals or wildcards.
- Jump to line in file: **filename:line number**
- Going back and forth: **CMD + [** for forward and **CMD + ]** for going backwards.
- Creating new file; **CMD + N** on the folder.
- Let's open multiple files. Knowing which files we had open is sometimes a pain or what we last worked on when we got back to our desks: **CMD + E** (recent)
- Recently edited files: **Shift + CMD + E**
- Creating packages/folders. Type the whole path. **not one at a time**.

#### Plugins

- Presentation Assist

> **Preferences > Plugins > Presentation Assist**. What this does is show the key you have pressed so next time you're coding with someone and they ask "What did you do? I wasn't watching. Just say look at my screen!"

#### Themes

Now let's powder our nose .... not in that sense ...

Plugin: Material UI Theme Editor > Color Scheme

Demonstrate going around the IDE menu by menu.

> Use IntelliJ Guide PDF.

## Debugging

Use debugging-example.java

- gist

Use debugging-example-2.java:

- Message.java from gist
- SayHello.java from gist

**Demonstrate**:

- Stepping over
- Stepping into
- Stack traces
- Variables

**Demonstrate**:

- Conditional breakpoints.
- Right click on breakpoint and add expression (boolean).

Use **debugging-example.java** on line 34 by looking for a name match.

Expression: `students.get(i).equals("John")` (on line 34)

We'll look at expressions later throughout the week.

**Demonstrate**:

- Adding watch expressions.
- Click on Watch icon under variables.

## Tips

Tips:

- Navigate to files: Double shift/CMD+Shift+O - Can use wildcards.
- To see the whole screen: CMD+Shift+f12 or better, *View* > *Enter Distraction Free Mode*.
- **CMD + Shift + V** - to see clipboard history
- Reformat code. The style used can be defined under **Preferences > Code Style > Java**

## Compiler Warnings

So what happens when things aren't right. How can we find issues?

```java
public class ImportExample {
  public static void main(String[] args) {
    Random r = new Random(); // DOES NOT COMPILE
    System.out.println(r.nextInt(10));
  }
}
```

The Java compiler helpfully gives you an error that looks like this:

*Random cannot be resolved to a type*

Now at first glance you might think you've made a typo, or done something wrong.

You will have noticed a 'red' light buld symbol over the code. This is IntelliJ helping you find a cure to your issue.

The issue here is you have not imported another file that you have said you want to use, Random.

You how does this work? EXPLAIN USING ANALOGY:

> Imagine you are mailing a letter to 123 Main St., Apartment 9. The mail carrier first brings the letter to 123 Main St. Then she looks for the mailbox for apartment number 9. The address is like the package name in Java. The apartment number is like the class name in Java. Just as the mail carrier only looks at apartment numbers in the building, Java only looks for class names in the package.

---

## Version Control

Back in my days when I was college, infact even at university, they had the concept of floppy discs where we saved all our work each time we wanted to store work.

Thankfully the world has moved on and now we have USB dongles, cloud storage etc.

For organisations to use the old concepts would have been laughable but again, times have changed.

All organisations now use some form of storage or have a retention policy to protect the data they use.

In software houses, we use a concept referred to as VCS, a version control system. It is also referred to as **source control**.

There are many source control systems out there that provide some form of storage/locking mechanism to ensure we do not lose our work.

There are many proprietary VCS systems out there:

| VCS |
|---|

ClearCase Perforce Vault

There are others however these are the main ones.

There are also open source/free VCS systems:

| VCS |
|---|

CVS Subversion Mercurial

VCS systems are however split into two domains. Client-Server and Distributed.

The ***client-server*** approach is where you have a server for VCS system housed within your organisation, on some mega PC/hardware and all the developers log into and connect to that server. This is good but it does open up the business' risk as they could lose that server if say they had a natural disaster/fire.

As a developer, you would log into a server and would only work whilst connected to the server. In some cases this would mean you're only able to work, if at the office, not at home.

DRAW ON BOARD

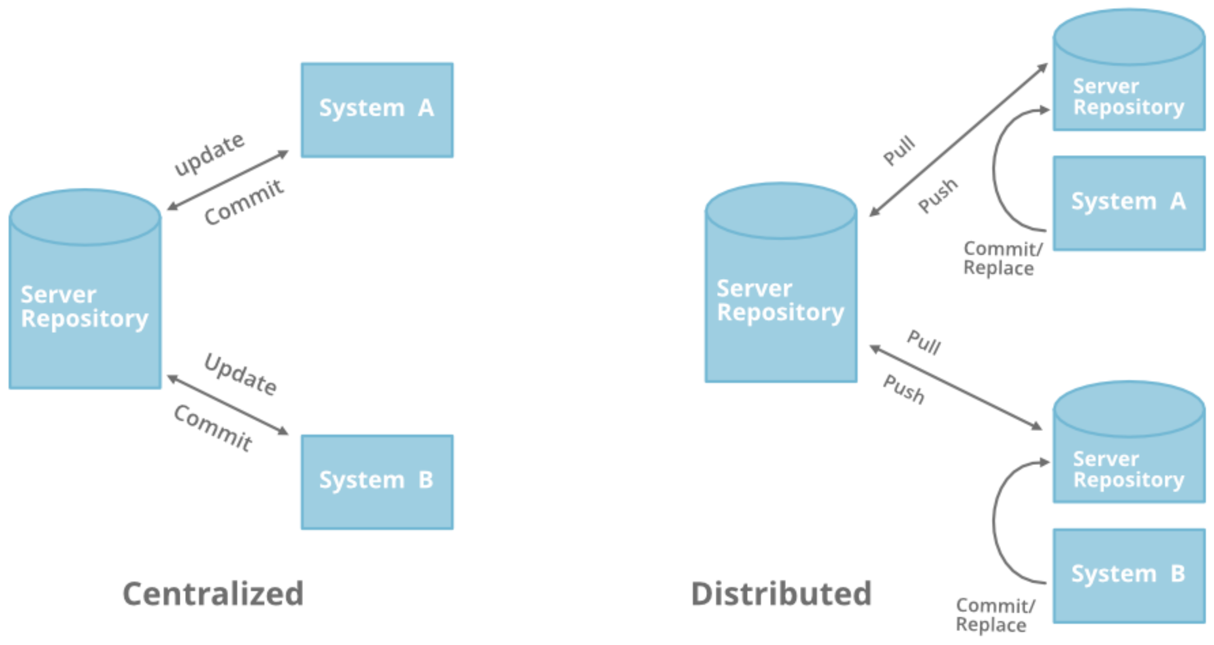
The other strategy of a VCS system is the **distributed** approach where you have your VCS system mirrored on every developers machine.

The benefit of this is:

- Allows users to work productively when they are not connected to the internet as they have a mirrored version of the codebase.
- There is no need for constant communication with others when making changes. If you make a change, you only need to communicate once you are in a position of sharing the content amongst your team.
- One key benefit is having a local copy. If for any reason a remote copy goes AWOL, you still have the code available locally.
- The other benefit, although somewhat cheeky, I get to work from home from time to time! Bonus!

Let me illustrate the Distributed VCS concept:

DRAW ON BOARD:



## Git

One of the main champions of a distributed VCS is Git which a free/open source system. It was developed back in 2005 by the creator of the Linux operating system.

The majority of organisations/software houses now use Git as their underlying version control system.

You're probably wondering why it's called Git. Well, this is what the creator said himself:

**SHOW SLIDE**

> "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'."

### Overview

What is Git?

- Version Control System (VCS) for tracking changes in computer files.
- It is operating system agnostic, meaning it can be used on Linux, Windows. It has no specific language/framework. It simply stores files!
- Distributed version control - meaning that you can have multiple developers making changes to the same project and they don't need to be on the same network to do so.
- Co-ordinates work between multiple developers and tracks every single change to every file within your project.
- Git tracks the files, when they were changed and by who. This is a crucial part of Git as it helps to identify issues in files and when they were introduced, i.e. when tracking bugs.
- Revert changes back at any time. Really helpful as this can and does occur!
- Git has a concept of repositories. Local and remote repositories are used within Git. Local repositories are on your own machine and do not need an internet connection but if you want to **push** your changes up to a remote repository, then you will. We'll see examples of this soon.

### Concepts of Git

- Keeps track of code history.
- Git takes "snapshots" of your files.
- You decide when to take a snapshot by making a **commit**. This is demonstrated later in the section.
- Great thing about Git is you can revisit snapshots at any time.
- You can also ***stage*** your files prior to committing them.
- When you push changes to a remote repository, other developers can only then see your changes.
- You can also create something called a branch. Explained later in this section.

Note: It is much more beneficial learning the Git commands in this section and using them via a terminal window, initially, than using/relying on an IDE (such as IntelliJ, Visual Studio etc). This way you will understand how Git *acutally* works under the hood.

### Basic Commands

Initialising a local repository:

```
$ git init
```

Adding files to the "index":

```
$ git add <file1> <file2> ...
```

To check the status of the "working tree", i.e. which files you have added/committed:

```
$ git status
```

When you want to ***commit*** your files, All changes will be committed to the "index":

```
$ git commit
```

The following commands are for **remote** repository usage:

To push changes to a remote repository:

```
$ git push
```

When you want the latest changes from a remote repository, i.e. when a colleague has made changes and you need them:

```
$ git pull
```

To clone/copy a remote repository:

```
$ git clone <url>
```

**Practical**

First lets check we have Git:

```
$ git --version

git version 2.20.1
```

1. Create a new directory **bootcamp-2019** and under than create another directory and call it **fundamentals-1**.
2. Open this folder in an editor of your choice. For example Atom/Visual Studio Code.
3. Now lets initialise Git for our new folder/project:

```
$ git init
```

You should now see a **.git** folder inside the **fundamentals-1**.

If you're on windows and you do not see this in the Explorer, go to options on the folder, then the View tab and tick the box that says "Show hidden files" (or similar). You may also choose to show extensions of files as another option to tick.

You can also see this **.git** folder in your editor.

The **.git** folder is where Git stores all of the snapshots and indexes.

Now we have initialised the Git project, we can start using our commands from earlier on.

Before we do this, we must add our name and email address to Git so that we can mark all our snapshots, comments and files with our name and email. This will help others identify who has made the changes which is important as you'll see later.

To do this:

```
$ git config --global user.name '<your name>'
$ git config --global user.email '<your email address>'
```

1. Now create an **index.html** file within the **fundamentals-1** folder.
2. Now create a **css** folder within the **fundamentals-1** folder.
3. Now create a **style.css** file within the **css** folder.
4. Lets now add our **index.html** file by navigating to our **fundamentals-1** folder:

```
$ git add index.html
```

This will not do anything other than *stage* this file within Git.

In git there are three different areas in which work resides (DRAW ON BOARD):

- The working directory where our project is.
- The staging area where all the changes you have added through **git add** will stay.
- The repository where all your commits will end up. We'll use the command that gets the files into our repository later.

5. To see what we have added, we can use our *status* command:

```
$ git status
```

You should see the index.html file listed as well as the css folder and style.css file. Can you see that they are not under the same sections?

The git status command shows us the current state of where files or folders are.

6. Lets commit our file 'index.html' file:

- COMMITTING USING NO FILE AT THE END
- COMMITTING USING FILE AT THE END (create an overview.md) file to demonstrate.

```
$ git commit -m "Initial commit" index.html
```

7. Let's also add our remaining changes:

```
$ git add css css/style.css
```

**Git Add All**

We have so far added files using the 'git add XXX' way doing things. There is also another way.

This way is to use the following:

```
$ git add .
```

This basically says "add everything". Great! Not! BE CAREFUL doing this.

What if we accidentally added a set of secret files or sensitive information? Fortunately for us, there is a way around this:

#### Git Ignore

1. Create a new file inside the **fundamentals-1** folder and call it **.gitignore**. Note this files starts with a '.' (dot).
2. Create a new file called **notes.txt** and add some text to it.
3. Now add **notes.txt** to the **.gitignore** file we just created.
4. Lets also add our changes:

```
$ git add .
$ git status
```

Did you notice the staging area? The only file should be **.gitignore**!

You can also:

- Ignore directories (and their contents)
- Use wildcards

Let's try this out:

1. Inside the **fundamentals-1** folder and create another folder and call it *Folder1*.
2. Now add *Folder1* to the **.gitignore** file.
3. Lets also add our changes:

```
$ git add .
$ git status
```

After typing **git status** you should see only the **.gitignore** again!

### Git Reset

There are times where we make loads of changes and think "Oh heck, I want to start again".

Whenever we create a git repository, git uses a concept of HEAD. This the last point in which code was committed and pushed to a repository.

Let me demonstrate: DRAW ON BOARD.

1. Create a new file 'resetme.md'.
2. Add some content to the file.
3. Add the file to git by staging it.

Now what if we wanted to go back to start and undo the staging of the file **resetme.md**?

```
$ git reset HEAD resetme.md
```

Now run a status check:

```
$ git status
```

You will find that that file is no longer there.

I'll also show you how to can reset wholesale changes which you may do in the future.

**Git Log**

So far we've been working in git locally and have performed several commands. What if we wanted to see a trace of everything we've done?

Simple, we can use the **git log** command:

```
$ git log
```

A much better way of looking at the git log would be to use the following command:

```
$ git log --oneline
$ git log --oneline --graph --all
```

### Aliases

Only if they all use macbooks.

### Branches

Git you could think of conceptually as a tree. Git itself has this concept of a main branch, so the "trunk" of the tree. When we code, by default, all code would reside in the **master** branch as that is what you get out of the box from GitHub and the like.

When you make a branch, you are basically making a copy of the code from that point in time and giving that branch a name that represents:

- A feature
- A bug fix
- Some R&D/Spike

DRAW ON BOARD MAIN BRANCH AND BRANCHES.

So let's go through a scenario ...

> Let's say you're a developer and you've been working on say the JDSports website and you get asssigned a task of creating a new feature on the website which is to promote a superstar.

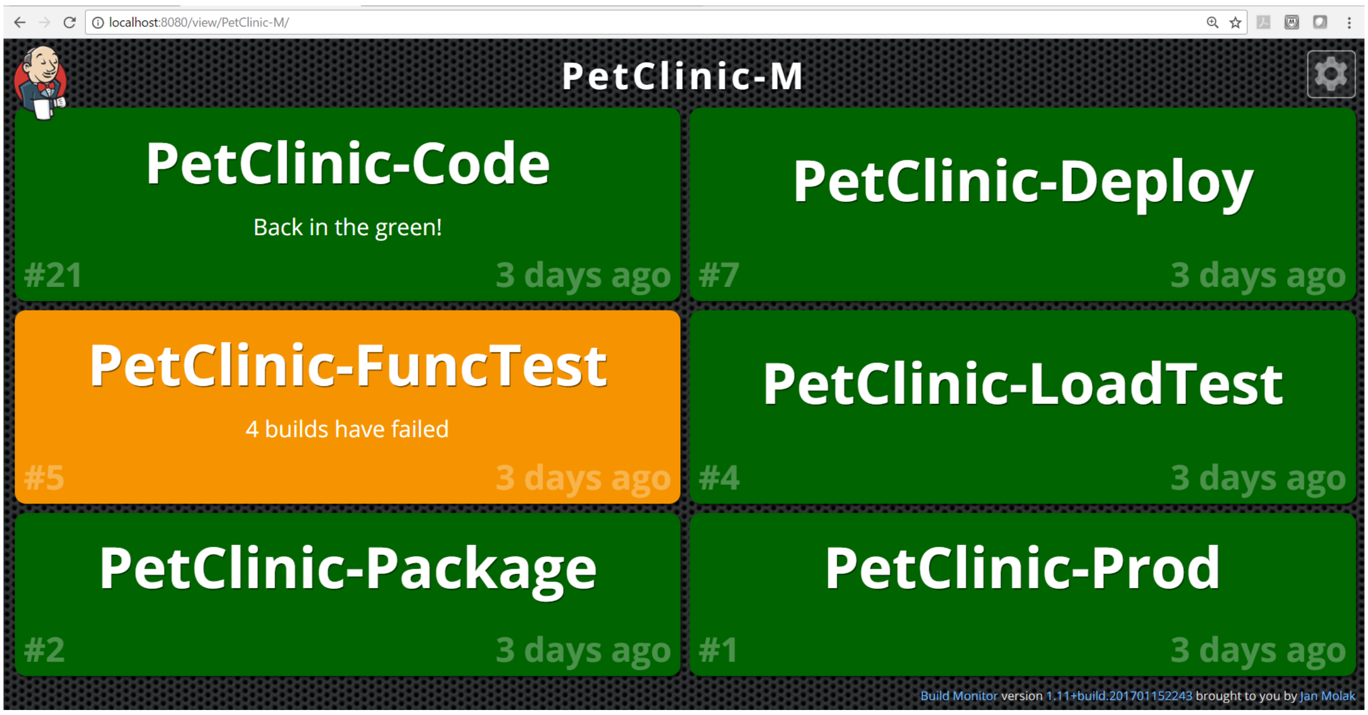
Now ideally you don't want to be making changes on code which may be changed by other developers in the same area. So to avoid this, or one way at least is to use a concept called **branching**.

A branch basically allows us to make a copy of the codebase at the point in time a source (this is usually the **master** branch (or another source branch) so we can then use it and work on it).

By default Git uses the **master** branch (you do not need to create this) but you may have noticed this whilst using Git in the previous steps.

**Different Branching Approaches**

- All on master
- Branches
- All development branches then merged weekly into master using automation.



**Practical**

Let's give creating a branch a go. To create a branch simply type the following command:

Firstly let's check which branch we are on:

```
$ git branch (to exit this type q)
```

(only for macbooks: `git branch | grep \* | cut -d ' ' -f2`)

```
$ git branch feature-login-form
$ git status
```

This command will create a new branch **feature-login-form** FROM the **master** branch.

Now creating a branch doesn't automatically switch to that branch as we can see this if we type:

```
$ git status
```

What do you see? It still has the asterisk against the **master** branch.

To switch to a branch, here the **feature-login-form** branch, simply type the following:

```
$ git checkout feature-login-form
$ git status
```

Let's make a change in this new branch:

1. Create a new file called 'login.css';
2. Also update the 'index.html' and type the following text:

```
# inside index.html:

<h1>Hello World</h1>
```

3. Add and commit these changes to the **feature-login-form** branch:

```
$ git add .
$ git commit -m "Added login css and updated HTML"
```

4. List your changes:

```
$ git status
```

Ok so let's switch back to the master branch:

```
$ git checkout master
$ git pull (always remember to pull to ensure we fetch all the latest
changes)
$ git status
```

You'll see that there is no 'login.css' file. If you open the 'index.html', you'll see the change from the previous steps is not there!

We can bring this in. Simply merge the change into your master branch:

```
$ git merge feature-login-form
```

Well done, you have successfully:

- Learnt basic Git Commands
- Created and used branches

---

## GitHub

### Overview

It's time to now work with remote repositories. This is where we use GitHub.

GitHub is simply a collaboration tool that allows you to use Git commands but through a UI (user interface). It also comes with a lot of useful features that will be covered in this session.

### Account Set Up

Go through:

https://git-scm.com/book/en/v2/GitHub-Account-Setup-and-Configuration

### Two Factor Authentication

- Make sure you set up SECURITY and Two Factor Authentication.
- Download APP Authenticator which generates a temporary 2FA passcode.

**For Local Development**

Steps for setting up 2FA local development

If you have issues pushing to the repository due to **"Permission Denied"**:

1. Open **.git/config**.
2. Change the https url to **ssh://git@github.com....**
3. Save the config.

### Creating Repositories

> DEMO: Let's create a new repository in GitHub and call it "BootCamp-2019".

Make it a public repository. You can also make a private repository but it's not free.

- Public - Anyone can see it and you can choose who can commit to it.
- Private - You can choose who can see it and who can commit to it.

### README file

Once created, follow the steps of initialising the Git repository as per the instructions given to you by GitHub under the section '...or create a new repository on the command line'.

```
Create a personal profile section in a README.md file.
```

### Cloning Repositories

You will be using this function heavily in the future where you will be direct contributors to a project.

To do this we use the **git clone** command.

Let's try this using both the command line and GitHub itself.

> DEMO: flight information

### Creating Branches

We have created branches via the command line. You can also do the same using the GitHub UI. This is where we'll see the full power of GitHub and it's usage.

Let's use a sample repository:

> DEMO: music lyrics repo

Now I need a few volunteers. Use randomiser to select 3 people.

1. Create a new folder on your device (it will only be for a temporary demo).
2. Add each chosen person as a Collaborator.
3. Clone this repository on your machine using the git clone command.

## Merge Conflicts

When we work on branches, up to now we've worked on a **master** branch.

## Fork Off

It's time to fork off.

We can create repositories for our personal use and add files/content which is great.

However, there may be a time where you want to contribute to a project. There are loads of projects in GitHub.

However to contribute to projects that you don't have access to, you have to "fork" the repository.

when you fork a project, GitHub makes a clone/copy of that project which becomes your own.

Agile Manifesto - GIVE EACH PERSON A NUMBER 1-12.

https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/

```
Agile-Co is looking at empowering staff with the agile manifesto but they
have yet to create a set of principles for them to abide by.

To help them with this, the HR team have found that there are 12 Agile
manifesto principles.

You have been given a number on your desk.

Challenge:

FIND THE PRINCIPLE THAT MATCHES YOUR NUMBER AND CONTRIBUTE TO THE
PRINCIPLES SET OUT FOR AGILE-CO TO USE.

YOUR CHALLENGE MUST INCLUDE:

- FORK (PROJECT agile-co-example)
- MAKE NECESSARY CHANGES
- REQUESTING A PULL REQUEST
```

PAIRED - Missing Lyrics!

```
Sony Music have asked Adele to send her them her new song. However she's
accidentally dropped all the pages for the script and they've all become
scrambled.

Adele has asked if you could help out and submit the lyrics so that she
```

```
can accumulate them and submit them as a unit back to Sony.

Your task:

To fork the repository: music-lyrics

To create a new text file and add the necessary lyrics of the song.

Submit the lyrics as a pull request, for acceptance(?).
```

### Pull Requests

When we move to branches, or even work on a master branch in git, we create something called a pull request.

Why is it called a pull request and not a push request?

A "pull request" is you requesting the target repository to please grab your changes. A "push request" would be the target repository requesting you to push your changes. When you send a pull request, you're asking (requesting) the official repo owner to pull some changes from your own repo. Hence "pull request".

Why do we need pull requests:

- Interested parties can review your changes, increases BUS FACTOR levels and awareness.
- Encourages discussions amongst people
- Also highlights potential clashes incase multiple people are working on the same stream.

> DEMO: Let's create a new branch and call it, **fundamentals-1**.

```
Add content to the README.md file to describe the session and commit your
changes.
```

Push the change and create a pull-request with the group.

> DEMO: Go through the process of a pull request and code review.

### Code Reviews

Demonstrate pull requests.

**THROUGHOUT THE BOOTCAMP WE WILL CREATE A NEW BRANCH FOR EACH DAY AND COMMIT CHANGES THROUGHOUT THE DAY.**.

### Dependabot

Mention how auto pull requests can fix issues.

Show example (show email).

Quiz: https://www.flexiquiz.com/SC/N/764d2033-efaf-4b4e-820a-070e3d0e8f60