Notes about CMD and Other Info

Interpreted languages

* Source code
  + Interpreter
    - Program runs
* Interpreters read the source code, parse it and follow the instructions one by one
  + An example is Python
    - Remember how you do it in the Python Shell, send one command then enter, then the other and so on and so forth
* Benefits
  + Easy for developer
    - Source code can be changed and run immediately
    - Console can be provided
  + Works on many pc´s
    - Interpreters are different based on the pc that the interpreter is running on
    - The result of running on different pc´s will be the same for a language even though the interpreters are different
    - This allows for each pc to parse source code differently making it more efficient
* Disadvantages
  + Overhead
    - Parsing the source code requires CPU processing and memory
    - As it is done constantly, the program runs slower
  + Deployment issues
    - Source code must be given to change the program
    - Interpreter has to be given

Compiled languages

* Source code
  + Compiler
    - Native Executable
      * Loader or OS
        + Program runs
* Compiler reads the source code, parses it, but doesn´t perform the instructions immediately
  + Converts to machine code
    - Machine code and metadata put into native executable
      * The OS loads and runs the native executable by loading and tell it to run
      * The other info in the file tells the OS what memory is needed and how to share the CPU with other programs
* Benefits
  + No extra overhead
    - Program runs in machine code
    - Can run directly on the CPU with no needed middle steps
  + Safe deployment
    - Give native executable
    - Machine code is hard to understand and change
  + Lowest level access
    - Closer to hardware
    - More control over the machine code generated
* Disadvantages
  + Hard for developer
    - Changed source code has to be recompiled which is slow
    - Finding errors involves looking at the source code that gave the machine code error
  + Limited PC´s
    - Native executable made is specific to OS and machine code
  + Less Safe
    - Directly runs on CPU
    - Can crash PC
* Examples
  + C++
    - High performance or low level access needed
  + Every program has to run in machine code, an interpreter also

Cross Compilation

* Source Code
  + Cross Compiler
    - Guest Executable
      * Path 1:
        + Simulator

Program runs on local machine

* + - * Path 2:
        + Remote Loader

Program runs on another device

* Used when writing a program that runs on a different kind of system to the one you are writing the program on
  + Behaves the same as compiler but the executable made can´t be run on the developing device
  + Simulator can do this on the developing device
  + Or make use of a Remote Loader to load the program to the end device

Intermediate Compiled Languages

* Source Code
  + Bytecode Compiler
    - Bytecode Executable
      * VM
        + Program Runs
* Combines interpreted and compiled languages
  + Bytecode Compiler reads source code, parses it and makes it bytecode
    - Bytecode is similar to machine code that is very low level, not human readable
    - Bytecode can´t run on CPU
      * Bytecode is something that is made by the creator of the language
      * VM runs it
      * VM may interpret the bytecode or compile it and run the machine code – JIT
* Benefits
  + Minimal overhead
    - As fast and efficient as it can be
  + Easier to debug
    - Special VM debugging can be used
  + Safer
    - VM´s can refuse commands
  + Can run on different PC´s
    - VM´s are specific to a PC and OS but can run any bytecode executable
* Disadvantages
  + VM Overhead
    - Not 0 overhead
    - Interpreter VM´s slow down the program
    - JIT compiler VM slows down program start-up
  + Hard to deploy
    - Bytecode executable and VM needed
    - Bad deployment gives many copies of unneeded VM´s
    - De-compilers can´t give source code
* Java is an example of this
  + Java compiles to JVM
    - Java Virtual Machine

Part 2

Usage of CMD

* Control the OS
* When GUI is not available
* Automation
* When tasks are awkward to do with GUI

GUI v CMD

* GUI and CMD might have to be used together (Think PEC)
  + Doing it only with the GUI can be hard and not easy
  + So most language tools run on the CMD and if a GUI is provided then the CMD functions are called (Think PEC and Design Patterns)

Using the CMD

* An input to the CMD is made up of a command, a space and then parameters
  + Each parameter is differentiated with a space
    - Eg: copy test.txt test2.txt
  + If the parameter value has a space in it such as Work flow copy.txt then the whole phrase is put into speech marks
    - “Work flow copy.txt”

Basic Operations

* List files in a directory
  + dir
* Change directory
  + cd
* Copy file
  + copy
* Delete file
  + del
* Make directory
  + md
* Remove directory
  + rd
* Opening a file with an app
  + Start
* Rename a file
  + Rename
* Move files between directories
  + Move
* Output contents of a file
  + Type
* Output contents of a file with pauses
  + More
* Output a message on the console
  + Echo
* Show the logged in username
  + Echo %USERNAME%

Files in directories

* The current directory is seen in the CMD
  + If you perform an operation on a file, it is assumed that it is in the same directory
    - To access files in different directories, change the directory or use pathnames
    - Pathname
      * Directory \ Name of File
        + Eg: work \ test1.txt

This is the file test1 in the directory work

But work is part of the current directory

* + - * + Eg: ..\ test.txt

This is the file test in the directory 1 above the current directory

File System

* On all OS´s there is 1 directory which has every other directory in it
  + This is the root directory
* On Windows different storage devices has their own separate file systems and their own root directories

File System Structure

* A storage device has a drive letter
  + Eg: C:
* Hard disk is Drive C
* Other drives like USB, CD are drives D, E, F
* An absolute pathname starts with the drive letter and then the directories
  + If you want to perform an action on a directory or file in another drive (storage device) you type the name of the drive directly in the CMD
  + Can´t use cd command to change the drive you operate on
    - Eg: cd C:
      * NOT VALID
    - But C:
      * VALID
* The way that the file system structure is organized on Windows 10 is
  + C:\Users\Username
  + C:\Users\Public

Absolute Pathname

* Exact location of a file based on the root directory
* The absolute name comes from the fact that is doesn´t get affected by the current directory
* But it may not be portable between machines
  + Remember website upload and transfer

Windows Directory Tracking

* Windows tracks the current directory individually for each drive
* Entering
  + H:
  + Cd work
  + C:
    - This results in the C drive active
    - But work is the active directory on Drive H
* If you then type copy \*.\* h:
  + Copies all the files in the current directory to work on Drive H
* If you enter cd h:\work
  + This makes the directory work the active one on drive H, but the actual H drive is not active
  + The active drive is still the one you are in
  + To make H the active drive you just have to type H:

Running software from CMD

* Every program is stored as an exe
* Typing the name of the exe in the CMD, runs the program
  + The exe has to be in the active directory though
* To run the program you give a pathname or the file must be in one of the directories in the search path
  + To view the path just type path
* On windows modern software like Office doesn´t add itself to the path
  + This is because it is thought that it will be run from the desktop
    - Or the start command

Wildcard

* To perform an operation on many files use a wildcard
* ? in a file name means any character
* \* in a file name means a set of characters
* Giving the name of a directory instead of a filename means all the files in that directory

Wildcard 2

* Copy \*.\* backup
  + Means copy all the files in the **current directory** to the subdirectory backup
  + When speaking about files it doesn´t mean shortcuts (Eg. Desktop)
* Copy \*.\* g:\
  + Copy all files in the current directory to the root directory of another device
* Copy new old
  + Copy all files in directory new to directory old
* Copy work\*.\* work
  + Copy any file with the start name as “work” to directory work
  + TO DO IT FOR THE NAME OF THE FILE DON´T ADD A SPACE BETWEEN THE LETTERS YOU ARE LOOKING FOR

Wildcard 3

* Copy work?.\*. work
  + Copies all files with the name work and exactly one more character
  + To the directory work
* Copy \*.xls work
  + Copies all files with the .xls extension to the directory work
* Copy \*.xls\*.oldx
  + Copy all files with the extension .xls to the same directory but change the name for it to be \*.oldx

Switches

* Apart from normal parameters there are extra ones called switches
* In Windows usually they are seen with a / before them
* Some switches have parameters inside themselves
* /s
  + Subdirectories
* /r
  + Recursive
* These 2 mean the same
  + What they do is affect the directory you perform the operation on and its subdirectories

Wildcard 4

* xcopy /s \*.\* ..\backup
  + copy all files in current directory and its subdirectories into the directory backup which is one above the current directory
* dir moose.txt /s
  + List all files with the name moose.txt in the current and its subdirectories
* Del \*.\* /s
  + Delete everything in the current directory and all subdirectories

Command Line Part 2

* CMD programs have plain text input and output
  + Easy for a another program to generate or read in

Input and Output redirect

* > filename
  + Instead of outputting, send here
* >> filename
  + Done if the file exists, and you want to add to it
* < filename
  + Done if you want input to come from a file

Programs at the CMD

* Run python by saying
  + Python filename
* To do this python has to be on your path

Output channels

* Default > redirects to standard output channel
* A 2nd channel is the error channel
  + User´s sees the output even if user has a redirect
* When not using redirection there is no way of telling which channel is which

Exit code

* Every programs returns an exit code to the OS
* Normal value is 0
* But CMD programs can set exit codes and they can be read at the CMD
* To add an exit code, type
  + Sys.exit(1)

Scripts

* Write programs in text file and run from cmd
  + Scripts or batch files
* To create one, enter some commands in a text file and save it as .bat
* You can use any of the commands you would use in the CMD
  + Dir, copy and so on
* The echo command prints out a message
* Variables can be used in cmd also
  + To do this first enable the setlocal enableextensions line
  + Eg: set name = 23
    - If the value assigned to the variable is not a value, add a switch to the value
    - Eg: set /a sum = 1+2+3
    - If you want to get the user to type some information then you do it with the p version of the set command
      * Eg: set /p name = What is your name?
  + To refer to the value of a variable, use the %-% syntax
    - In CMD you can use the value of the variable anywhere
      * Eg: set ugh = ho Test
      * Ec%ugh%
        + This makes it Echo Test

Loops

* Loops were added to go through a set of files
  + So the loop is given using the filename wildcard notation
    - You can perform 1 command on a set of files
    - Eg: for %x in (\*.doc) do copy %~x wordfiles
      * Copies all .doc files in the directory to another directory
      * The %% is used to show the loop variable
      * %~ is used when the variable holds a filename
  + You can also do numbered loops
    - For /l %x in (1,1,5) do echo %x
    - The numbers in the brackets are
      * Where to start
      * Jump value
      * Where to end

If´s

* Checks for equality of strings
* Check existence of files
* Checks error code of last program
* Checks numbers if converted to strings
  + Eg: if %name% == Mark echo Hello

Labels and Goto

* :loop
* Set /p name=what is your name?
* If %name==Mark echo hello
* If %name%==Mark goto ok
* Goto loop
* :ok

Using another program

* :loop
* Set /p name=Type the name of a Python program to run?
* If not exists %name%.py goto missing
* python %name%.py
* If errorlevel 0 goto ok
* Echo Something went wrong
* Goto:eof
* :missing
* Echo that program doesn´t exist
  + Asks the user for the name of a program
  + Checks if the program exists
    - If it doesn´t then go to the missing section of the script
    - If it exists run the program
      * If the program´s error code is 0 then go to ok section
      * If not okay continue

Using Parameters

* Bad practice to ask user for inputs
* To use parameters you have to use the % syntax with the number of the parameter
* Echo you typed %1
* Echo then you typed %2
  + When you then open the script you have to give it the name of the file and then the parameters
  + Eg: cmd.bat kobe Bryant
    - Prints:
      * You type kobe
      * Then you typed Bryant

Lecture 2

Files and File types

* Type of file given by extension
  + Or type indicator inside the file
* Windows needs file extensions
  + File manager hides them but you can change this

Media containers

* File formats can be used to contain other files
  + Container / archive formats
* Media container format
  + Keeps parts of a media piece together
    - Eg: Audio and Video for a movie encoded differently
      * Eg: Adobe Acrobat Container - .pdf (text and image)
* Application container format
  + Keeps parts of a piece of software together
    - Machine code + data files or media
      * Android packed application - .apk
      * Java packed application - .jar
* Archives formats
  + Keep sets of files compressed and combined
    - .zip, zipx

Text Formats

* Text files have an encoding
* All data is stored as numbers and each character in text is mapped to a number
  + A = number 65
  + a = number 97
* ASCII is the standard
  + Only supports some languages not all
* 1987 Unicode was introduced
  + All alphabets in 1 coding standard
* Large numbers take more space to store in binary.
  + ASCII limited to 255 chars each
    - Stored in 1 byte
  + Unicode stores 128,237
    - Stored in 4 bytes
* Unicode is not stored in 4 bytes though
  + UTF allocates a minimum number of bytes to each character
  + If the character can´t be stored in this minimum then send a signal byte meaning that the character stored as multiple bytes
    - UTF-32
      * 4 bytes
    - UTF-16
      * 2 bytes
    - UTF-8
      * 1 byte
      * Similar to ASCII

Other types

* Type of the file can be determined using MIME
* MIME is indicated by a string inside the file or sent with it – Content-Type
* Windows uses CLSID
  + Long hex numbers
  + Easy to make unique
    - If program makes its own type of file easy to make random ones

Formats to write software

* When writing software we use file formats that aren´t common
* Source code and executable are most common
* Exe files have program changed to machine code / virtual machine code and other data to interact with the OS
* Windows executables have the .exe extension
  + Executables for the JVM have the .class extension
  + The extension of an .exe file doesn´t say what OS will run the file
    - Or what machine code it uses

Source Code

* Source code files are given an extension based on their language
* Source code can usually be edited in a text editor
  + The extension is added to know which interpreter / Compiler to use
* Most languages store SC in UTF8 or ASCII
* Eg: Python - .py, Java .java, C - .c

Libraries

* Most programs use libraries
  + Blocks of code which can be re-used
    - Used to simplify programming
    - Programmer avoids writing same code again
* Libraries can be given as source code
  + Additional source code file together with source code
* Or given as object code
  + Writer doesn´t want to give you the source code
  + Pre-compiled machine code
    - Speeds up compilation as library doesn´t have to be changed to machine code

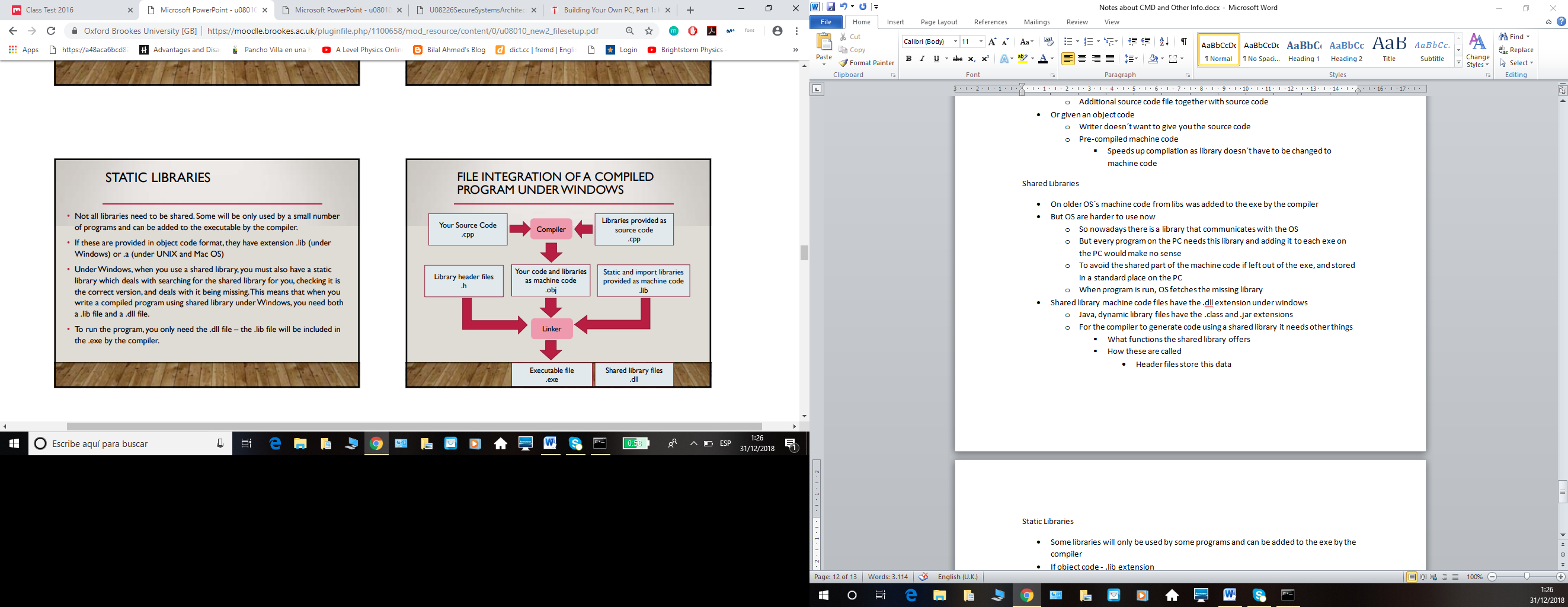
Shared Libraries

* On older OS´s machine code from libs was added to the exe by the compiler
* But OS are harder to use now
  + So nowadays there is a library that communicates with the OS
  + But every program on the PC needs this library and adding it to each exe on the PC would make no sense
  + To avoid the shared part of the machine code if left out of the exe, and stored in a standard place on the PC
  + When program is run, OS fetches the missing library
* Shared library machine code files have the .dll extension under windows
  + Java, dynamic library files have the .class and .jar extensions
  + For the compiler to generate code using a shared library it needs other things
    - What functions the shared library offers
    - How these are called
      * Header files store this data

Static Libraries

* Some libraries will only be used by some programs and can be added to the exe by the compiler
* If object code - .lib extension
* When using a shared library on windows you also have a static library
  + Deals with
    - Searching for shared library
    - Version
    - Missing from machine code
  + So when writing a compiled program in Windows you need .dll and .lib file
    - To run the program only need the .dll file
      * The .lib file is included in the .exe by the compiler

Compiled program under windows



* Your source code (.cpp) and libraries (.cpp) given as source code are sent to the compiler
  + They become machine code
  + Together with header files (.h) and static libraries given as machine code (.lib)
    - These are all sent to the Linker
      * Linker makes it an executable (.exe) and is paired with the share library files (.dll)

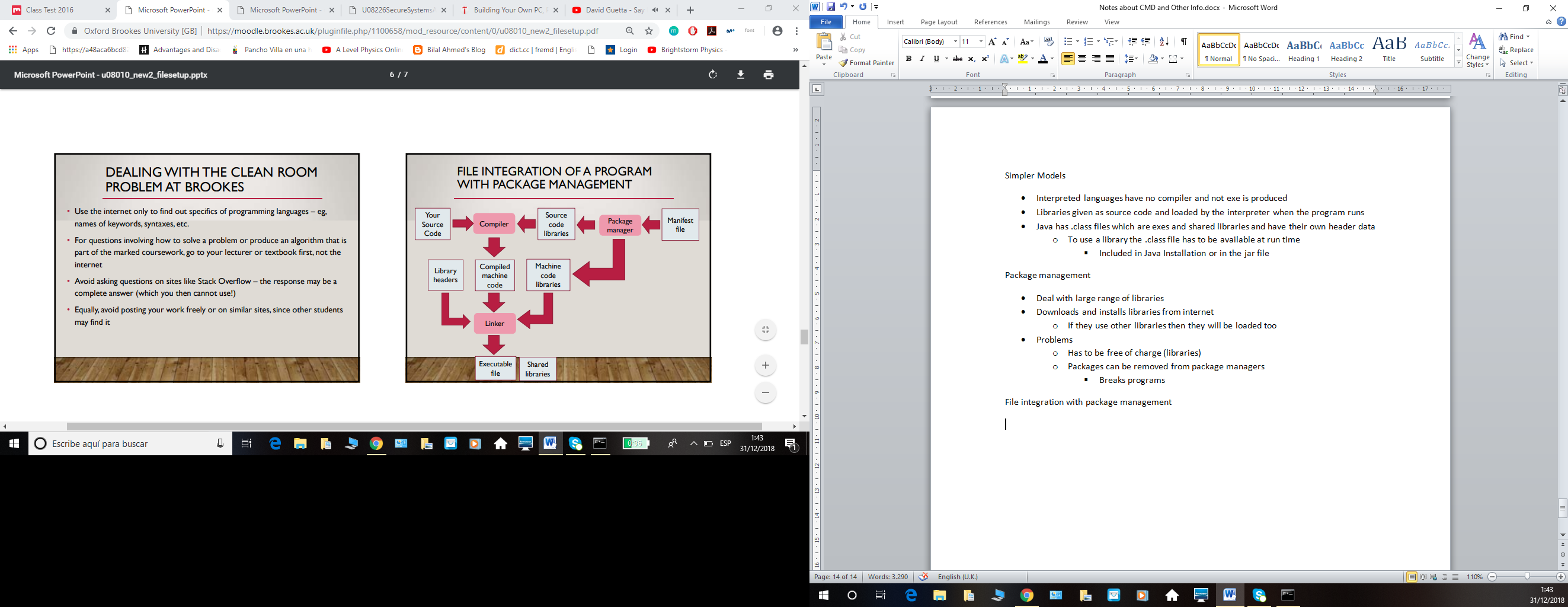
Simpler Models

* Interpreted languages have no compiler and not exe is produced
* Libraries given as source code and loaded by the interpreter when the program runs
* Java has .class files which are exes and shared libraries and have their own header data
  + To use a library the .class file has to be available at run time
    - Included in Java Installation or in the jar file

Package management

* Deal with large range of libraries
* Downloads and installs libraries from internet
  + If they use other libraries then they will be loaded too
* Problems
  + Has to be free of charge (libraries)
  + Packages can be removed from package managers
    - Breaks programs

File integration with package management



* Source code and source code libraries sent to compiler
  + Source code libraries includes manifest file (Config file letting others know how to download libs needed) from package manager
  + Compiler makes this into machine code and sends machine code libraries and library headers to the linker
  + Package manager manifest file is sent to the machine code library also
    - Linker makes .exe and combines with shared libraries

Organizing a large program

* Src
  + Source code
    - Divided into files or other folders
* Lib
  + Libraries
    - Source / machine code
* Include
  + Include files for libraries
* Build
  + Files produced by the compiler
    - Includes machine code and exe
* Res
  + Resources and other data files needed when program runs
* Dist
  + Holds program to be supplied to final user

Git Lecture U08008

VCS

* Multiple versions of plain text files
  + Usually code
* Github is based on snapshot version control

Repository

* Stores data about your files
* Stores actual files, their previous versions and info used by git to synch them
* It is put inside the .git directory
  + Hidden one by default
* When you create a repository the directory you specify to git is the working directory
  + Current version
  + Make changes on this one
    - Changes are copied to the repo by git
* Files in Git are
  + Committed
    - Repo has a copy of the current version
  + Modified
    - Changed since it was sent to the repo
  + Staged
    - Changed and ready to be sent to repo
      * But hasn´t been sent yet

Modifying

* Staging a file makes a copy
* Modifying a staged file won´t change the staged version
  + Won´t change the version ready to be sent to the repo
* When you commit to the repo, the staged file becomes committed
  + The one committed will be the one without the changes

Saving to Repo

* Type a commit message then commit
  + Then the changes are committed
* Your file doesn´t disappear once saved to the repo
  + Continue working on it
    - Rescan
      * Stage
        + Commit

Git keeps a record of every committed version

Undoing changes

* If your changes were a mistake you can go back to a previous committed version
  + Commit > Revert Changes
* To unstage
  + Commit > Unstage from Commit
* If you revert then you can´t go back to the newer file again

Version History

* If you store a commit, git adds it to the history in your repo
* A -> B -> C -> D
  + Many commits form this chain

Using History

* If you test your program and it works
  + Commit
* If you change a program and break it
  + Restore last committed version

Going back by more

* Git allows you to work on commit B and keeps C and D

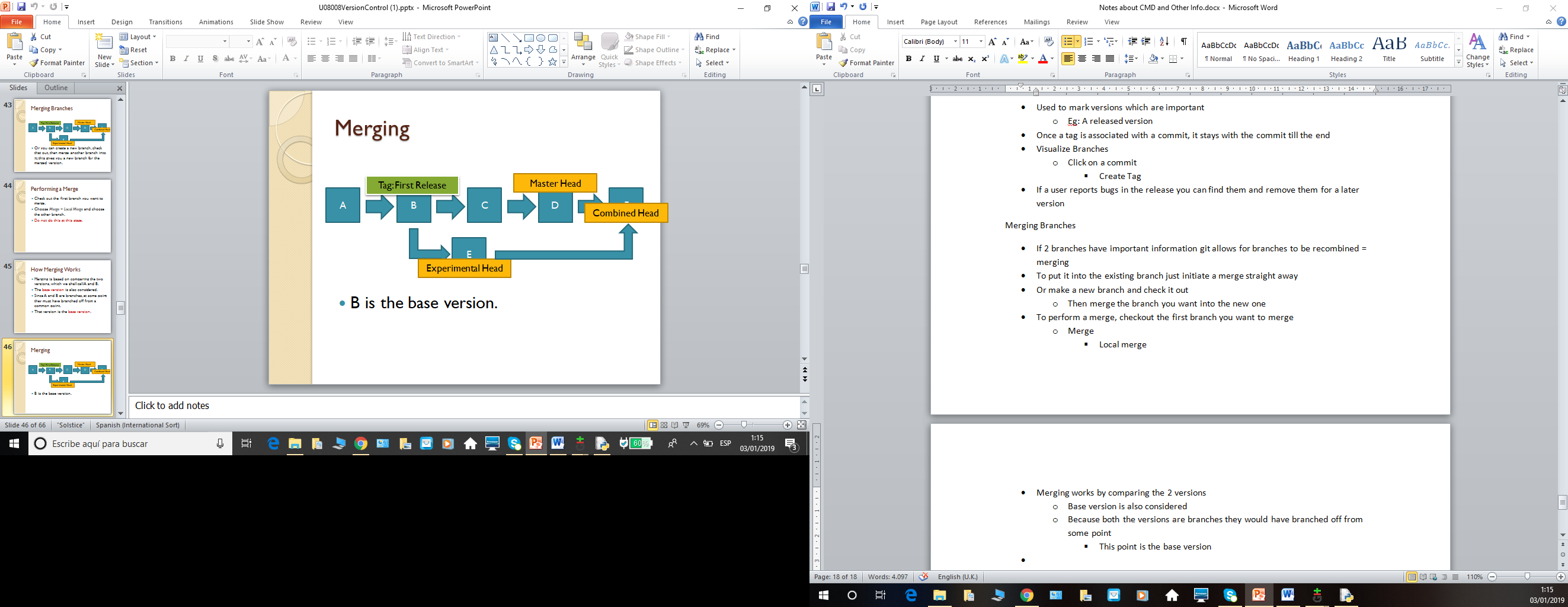
Branches

* A project can have many commit chains by using branches
  + Most recent commit of a chain is the head
    - First branch is the master branch
* To make a branch go to
  + Repository
    - Visualize All Branch History
      * Right Click
        + Create New Branch
* To make changes to any file on a branch switch your work directory to that branch
  + Branch
    - Checkout
      * Choose the new branch
* By switching to this branch the file will be in the state of the commit you made for that branch
  + Branch head moves to most recent commit
  + The red line is where the master branch diverges
* If changes are made that break you program but you might fix them later
  + Commit
    - Create a new branch at the last working version
* If you have 2 ways of solving a problem
  + Commit program before starting then create a new branch
    - Try idea 1
      * Commit
        + Restore to original branch
    - Create 2nd branch
      * Try idea 2
        + Switch between original and this one

Tags

* Used to mark versions which are important
  + Eg: A released version
* Once a tag is associated with a commit, it stays with the commit till the end
* Visualize Branches
  + Click on a commit
    - Create Tag
* If a user reports bugs in the release you can find them and remove them for a later version

Merging Branches

* If 2 branches have important information git allows for branches to be recombined = merging
* To put it into the existing branch just initiate a merge straight away
* Or make a new branch and check it out
  + Then merge the branch you want into the new one
* To perform a merge, checkout the first branch you want to merge
  + Merge
    - Local merge
* Merging works by comparing the 2 versions
  + Base version is also considered
  + Because both the versions are branches they would have branched off from some point
    - This point is the base version
* 
  + B is the base version
* If file unchanged from A and B stays the same
  + If file is changed from the base in A but not B the changed version is kept
  + If a file is changed in both and the changes are different a conflict occurs
* When a file is changed in 2 branches a person has to combine the changes

Working with others

* Remote repositories
  + You work on your local repository and your work becomes a branch in the remote repository
  + Others can merge with your branch to incorporate your work into theirs
* Usually only one person in charge of the repo will be allowed to update the master branch on the remote repo
  + You will push your changes to a branch
* If you are working on your branch and someone updates the master branch
  + Upload your branch anyway
    - Your code won´t have all the updates made to master
  + Update origin branch to the new one on the master (remote repo) then merge changes
    - Gives up to date version

Git from CMD

Setting up a repo

* Git init
  + Turns current directory into a git repo by creating the .git folder
* Git clone (URL)
  + Downloads a remote repo
    - Makes a directory with the project name and sets a copy of the repo inside

Checking in files

* Git add (filename)
  + Stages the current version of the file
    - You can also use wildcards like in CMD
      * Eg: git add .
        + Adds all files in the directory and subdirectories
  + If a file is not tracked then when added it does become tracked
  + The current version is staged
    - So if changes made to the file you have to git add again (rescan on git gui)

Gitignore

* Gitignore is a text file with a filename or wildcard name per line
* Git will not track or stage nay files listed in this textfile
* What to ignore
  + Files generated by other files
    - Machine code or bytecode
    - Program output files
  + Files that even when changed don´t change the program
    - Test input files
    - Preference files for an IDE
  + Any files that are to be kept private
    - Passwords
    - Personal data

Mistakes

* Staging a file you didn´t want to
  + Git reset HEAD <filename>
    - Unstages the currently staged version of the file
* Staging and tracking a file you didn´t want to
  + Git rm –cached <filename>
    - Removes copy of the file in the index
    - Stops tracking the file

Commit

* Git commit –m “MESSAGE”
  + Commits staged version of the file with the message given
* Git´s repo is just a datastore
  + When a version of a file is stored in it, it is given a key and the content of the file is stored using that key
  + Git can store lots of types of data
    - Blob is common and it is a whole file you commit
    - Git also stores other files helping the management of your repo
      * These are objects

Git object types

* Tree
  + Contents of a directory
    - List of filenames in a directory you committed
    - Keys to the blobs storing those files
* Commits and tags
  + Perform a commit of create a tag
  + Commit
    - Has the message, key of the tree holding the objects you committed and the key of the previous commit
* Refs
  + References to the most important commit
  + Heads
    - Store the key of the most recent commit of a branch
  + Tags
    - The key of the commit where the tag was created
* HEAD
  + Reference to the commit you are working on now

What commit does

* Looks at files in staging area
* Adds them to the DB
* Adds each directory to the DB as a tree
* Creates a new commit object
* Moves head to the new commit object

Recovering from mistakes

* Fix the directory and recommit
* Git commit –amend
  + Collapses your fixes to the previous commit
* Git revert HEAD
  + Actually undoes changes made in the last commit

Checking status

* Git status
  + What branch you are on
  + What files are in the staging area and which can be committed
  + What files are not tracked
  + What files are tracked and have been changed

History

* Git log
  + Shows history of commits on the current branch
  + Long hex number is the key
    - Used to refer to the commit if need to extract files from history

Recovering history

* Git checkout <Reference>
  + All files in the commit are copied into the working directory
* Reference
  + Key of a commit object
  + Name of the branch
    - Checks out the head of that branch
  + Branch~
    - Parent of that branch
  + Branch~2
    - Grandparent of that branch

Detached Head

* If you checkout a commit which is not the head of a branch this warning comes up
  + Means that any commits made will not get a reference
    - Because they aren´t connected to a branch head or a tag
  + To fix
    - Create a branch after detaching the head

Creating branch

* Git branch <name>
  + Doesn´t select branch as the current head
    - Use git checkout name to do this

Merging a branch

* Git merge <branch>
  + Follows the current HEAD, and the commit referred to by nama through commits to find the base head
  + Checks what files have been changed
    - If only 1 file changed copies the changed version to the working directory
    - If file has changed in both branches, a merge resolution file is written and put in the directory

Merge resolution

* Summary of both version of the file with changed sections marked
* To fix, manually edit the merge resolution file to fix conflicts then git add to confirm the merge and git commit
* Git merge –abort stops all of it

Setting up remotes

* Git clone
* Git remote add name “URL”
  + Adds remotes to an existing repo

Getting data from a remote

* Git fetch <remote>
  + Downloads data from the remote you don´t have
* Git merge <remote>/<name>
  + Merges data from named remote branch in the normal way
* Git pull
  + Does a fetch then a merge on the current branch

Sending data to a remote

* Git push <remote><branch>
  + Updates named remote branch to match your current HEAD
  + Git pull to merge work or create a new branch and push that if any conflcits