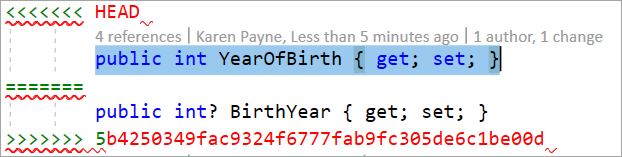
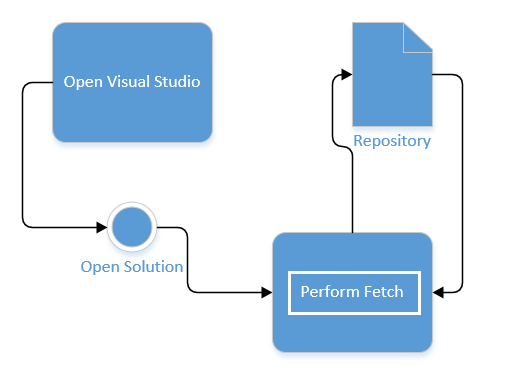


# Basic operations before starting any coding

* Always perform a GIT Fetch operations to ensure there are not any code in the solution repository newer than what is on your computer
  + If there are changes, view the changes, if there are changes which conflict with your code contact the developer who is responsible for these changes.
* If possible to perform a merge, perform the merge. In some cases Visual Studio can figure out the merge for you while in other cases you must go through each conflict and resolve them. Proir to resolving any conflicts, review each of the conflicts then if acceptable perform the merge. Once the merge has been completed run all available unit test (if any) to ensure changes didn’t break anything.  
    
  This is presented when Visual Studio can not perform an auto merge  
    
    
    
  

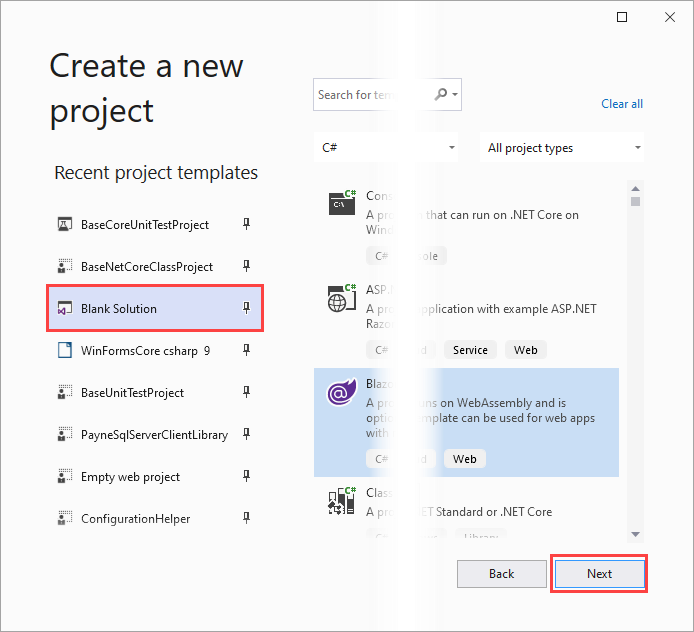
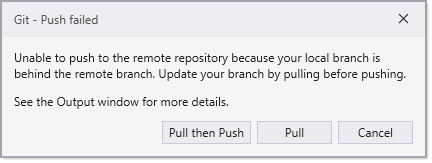
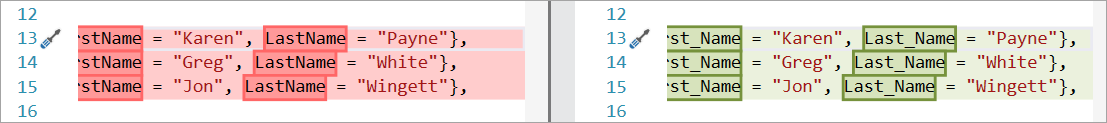
Basic flow  
  


# How to get an idea of auto-merge and manual merges

You will need to create a personal GitHub repository and work on the repository with two copies of Visual Studio. From there follow the instructions below. Total time should take between one to one and a half hour to complete.

## Downside

When working each week in class performing a merge is going to be complicated and may not be worthwhile. We will discuss alternates that exclude merging and keeping each person’s code but instead have the option to use my pre-done code instead.

1. Create a new Visual Studio solution using the blank template option under C:\OED\Dotnetland\VS2019\LearnGit.   
     
   Name of solution should not contain spaces in the name and should begin with an upper cased character e.g. LearnGit.  
     
   
2. Create the following solution folders
   1. Class projects
      1. Add a new C# .NET Core class project
   2. Unit test projects
      1. Add a new C# .NET Core test project
3. Under Class projects, in the class project created above
   1. Add a new folder named Interfaces
      1. Add a new Interface named IBase
      2. Add the code below after the namespace, between the Bob Hope brackets
   2. Add a new folder named Classes
      1. Add a new class named Person
      2. Add the code below after the namespace, between the Bob Hope brackets
         1. You will need to add a using statement to the Interface namespace
4. Login to your personal GitHub account
   1. Create a new repository, name it learn-git
   2. Copy the URL from your browser’s address bar into notepad
5. In Visual Studio, from the menu select Git then Create Git Repository
   1. Click Existing repository
   2. Make sure Local path is under C:\OED\Dotnetland\VS2019\LearnGit
   3. From notepad, paste in the URL into Remote URL
   4. Click Create and Push
   5. Under the GitHub page
      1. Next to About, click the gear, enter a short description
      2. Save changes
      3. There is a button Add readme, click it
      4. Add some text
      5. Click commit new file
6. In Visual Studio
   1. Open IBase
   2. Add /// above Id property, add this is the primary key
   3. Save the file
   4. Click on Git Changes window
   5. Enter a comment
   6. Click Commit all and push
   7. The following appears (because of the readme file created online)  
      
   8. Click Pull then Push
   9. Open Solution Explorer, note that even though the readme file was added it’s not seen.
   10. In Solution Explorer, right click, Add, Existing item
   11. Add readme.md, Visual Studio places it under a virtual folder named Solution items.
7. Open **another Visual Studio**
   1. Click on Git menu, Clone repository
   2. Enter the URL from notepad into Repository Location
   3. Enter a path with the original path above with Clone on the end e.g.   
      C:\OED\Dotnetland\VS2019\LearnGitClone
   4. Click Clone
   5. Under the file menu click Open, Project/Solution
   6. Enter C:\OED\Dotnetland\VS2019\LearnGitClone
   7. Click on LearnGit.sln, click Open
8. Go back to the first opened Visual Studio.
   1. Append text to the summary for Id property of IBase
   2. In Git Changes, add comment, click Commit all and Push
9. Go back to the second opened Visual Studio
   1. Click Fetch in Git Changes
   2. Click Git, view history, there is one incoming change
   3. Right click on the line, view commit details. Since this is just a comment
   4. Go back to the History window and click pull
   5. Open IBase and note the pull was successful
10. Has anyone noticed two readme.md files under solution items? We will get to that later
11. Go back to the first Visual Studio
    1. Under Classes folder add a new class Worker (see Worker version 1)
    2. Go to Git Changes, enter a commend
    3. Select command all and push
12. Go back to the second Visual Studio
    1. Perform a fetch
    2. Select Git menu, view history
    3. If incoming shows 0, click the refresh button
    4. Right click, view history, one file added
    5. Perform a pull
    6. See the new class in Solution Explorer
    7. Under Classes folder add a new class, Mocked (see Mocked 1)
    8. Under Classes folder add a new class Worker (see Worker version 2)
    9. Commit and push (as done several times above)
    10. Perform a fetch (as done above several times)
    11. View history, we have a new file and one which will become a merge? Or a discussion on which one to pick.
    12. Let’s do a commit all and get  
        
    13. All is good because there were no conflicts
13. Go back to the first Visual Studio
    1. In Person class, change FirstName to First\_Name, LastName to Last\_Name (this also changed code in the Mocked class)
    2. Commit and push
14. Go back to the second Visual Studio
    1. Perform a fetch
    2. View history  
         
       
    3. Do a pull we get  
         
       
    4. Code is updated to reflect the merge in regards to properties in (b)
    5. Take the underscored out of first and last names
    6. Commit all and push
    7. Do pull in first Visual Studio

*Git is good at automatically merging file changes in most circumstances, as long as the file contents don't change dramatically between commits*.

Okay, let’s get into a **merge conflict**.

In either or project, add a string property PostalCode to Person than in the Mocked class set the properties. Commit all and push

In the other project add an int property PostalCode to Person than in the Mocked class set the properties. Do and commit all and push. We get a dialog, select pull.

We get (just remember, seeing <<<<HEAD means to use the merge editor)

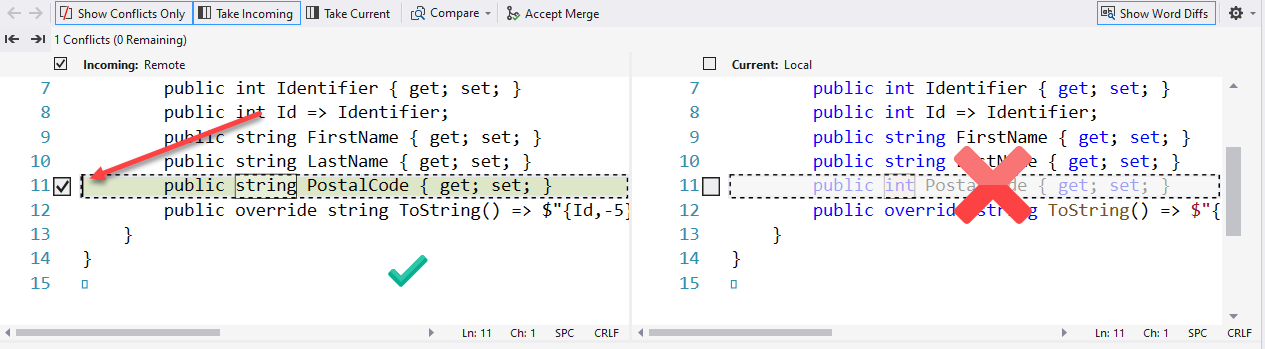


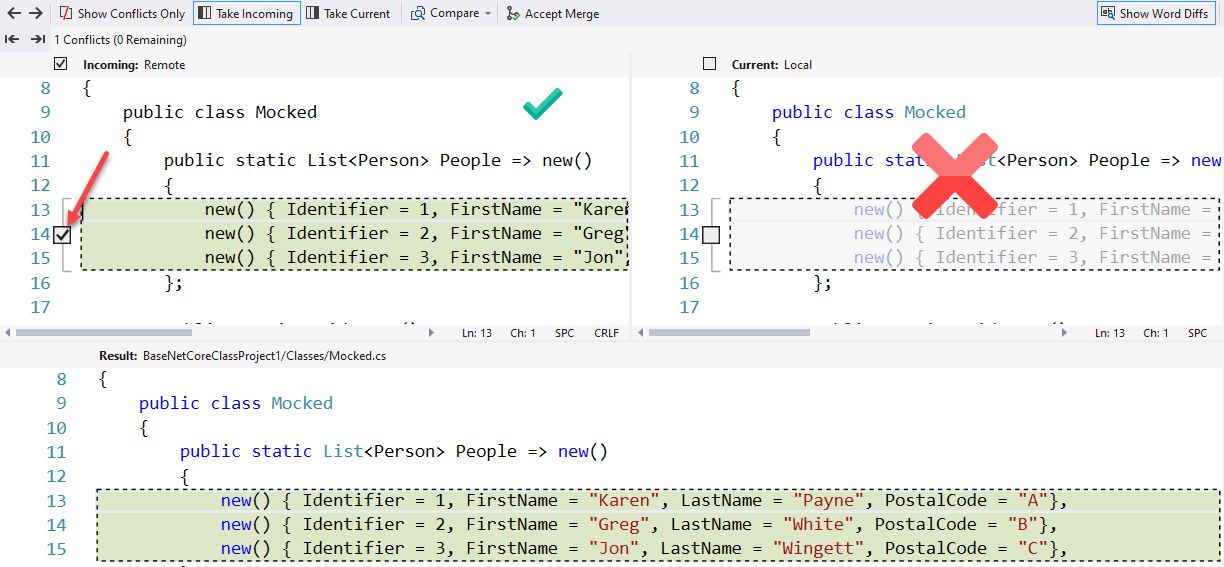
Open merge editor



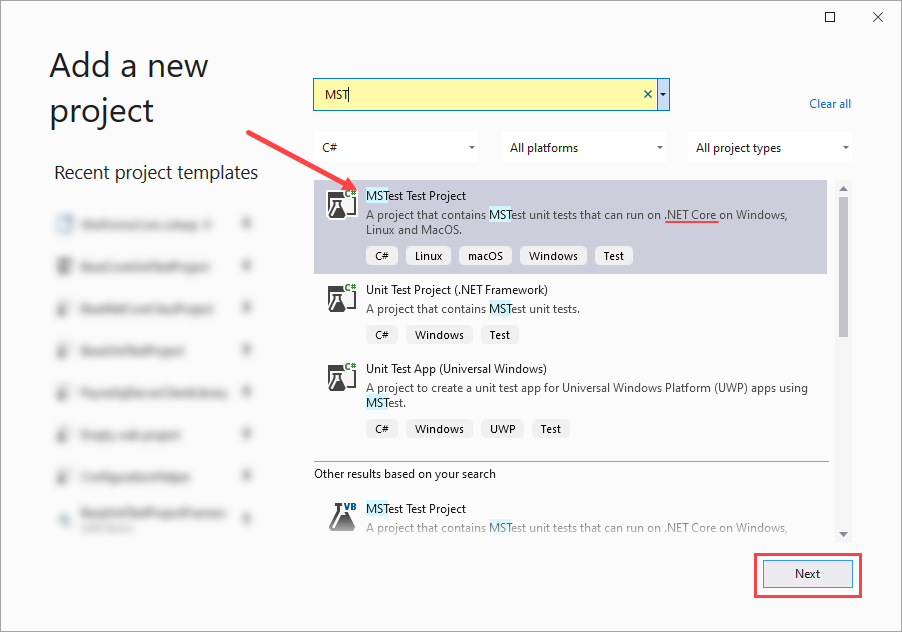
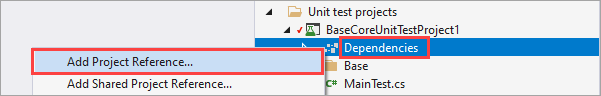
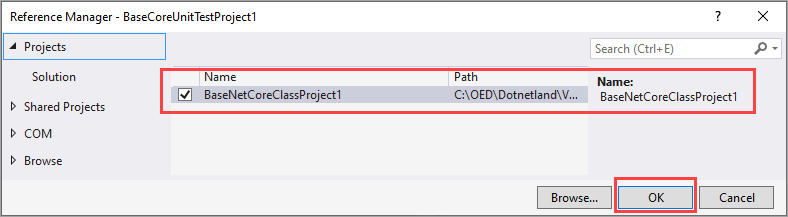
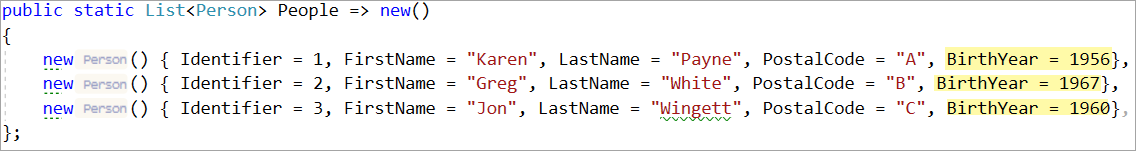
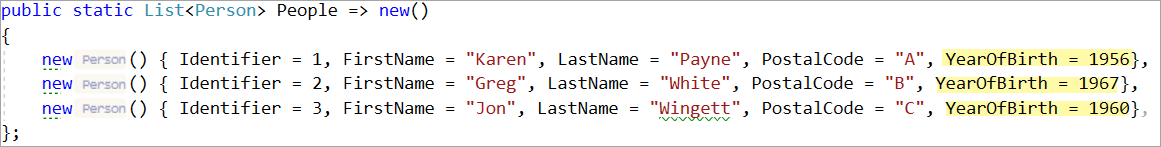
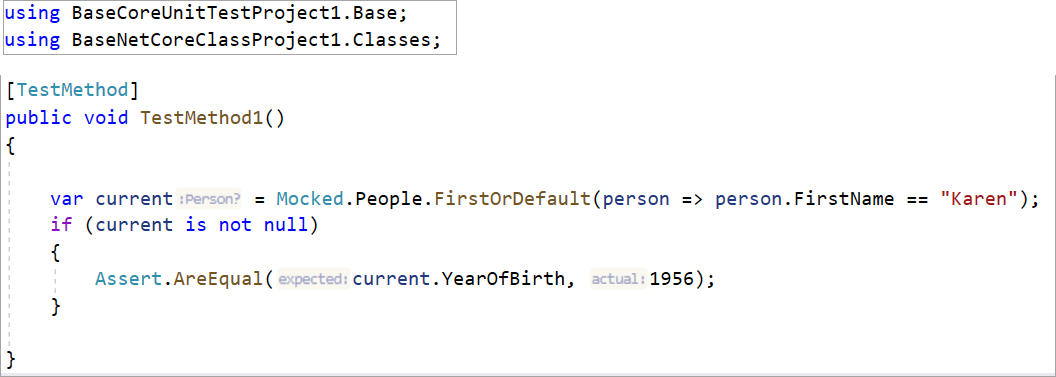
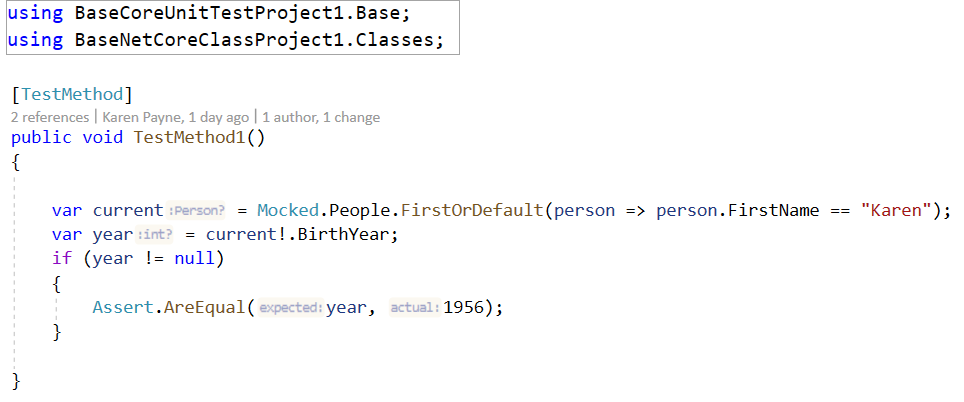
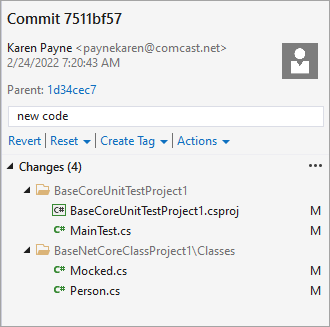
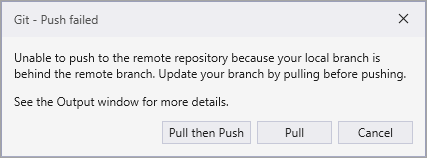
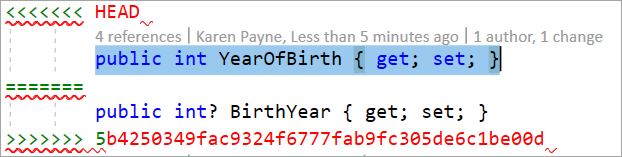
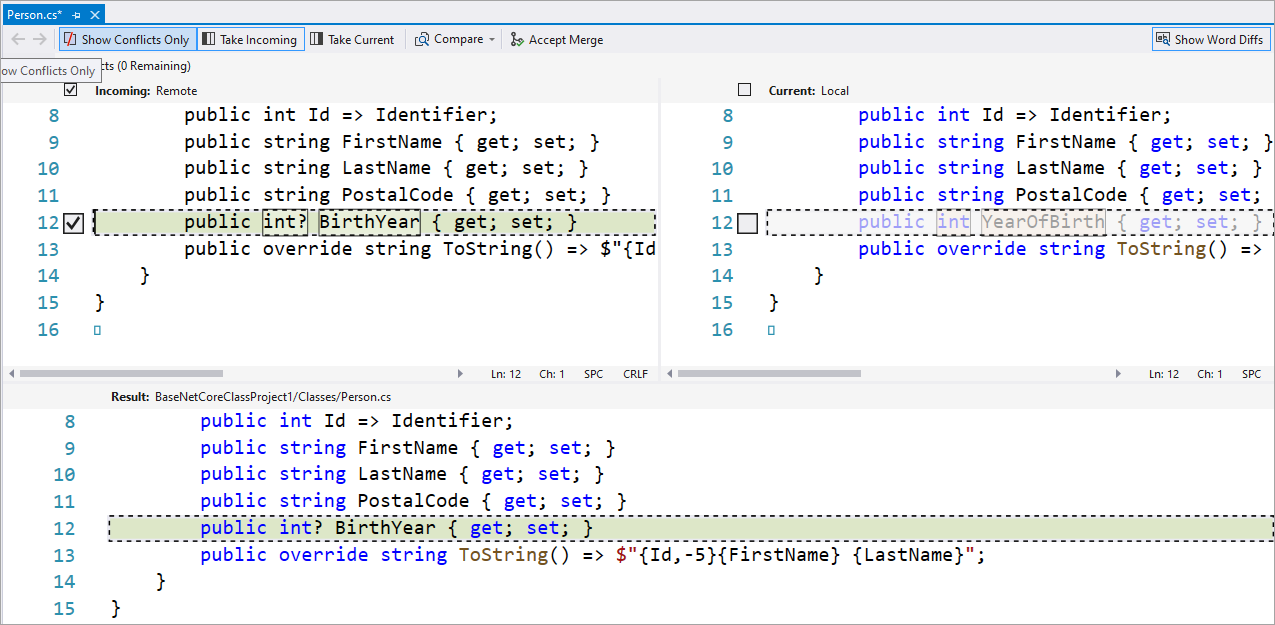
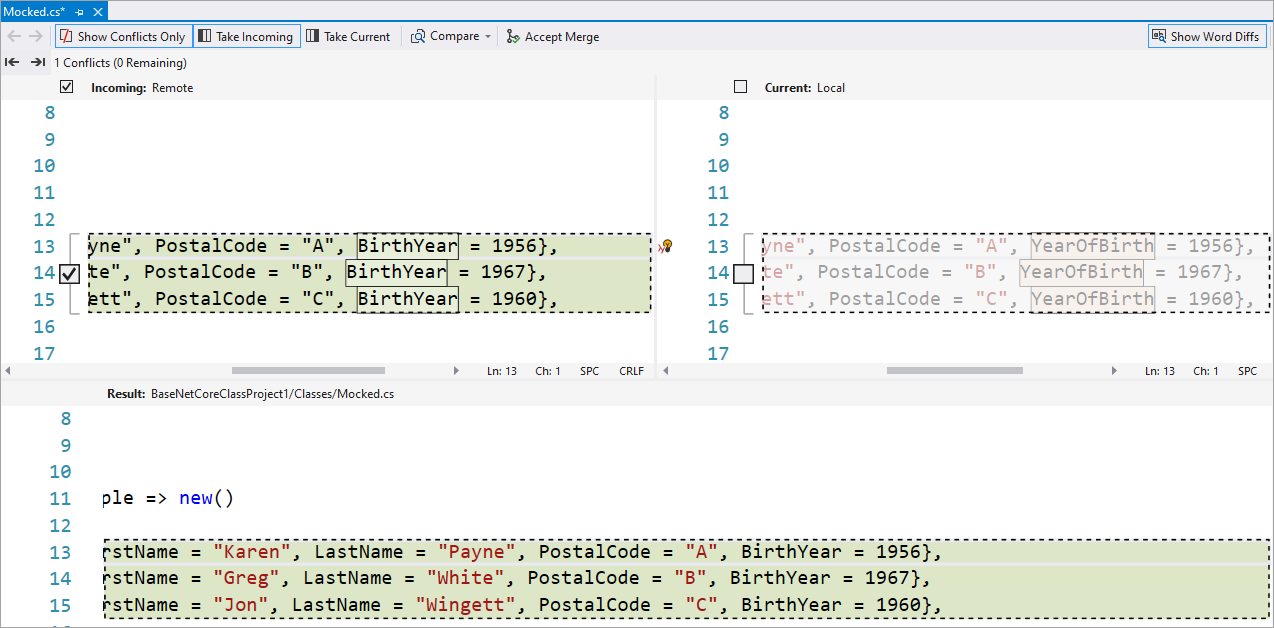
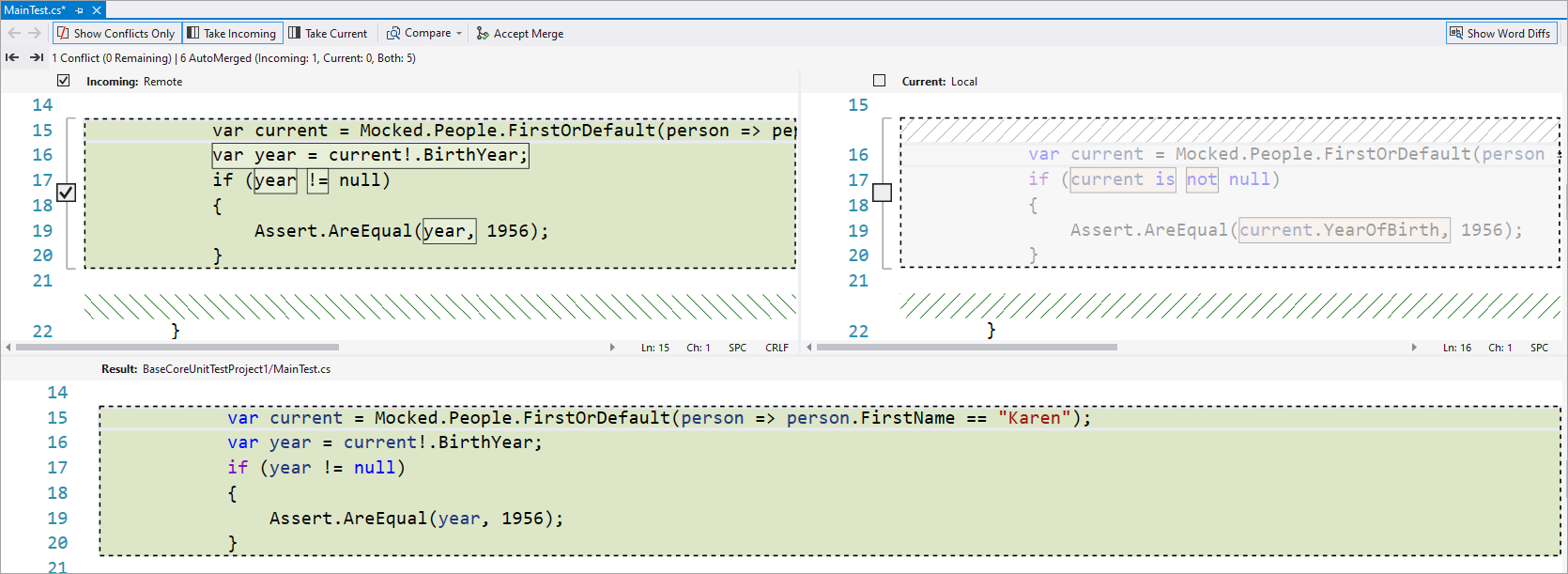
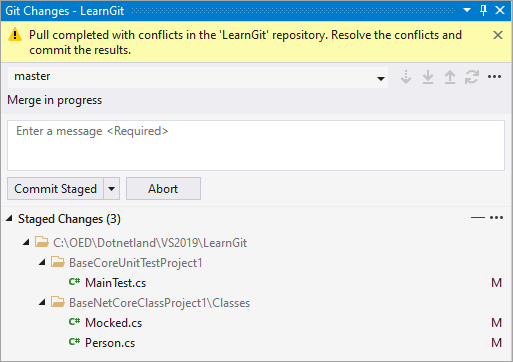
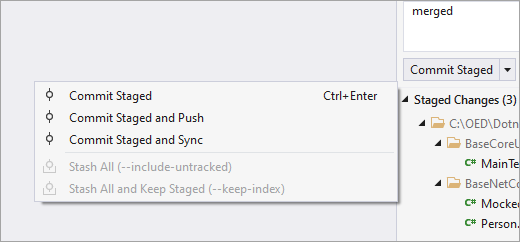
Okay, who would have postal code as an int so let’s go with string.

We must perform a merge on both Mocked and Person classes else we can’t compile because of type differences in the property PostalCode

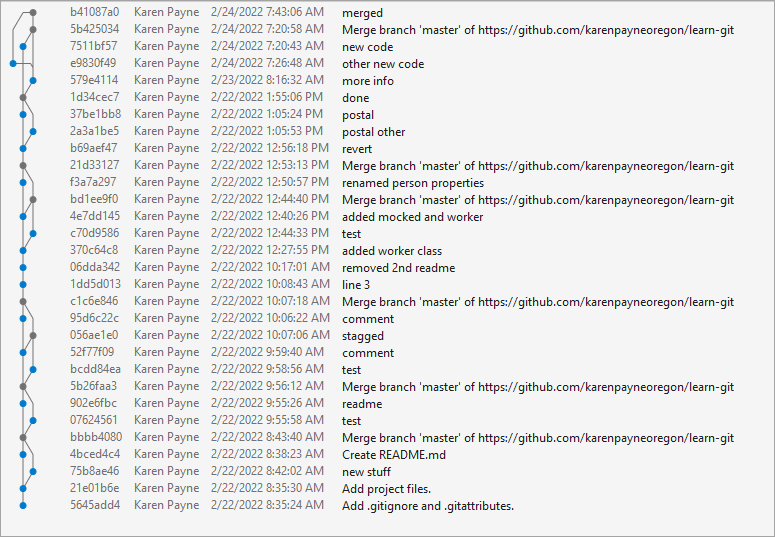




In both Visual Studio’s

1. Create a new solution folder, named Unit test projects
2. Add a new Unit test project  
   
3. Add a project reference (step 1)  
   
4. Add a project reference (step 2)  
   
5. Next we will modify the Person class in the first Visual Studio by adding a new property  
   public int YearOfBirth { get; set; }
6. In the second Visual Studio add the following property to the Person class  
   public int? BirthYear { get; set; }
7. In the first Visual Studio, Mocked class  
   
8. In the second Visual Studio, Mocked class  
   
9. Staying in the second Visual Studio, test project, create a test method below along with the following using statements  
   
10. Switch back to the first Visual Studio, create a test method below along with the following using statements  
    
11. In Git Changes tab, add a comment and perform a comment all/and push
12. Change to the other copy of Visual Studio
13. In Git Changes tab, do a fetch
14. Under the Git menu, view history
15. Note what is shown under Incoming.
16. View incoming details via right clicking on the item  
    
17. Although both copies of Visual Studio have similar changes we now need to decide what to do. Let’s do a Commit All and push, we get  
    
18. Now we see the following  
    
19. And at the top of the file  
    
20. Click on Open Merge Editor
21. We are going with the incoming, nullable birth year  
    
22. Next we need to resolve Mocked class, we accept incoming as the property values are the same but could take either or.  
    
23. Now we need to resolve code in the test method.
    1. Incoming is nullable int birth year while current is a not nullable. When dealing with nullable we can ask if the value is null or not via HasValue or in this case use the null forgiving operator **!**.
    2. For incoming we check if year is null while in current the birth year if not set is defaulted to 0.
    3. We need to accept incoming   
       
    4. We now need to commit  
         
         
       
    5. Select Commit Staged and Push.
    6. Change over to the other copy of Visual Studio, perform a fetch, view history, perform a pull.
    7. Both copies are sync’d

For kicks, here is my history while writing the above



In the above exercise all could had been avoided if both developers discussed code while instead they silo’d which is never good.

Lesson learned

* Communication is important
* You have seen an easy merge, they can get ugly or there is a need to reject either or code

https://github.com/karenpayneoregon/learn-git

**IBase interface**

public interface IBase

{

    public int Id { get; }

}

**Person class**

Replace BaseNetCoreClassProject1 with your project namespace

using BaseNetCoreClassProject1.Interfaces;

namespace BaseNetCoreClassProject1.Classes

{

    public class Person : IBase

    {

        public int Identifier { get; set; }

        public int Id => Identifier;

        public string FirstName { get; set; }

        public string LastName { get; set; }

        public override string ToString() => $"{Id,-5}{FirstName} {LastName}";

    }

}

**Worker version 1**

Replace BaseNetCoreClassProject1 with your project namespace

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using BaseNetCoreClassProject1.Interfaces;

namespace BaseNetCoreClassProject1.Classes

{

    public class Worker

    {

        public static void CompareValue<T>(List<T> sender) where T : class

        {

            foreach (var item in sender)

            {

                if (item is IBase data)

                {

                    Debug.WriteLine(data.Id);

                }

            }

        }

    }

}

**Worker version 2**

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using BaseNetCoreClassProject1.Interfaces;

namespace BaseNetCoreClassProject1.Classes

{

    public class Worker

    {

        public static void CompareValue<T>(List<T> sender) where T : class, IBase

        {

            foreach (var item in sender)

            {

                Debug.WriteLine(item.Id);

            }

        }

    }

}

**Mocked class 1**

Replace BaseNetCoreClassProject1 with your project namespace

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace BaseNetCoreClassProject1.Classes

{

    public class Mocked

    {

        public static List<Person> People => new()

        {

            new() { Identifier = 1, FirstName = "Karen", LastName = "Payne"},

            new() { Identifier = 2, FirstName = "Greg", LastName = "White"},

            new() { Identifier = 3, FirstName = "Jon", LastName = "Wingett"},

        };

        public static void Demo()

        {

            IEnumerable<Person> query = from p in People select p;

            Worker.CompareValue(People);

        }

    }

}

## Definitions

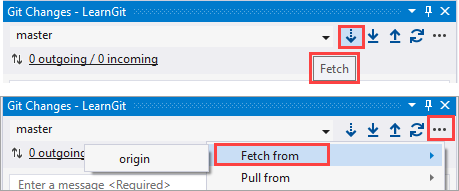
**Bob Hope brackets**

{ }

**Git Fetch**

The fetch command downloads commits, files, and refs from a remote repository into your local repo. Fetching is what you do when you want to see what everybody else has been working on. It’s similar to svn update in that it lets you see how the central history has progressed, but it doesn’t force you to actually merge the changes into your repository. Git isolates fetched content from existing local content; it has absolutely no effect on your local development work. Fetched content has to be explicitly checked out using the git checkout command. This makes fetching a safe way to review commits before integrating them with your local repository.

*It's important to fetch and pull before you push. Fetching checks if there are any remote commits that you should incorporate into your local changes. If you see any, pull first to prevent any upstream merge conflicts*.



**Head**

HEAD always points to the most recent commit which is reflected in the working tree. **Most git commands** which make changes to the working tree will start by changing HEAD.

**Push**

The git push command is used to upload local repository content to a remote repository. Pushing is how you transfer commits from your local repository to a remote repo. It's the counterpart to git fetch, but whereas fetching imports commits to local branches, pushing exports commits to remote branches.

**Pull**

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match that content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration work flows. The git pull command is actually a combination of two other commands, git fetch followed by git merge. In the first stage of operation git pull will execute a git fetch scoped to the local branch that HEAD is pointed at. Once the content is downloaded, git pull will enter a merge workflow. A new merge commit will be-created and HEAD updated to point at the new commit.

**Staging**

* Staging helps you split up one large change into multiple commits - Let's say you worked on a large-ish change, involving a lot of files and quite a few different subtasks. You didn't actually commit any of these -- you were "in the zone", as they say, and you didn't want to think about splitting up the commits the right way just then. (And you're smart enough not to make the whole thing on honking big commit!). Now the change is all tested and working, you need to commit all this properly, in several clean commits each focused on one aspect of the code changes. With the index, just stage each set of changes and commit until no more changes are pending. Really works well with git gui if you're into that too, or you can use git add -p or, with newer gits, git add -e.
* Staging helps in reviewing changes - Staging helps you "check off" individual changes as you review a complex commit, and to concentrate on the stuff that has not yet passed your review. Let me explain. Before you commit, you'll probably review the whole change by using git diff. If you stage each change as you review it, you'll find that you can concentrate better on the changes that are not yet staged.
* Staging helps when a merge has conflicts - When a merge happens, changes that merge cleanly are updated both in the staging area as well as in your work tree. Only changes that did not merge cleanly (i.e., caused a conflict) will show up when you do a git diff. Again, this lets you concentrate on the stuff that needs your attention -- the merge conflicts.
* Staging helps you sneak in small changes - Let's say you're in the middle of a somewhat large-ish change and you are told about a very important bug that needs to be fixed ASAP. The usual recommendation is to do this on a separate branch, but let's say this fix is really just a line or two, and can be tested just as easily without affecting your current work.

**Feature**

The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the main branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the main branch will never contain broken code, which is a huge advantage for continuous integration environments.

# Resources

* [git - the simple guide](http://up1.github.io/git-guide/index.html)
* [git syncing](https://www.atlassian.com/git/tutorials/syncing)
* [How Git Works](https://app.pluralsight.com/library/courses/how-git-works/table-of-contents) – two hour course
* [Learn Git Branching](https://learngitbranching.js.org/?locale=en_US)
* [Pro Git book](http://book.git-scm.com/book/en/v2) (online/free)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Operation |  | Git | SVN | X faster |
| Commit Files (A) | Add, commit and push 113 modified files (2164+, 2259-) | 0.64 | 2.60 | 4x |
| Commit Images (B) | Add, commit and push a thousand 1 kB images | 1.53 | 24.70 | 16x |
| Diff Current | Diff 187 changed files (1664+, 4859-) against last commit | 0.25 | 1.09 | 4x |
| Diff Recent | Diff against 4 commits back (269 changed/3609+,6898-) | 0.25 | 3.99 | 16x |
| Diff Tags | Diff two tags against each other (v1.9.1.0/v1.9.3.0) | 1.17 | 83.57 | 71x |
| Log (50) | Log of the last 50 commits (19 kB of output) | 0.01 | 0.38 | 31x |
| Log (All) | Log of all commits (26,056 commits – 9.4 MB of output) | 0.52 | 169.20 | 325x |
| Log (File) | Log of the history of a single file (array.c – 483 revs) | 0.60 | 82.84 | 138x |
| Update | Pull of Commit A scenario (113 files changed, 2164+, 2259-) | 0.90 | 2.82 | 3x |
| Blame | Line annotation of a single file (array.c) | 1.91 | 3.04 | 1x |