Exercise – Git Conflict Resolution

Prerequisite: Git Branching

Before you begin this tutorial, ensure you have already created a repository containing files that you can work with.

You should also be familiar with the concept of creating branches in a Git repository and integrating work between them by merging them together.

What are "merge conflicts"?

Merge conflicts occur when Git tries to merge work between two branches that have made changes to the same part of a file since the last time they were merged.

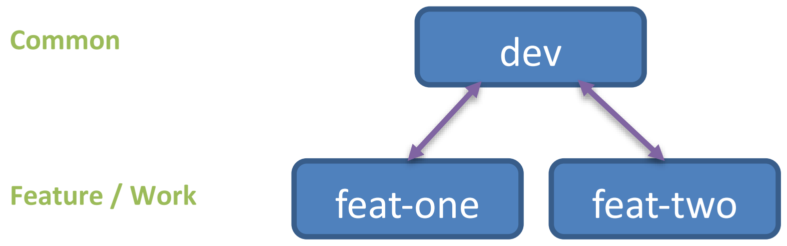
This is normal and can occur easily when the team is working, such as…

- Two programmers need to change the way health works to add buffs and powerups
- A programmer adds a new object to a Unity scene while a designer is changing values
- Artists need to update textures while a technical artist is adding alpha channels to them

In this exercise, we'll focus on how to resolve merge conflicts between two programmers.

Creating a Merge Conflict

To begin, create two branches called “**feat-one**” and “**feat-two**” off a common branch, like **develop**:



Checkout the first branch and make some changes to a file in your project. Make sure to add and commit those changes. Repeat that process with the second branch, making some different changes to the same file in that project. Once again, add and commit those changes.

For example, here’s two conflicting changes that could be made to a “Hello World” project:

dev branch (the original)

```cpp
#include <iostream>

int main()
{
	std::cout << "Welcome to the Australian Science Center." << std::endl;
	std::cout << "We hope you've had a good time." << std::endl;
	std::cout << "For your safety, save before exiting.." << std::endl;

	std::cout << "Copyright (c) ASC" << std::endl;

	return 0;
}
```

feat-one branch – Reminds users to save first

```cpp
#include <iostream>

int main()
{
	std::cout << "Welcome to the Australian Science Center." << std::endl;
	std::cout << "We hope you've had a good time." << std::endl;
	std::cout << "Please make sure to save before committing." << std::endl;

	std::cout << "Copyright (c) ASC" << std::endl;

	return 0;
}
```

feat-two branch – Reminds users to donate if helped

```cpp
#include <iostream>

int main()
{
	std::cout << "Welcome to the Australian Science Center." << std::endl;
	std::cout << "We hope you've had a good time." << std::endl;
	std::cout << "Don't forget to donate if this helped." << std::endl;

	std::cout << "Copyright (c) ASC" << std::endl;

	return 0;
}
```

Once all branches have been created, begin the process by merging **feat-one** into **develop**. This should proceed without issues, because **develop** is just behind and has no new work to combine in addition to the work being merged in. Once merged, push both branches you modified.

Then switch to the **feat-two** branch and merge **develop** into **feat-two**.

❓ **Why are we merging 'develop' into 'feat-two' and not vice-versa?**
We can't merge up into develop yet because **feat-two** is now out of date and missing commits that present in **develop**. To introduce those changes into **feat-two**, we have to merge in **develop** before we can merge back into **develop**.

This should cause a merge conflict since **develop** contains new commits that modify files that we've modified in **feat-two**.

Reading Conflict Markers

When a conflict occurs, Git leaves behind conflict markers that indicate the state of the code as expressed by **our** version (the current branch) and **their** version (the branch we are merging changes from).

If we use the example from above, Git will generate a file that looks like this:

```
#include <iostream>

int main()
{
	std::cout << "Welcome to the Australian Science Center." << std::endl;
	std::cout << "We hope you've had a good time." << std::endl;
<<<<<<< ours
	std::cout << "Don't forget to donate if this helped." << std::endl;
=======
	std::cout << "Please make sure to save before committing." << std::endl;
>>>>>>> theirs

	std::cout << "Copyright (c) ASC" << std::endl;

	return 0;
}
```

*In this example, **ours** refers to the **feat-two** branch while **theirs** refers to the **develop** branch.*

The person that is merging must review each set of conflict markers and edit the file, leaving behind only valid code. The code will likely not compile if these conflict markers are left in the file.

In the scenario presented above, the merged code could include both passages of text:

```
#include <iostream>

int main()
{
	std::cout << "Welcome to the Australian Science Center." << std::endl;
	std::cout << "We hope you've had a good time." << std::endl;
	std::cout << "Don't forget to donate if this helped." << std::endl;
	std::cout << "Please make sure to save before committing." << std::endl;

	std::cout << "Copyright (c) ASC" << std::endl;

	return 0;
}
```

In this example, we chose to resolve the conflict by keeping changes from both branches.

Once you finish resolving conflicts in a single file, you must **add** it, making it staged and thus considered resolved. After all conflicts are resolved, you can finish the merge by making the merge commit, which contains all of the work done to fix the conflicts.

Resolving Conflicts

When you merged **develop** into **feat-two**, you should have created merge conflicts due to the conflicting changes to the same file.

Review those conflicts and resolve all each one before committing and finishing the merge. Make sure to push after resolving those conflicts to update the version of **develop** and **feat-two** that are available on GitHub.