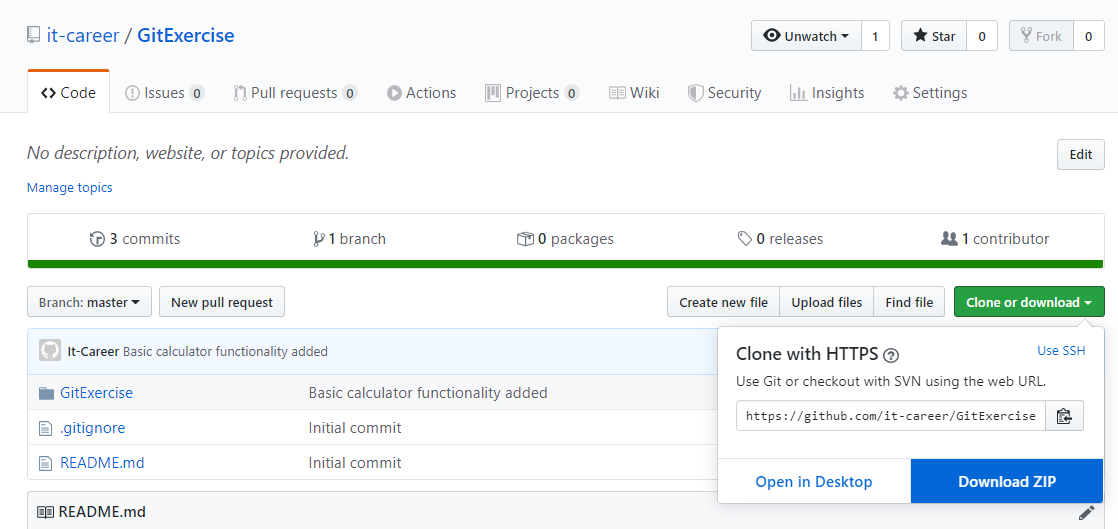
# Упражнение: Git в практиката

## Какво ще постигнем

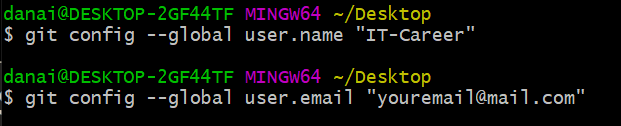
Следвайки упражнението, ще упражните основните Git команди, които всеки разработчик на софтуер използва ежедневно. За целта се разделете на екипи от по трима души. Всеки от екипа ще има собствена задача, която ще изисква от него да изпозлва научените Git команди. За улеснение ще наречем хората в екипа Разработчик А, Разработчик Б и Разработчик В. За упражнението ще използваме малък и много опростен проект – конзолен калкулатор. По този начин ще се фокусираме върху самите Git команди, а не върху имплементацията на сложни алгоритми.

## Хост на репозитори

Проектът е хостнат в GitHub и може да бъде открит на адрес <https://github.com/it-career/GitExercise>. За да започнете работата си по него, нека вътрешно избран член на екипа да го изтегли на машината си.



След като проектът е изтеглен, трябва да се създаде локално Git репозитори, като преди това е добре да се конфигурират личните данни за потребителят (тази стъпка е за всички членове на екипа).



След като всеки потребител конфигурира личните си данни, е време да пристъпим към създаването на локално репозитори.

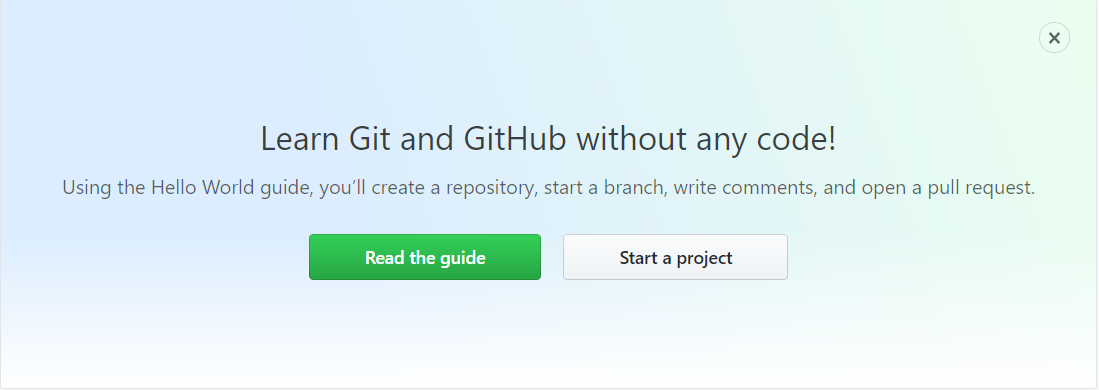
Тази стъпка трябва да бъде изпълнена от члена на екипа, който е изтеглил проекта. Първо, използвйки конзолата, трябва да отидем в директорията на изтегления проект (използвайте командата cd <directory\_path>, а за да сте сигурни, че сте на правилното място можете да проверите съдържанието на папката с командата ls).



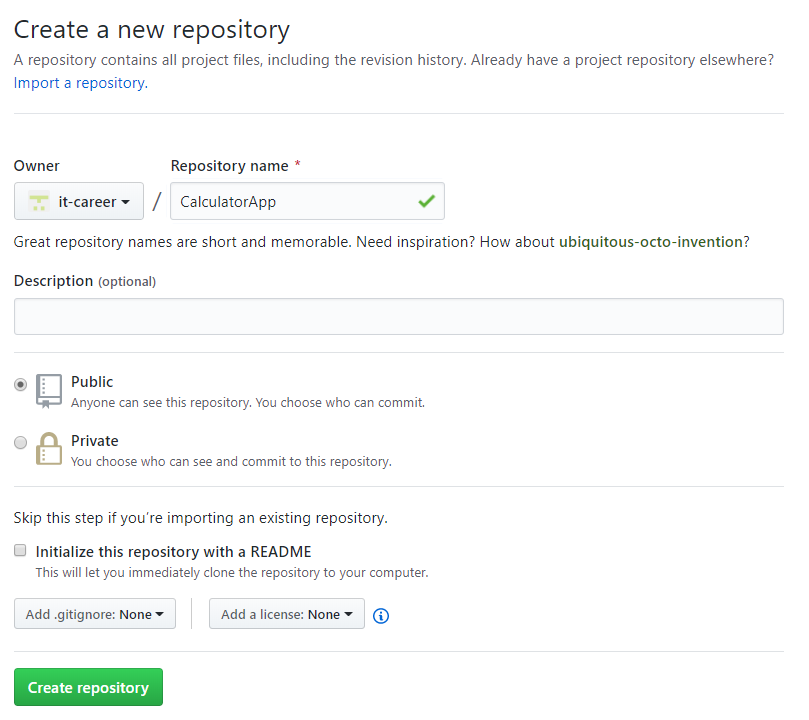
След като сме сигурни, че сме в правилната директория е време да инициализираме локално репозитори за нашия проект.

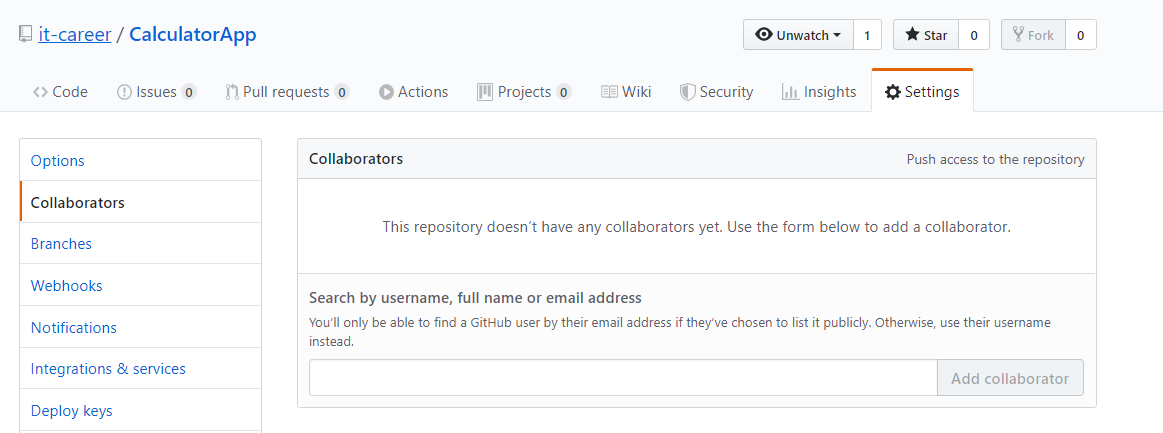


Време е да хостнем репозиторито си в GitHub. От <https://github.com/>, след като влезете в акаунта си, старитрайте нов проект.

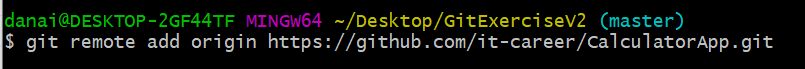


Дайте му подходящо име и изберете дали да е публичен или частен. В този случай ние ще създадем публичен проект. След създаването на проекта трябва да се добавят другите членове на екипа към проекта, за да могат да работят по него директно в GitHub хостнатото репозитори. За целта отидете на Options -> Collaborators и въведете потребителското име на вашите колеги.

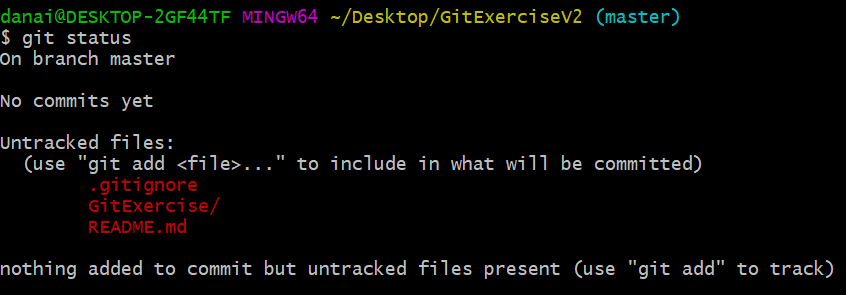




След като сме изпълнили и тези стъпки е време да хостнем локалния си проект на току-що създаденото от нас GitHub репозитори.

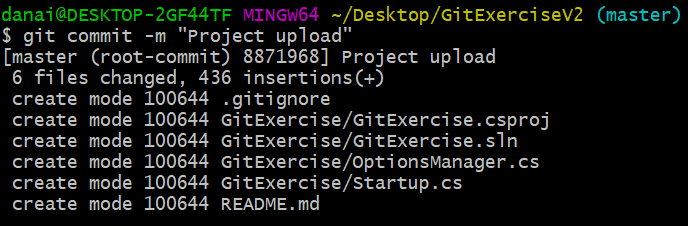


Нека проверим текущия статус на нашето репозитори.

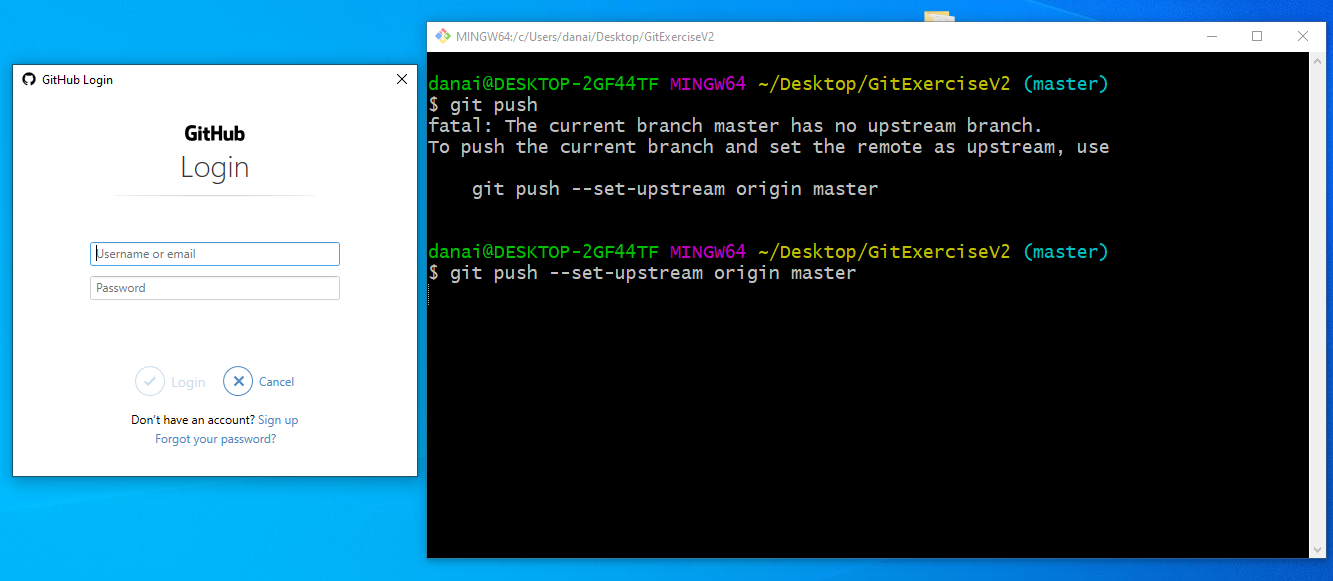


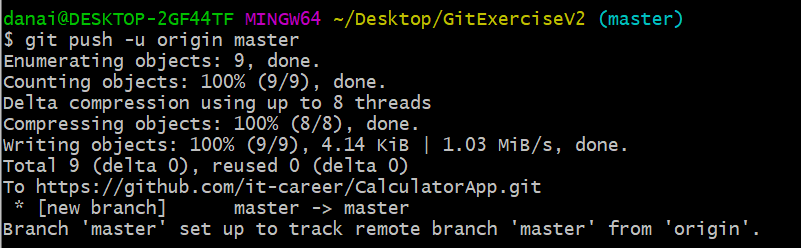
След като се уверим, че всичко изглежда наред е време да добавим промените и да ги commit-нем към GitHub, за да качим началния вид на проекта. Ще използваме командата git add . , за да добавим за следене файловете, намиращи се в нашата директория, след което ще ги пакетираме в commit, за който ще изберем подходящо съобщение.





Възможно е, при опит да изпратим commit-а към сървъра (git push) да получим грешка, защото за текущия клон няма зададено указание, къде да изпраща промените. Git е достатъчно умен и ще ни даде и решението на този проблем. След тази настройка ще ни бъдат поискани данните за нашия GitHub акаунт. Това се случва еднократно, след което тези данни се запазват и няма да ни бъдат поискани пак. Това е начинът, по който GitHub определя дали даден потребител има права да прави промени по даден проект. Инструкции как може да стане това има и на страницата на новосъздаденото от вас GitHub репозитори.

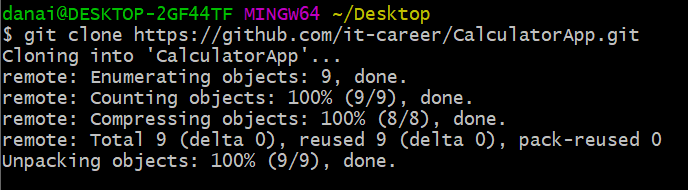




На този етап проектът е хостнат в GitHub и всеки член на екипа може да го свали, за да започне работа по него. Вътрешно в екипа изберете кой ще заеме ролята на разработчик А, Б и В.

## Клониране

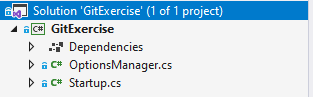
Всеки член на екипа трябва да има проекта в текущата си версия на своята машина. За да стане това, нека членовете на екипа го клонират, използвайки командата git clone <URL> - с HTTPS адреса на репозиторито, в избрана от потребителя директория.



След като всеки има проекта на своята машина, е време да се запознаем с проекта, който сме клонирали.

## Проектът

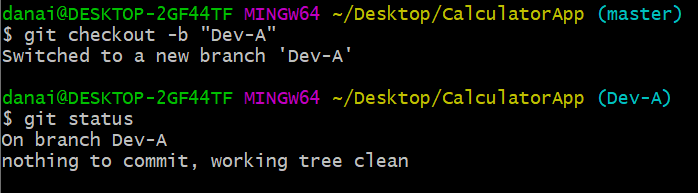
Отворете проекта чрез Visual Studio. От Solution Explorer-а може да забележите, че проекта съдържа само един слой – конзолно приложение с класове в него, като единият съдържа Main метода, а другият е статичен клас, който изпълнява помощна функция. Нека всеки отдели време да се запознае с проекта, за да преминем към следващата част, в която всеки разработчик изпълнява възложената си задача.



От тук нататък упражнението се разделя в три посоки, като всеки разработчик може спокойно да премине директно към частта, която се отнася за него.

## Разработчик А

Добра практика е всяка задача да се изпълнява на отделен клон, който по-късно съответно се одобрява и събира със основния клон. Затова ще направим нов клон с име “Dev-A” и директно ще преминем на него. За да проверим дали всичко е наред, можем да използваме командата git status.

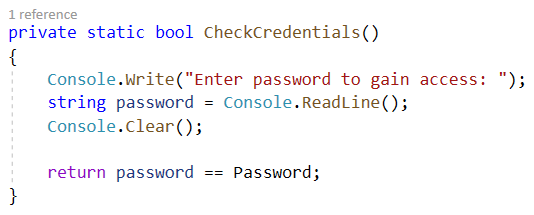


**Задачите**, които Разработчик А получава са следните:

* Да се добави нова опция за нашия калкулатор – деление на числа
* Да се добави нова опция за нашия калкулатор – изваждане с абсолютна стойност (на резултата от изваждането)
* Да се създаде метод, който проверява дали потребителят може да получи достъп към приложението. Достъп се дава при правилно въведена парола, а в противен случай изпълнението на приложението спира. Достатъчно е паролата да се пази като константа в основния ни клас – Startup. За парола задайте “abcd1234”. Преди да се затвори приложението при въведена неправилна парола да се изпише подходящо съобщение.

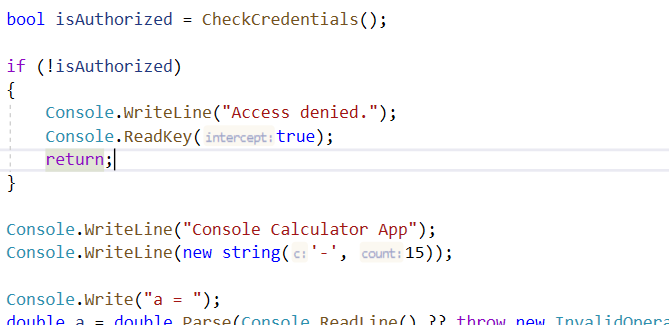
**Примерна имплементация:**

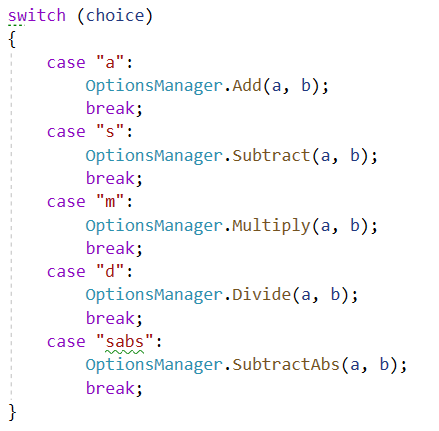
В класа Startup:



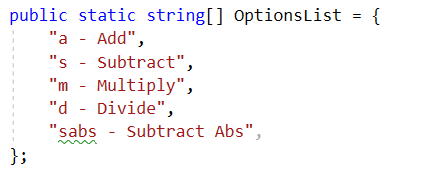


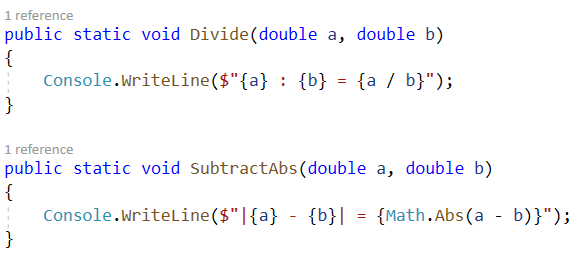
В Main метода:





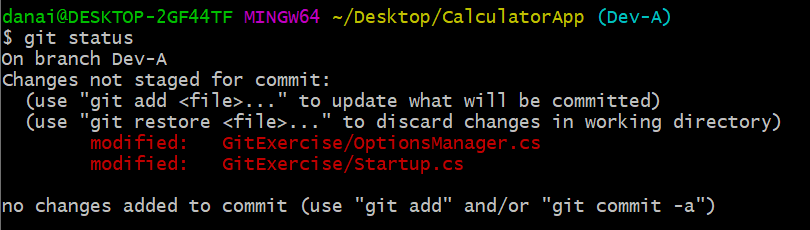
В класа OptionsManager:



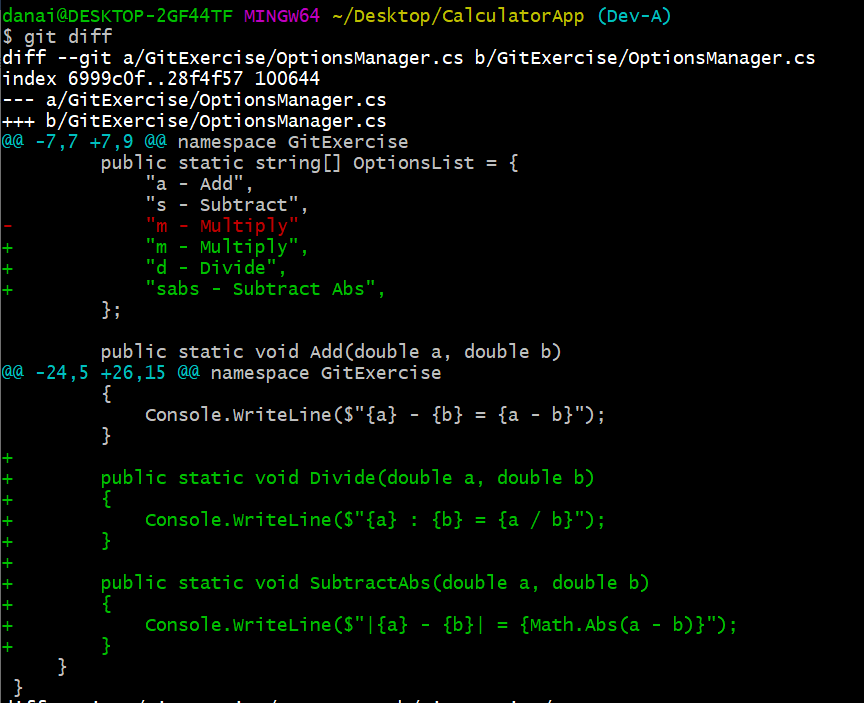


След изпълнението на задачите е време промените да бъдат отразени и в GitHub репозиторито. Това ще се случи в новосъздадения клон “Dev-A” и няма да се отрази по никакъв начин на проекта в master клона, който е и основен.

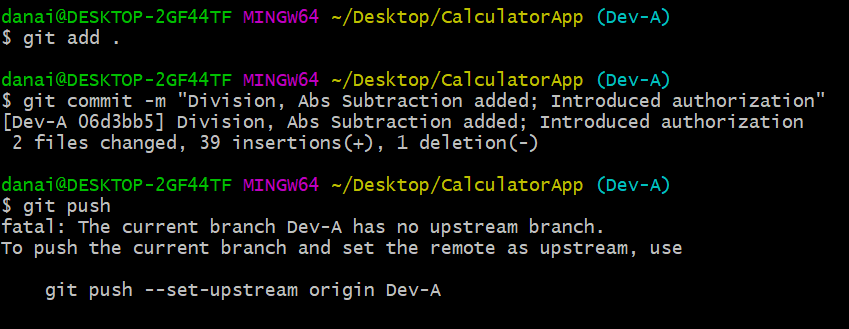
На първо място трябва да добавим промените в локалното ни репозитори с командата git add ., след което ще създадем и изпратим commit. Преди да направим това, можем да проверим текущия статус на локалното ни репозитори с командата git status.

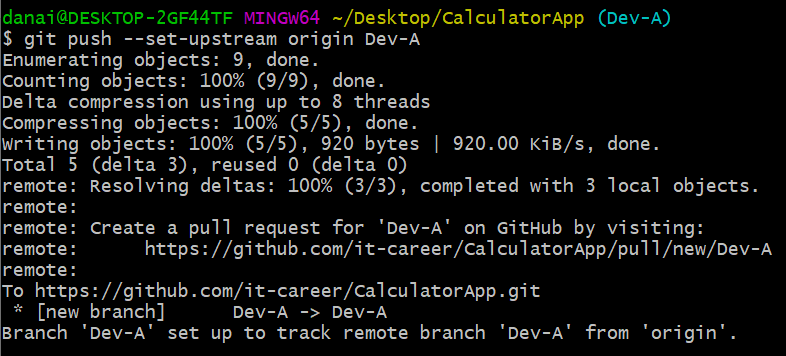


Командата връща полезна информация отностно състоянието на файловете ни. По файловете OptionsManager.cs и Startup.cs са направени промени, които не са добавени за commit. Друга полезна команда е git diff, която извежда подробна информация за всяка извършена промяна.



След като разгледахме резултата от тези опционални команди е време да се върнем към нашата работа. Ще добавим всички промени и ще създадем локален commit, като зададем подохдящо съобщение. След това ще изпратим локалния commit към сървъра.

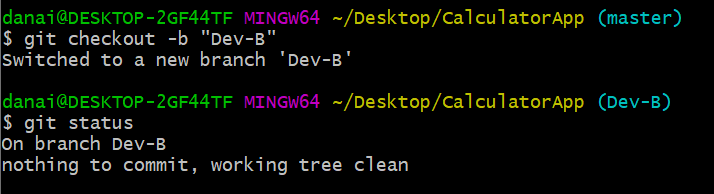




С това за сега приключва работата на Разработчик А.

## Разработчик Б

Добра практика е всяка задача да се изпълнява на отделен клон, който по-късно съответно се одобрява и събира със основния клон. Затова ще направим нов клон с име “Dev-B” и директно ще преминем на него. За да проверим дали всичко е наред, можем да използваме командата git status.

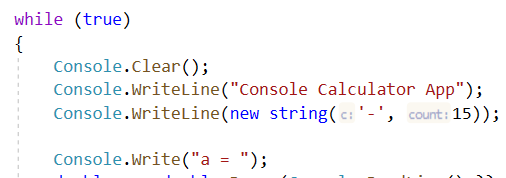


**Задачите**, които Разработчик Б получава, са следните:

* Да се добави нова опция за нашия калкулатор – остатъчно деление (деление, което връща като резултат остатъка)
* Повтаряемост на пресмятанията. Забележете, че след изпълнението на едно пресмятане нашият калкулатор прекъсва изпълнението си. Да се промени така логиката, че потребителят да бъде питан за следващо пресмятане, докато не избере опция за изход.

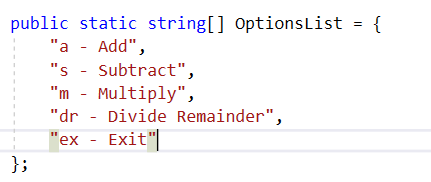
**Примерна имплементация:**

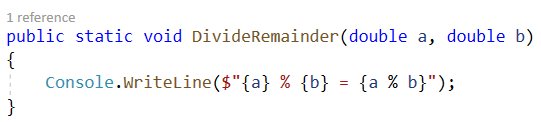
В Startup класа – Main():





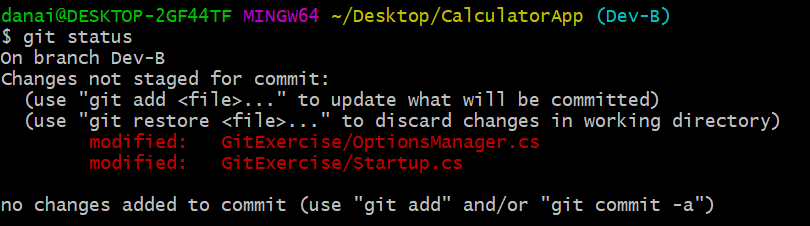
В класа OptionsManager:





След изпълнението на задачите е време промените да бъдат отразени и в GitHub репозиторито. Това ще се случи в новосъздадения клон “Dev-B” и няма да се отрази по никакъв начин на проекта в master клона, който е и основен.

На първо място трябва да добавим промените в локалното ни репозитори с командата git add ., след което ще създадем и изпратим commit. Преди да направим това, можем да проверим текущия статус на локалното ни репозитори с командата git status.

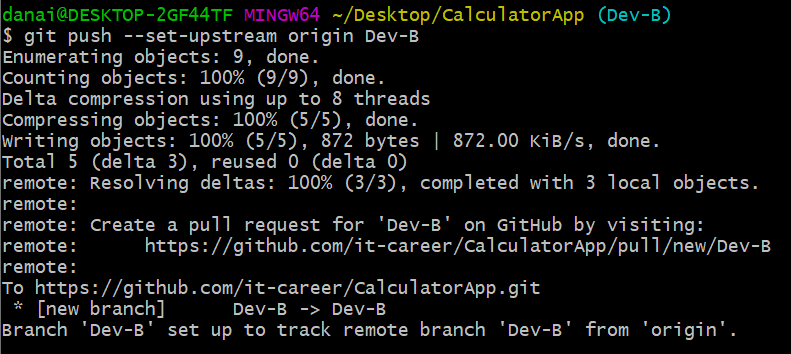


Командата връща полезна информация отностно състоянието на файловете ни. По файловете OptionsManager.cs и Startup.cs са направени промени, които не са добавени за commit. Друга полезна команда е git diff, която извежда подробна информация за всяка извършена промяна.



След като разгледахме резултата от тези опционални команди е време да се върнем към нашата работа. Ще добавим всички промени и ще създадем локален commit, като зададем подохдящо съобщение. След това ще изпратим локалния commit към сървъра.

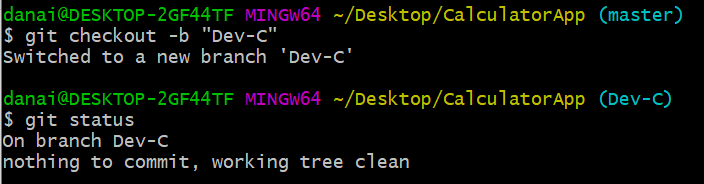




С това за сега приключва работата на Разработчик Б.

## Разработчик В

Добра практика е всяка задача да се изпълнява на отделен клон, който по-късно съответно се одобрява и събира със основния клон. Затова ще направим нов клон с име “Dev-C” и директно ще преминем на него. За да проверим дали всичко е наред, можем да използваме командата git status.

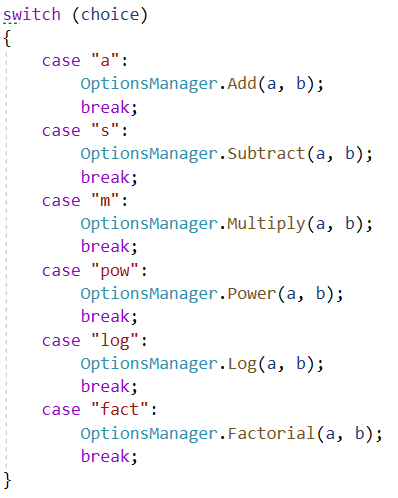


**Задачите**, които Разработчик В получава, са следните:

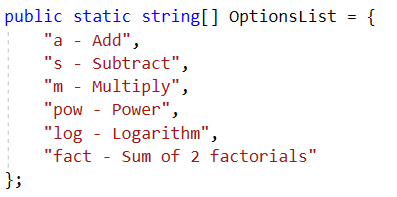
* Да се добави нова опция за нашия калкулатор – степенуване (изписва резултата от повдигането a на степен b)
* Да се добави нова опция за нашия калкулатор – логаритмуване (изписва резултата от логаритъм a с база b)
* Да се добави нова опция за нашия калкулатор – сумата на 2 факториела (изписва резултата от a! + b!)

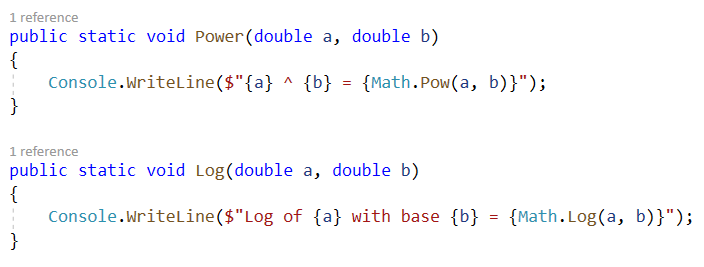
**Примерна имплементация:**

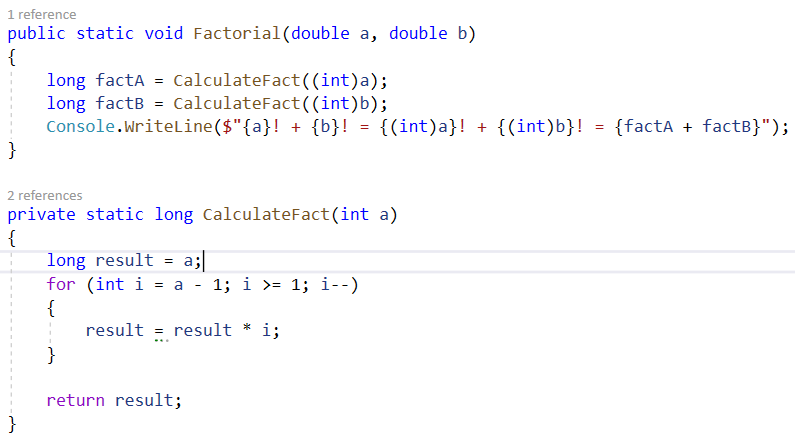
В Startup класа – Main():



В класът OptionsManager:

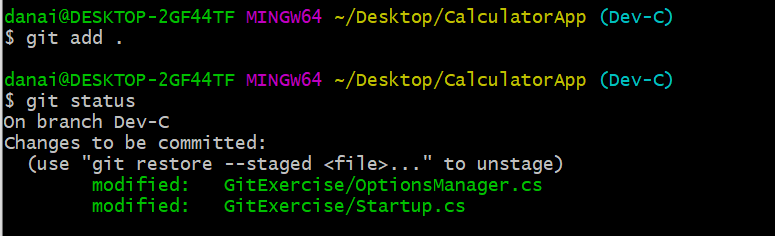






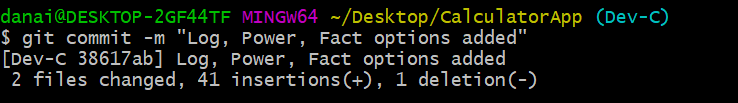
След изпълнението на задачите е време промените да бъдат отразени и в GitHub репозиторито. Това ще се случи в новосъздадения клон “Dev-C” и няма да се отрази по никакъв начин на проекта в master клона, който е и основен.

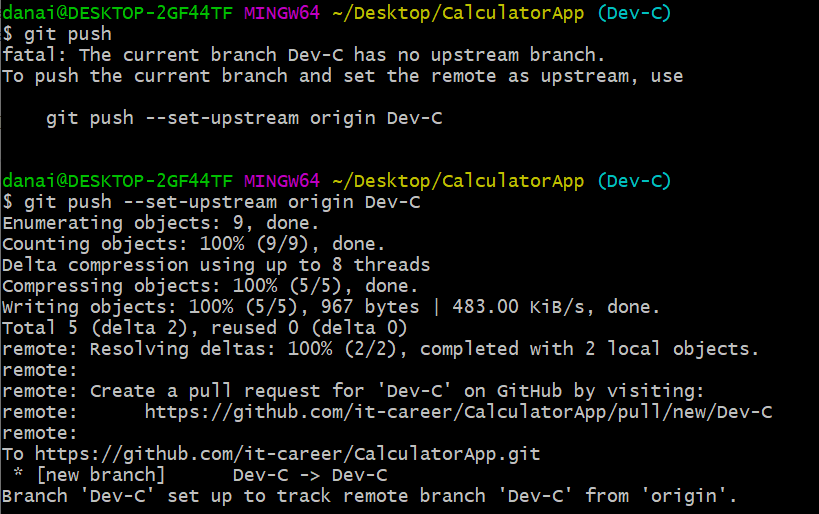
На първо място трябва да добавим промените в локалното ни репозитори с командата git add ., след което ще създадем и изпратим commit. Преди да направим това (или след това), можем да проверим текущия статус на локалното ни репозитори с командата git status.



Командата връща полезна информация отностно състоянието на файловете ни. По файловете OptionsManager.cs и Startup.cs са направени промени, които не са добавени за commit. Друга полезна команда е git diff, която извежда подробна информация за всяка извършена промяна.

След като разгледахме резултата от тези опционални команди е време да се върнем към нашата работа. Ще добавим всички промени и ще създадем локален commit, като зададем подохдящо съобщение. След това ще изпратим локалния commit към сървъра.



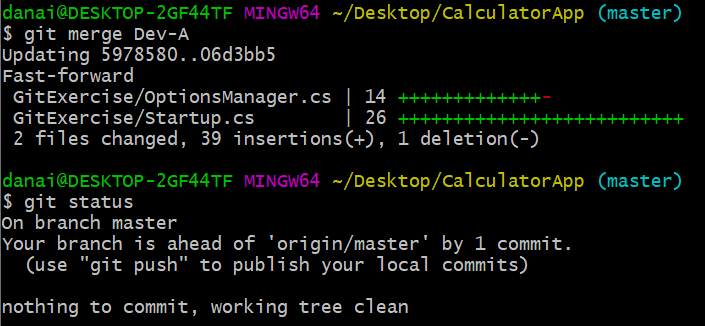


С това за сега приключва работата на Разработчик В.

## Сливане на клоновете

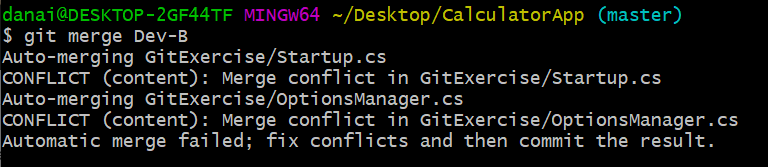
Дойде етапът, в който трябва да слеем клоновете на проекта си, за да получим цялата функционалност в главния клон. Нека слеем клоновете подред. Да кажем, че се намираме в следния сценарий: аз съм отговорен да одобрявам или не даден код в проекта. Разработчик А, Б и В са свършили със задачите си и са ми съобщили за това. Сега искам да слея клоновете с техните задачи в главния клон.

Да започнем с клон Dev-A (намирайки се в master):



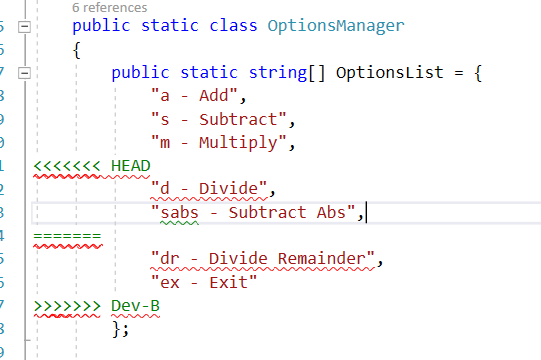
Резултатът е безпроблемно сливане, защото поначало клон Dev-A тръгва от копие на клон master и само го надгражда. За да завършим сливането можем да commit-нем промените, но ще оставим това за по-късно.

Нека слеем и клон Dev-B с master (в който вече сме слели Dev-A):

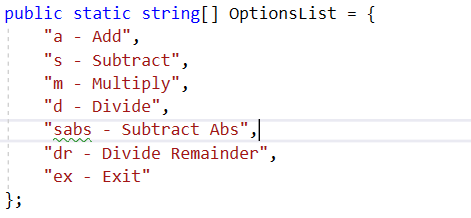


Резултатът е неуспешно автоматично сливане на двата клона – появяват се конфликти. Това е така, защото Dev-B надгражда копие на master, но master клонът се е изменил преждевременно (заради сливането му с Dev-A). Има няколко начина да се разрешат тези конфликти – ние ще го направим през Visual Studio. Като резултат от командата виждаме кои файлове не са успели да се слеят. Нека ги отворим.

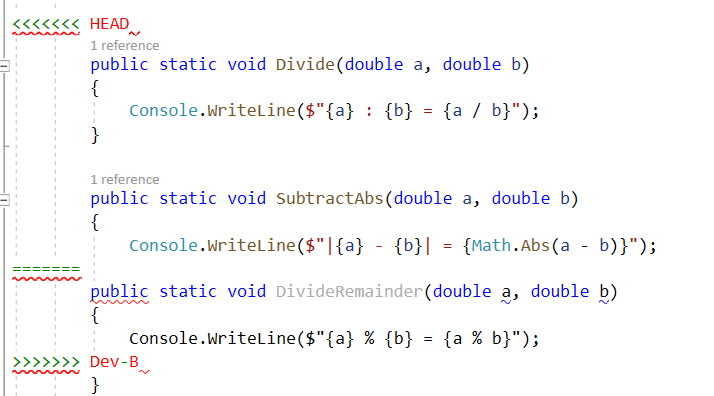
Ако отворим файла OptionsManager.cs, ще открием, че Git е маркирал в коя част от файла са възникнали проблемите. Също е и отбелязано коя версия на кода от кой клон идва:



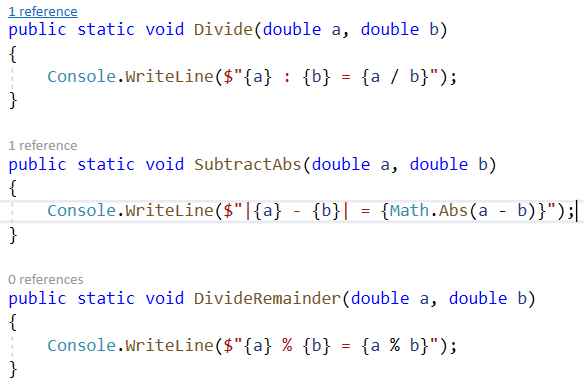
Лесно ще решим този проблем, като оставим и двете добавки към кода:



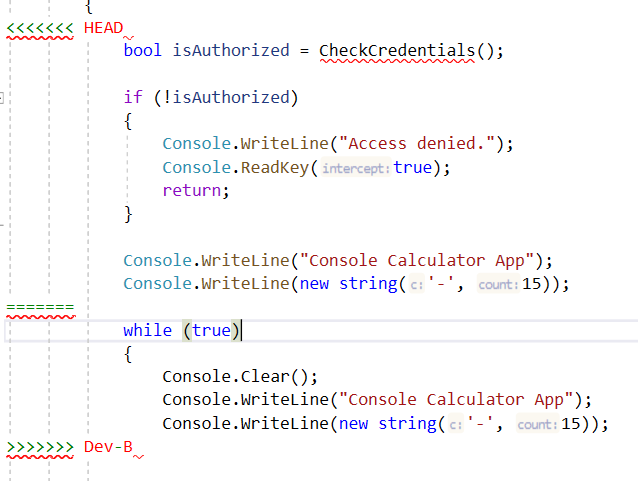
Друг проблем, който е възникнал в този файл е при новосъздадените методи (Git не може сам да реши кои методи да останат – Divide и SubtractAbs или DivideRemainder):



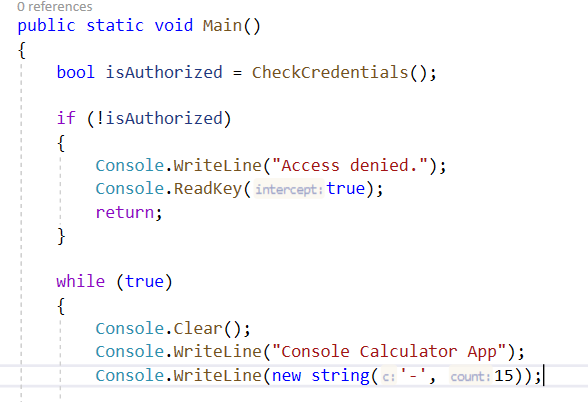
Аналогично ще оставим имплементацията на всички нови методи от двата клона, защото всички са ни нужни:



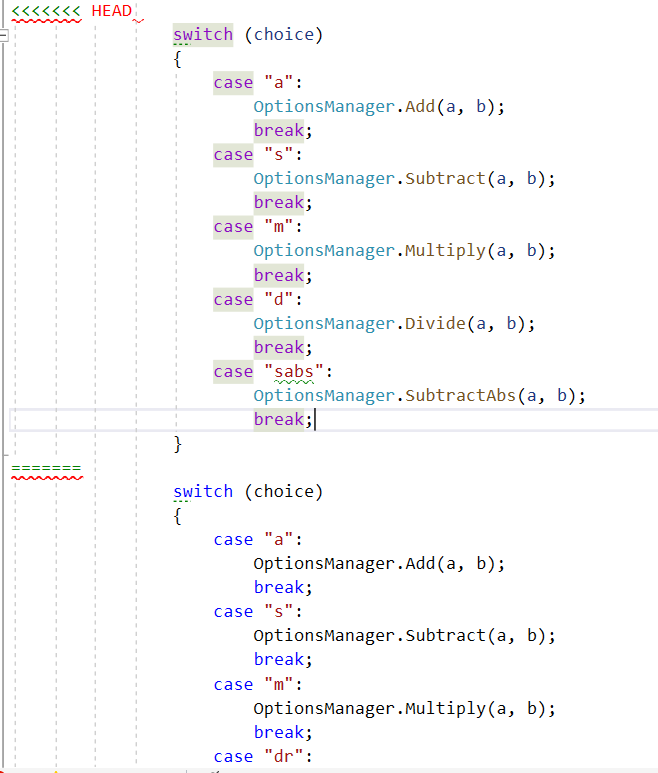
Да преминем към другия проблемен файл – Startup.cs. Първият конфликт тук е при въвеждането на повтаряемостта на събитията, както и проверката за достъп до приложението:



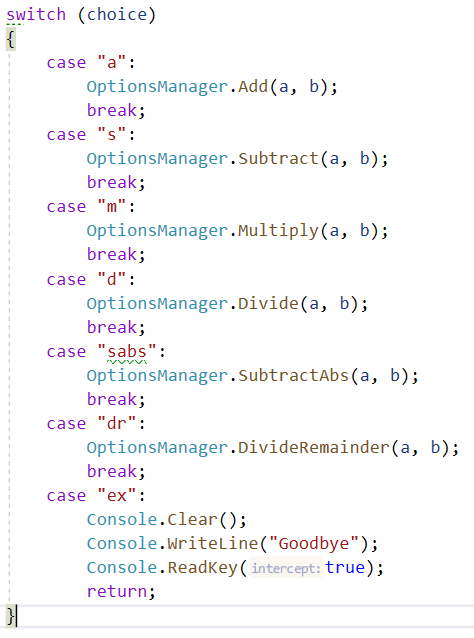
За да го решим, просто ще оставим както цикъла, така и проверката в началото, а повтарящият се код ще изтрием:



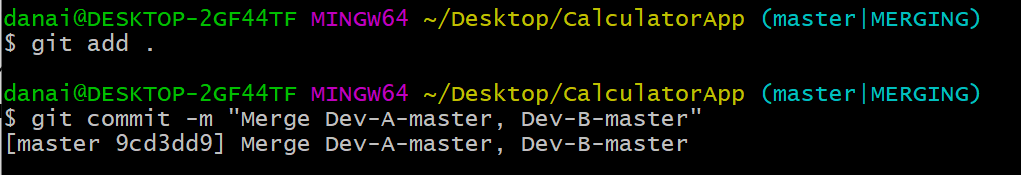
Аналогично ще оставим всички нови методи в switch-case:



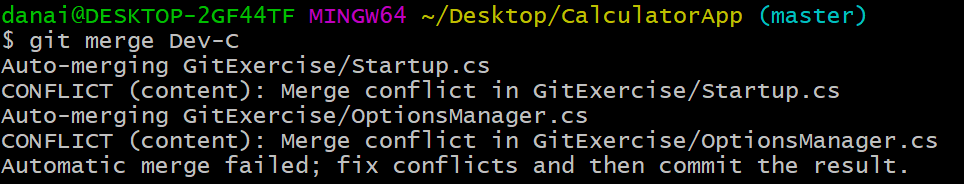
, за да получим:



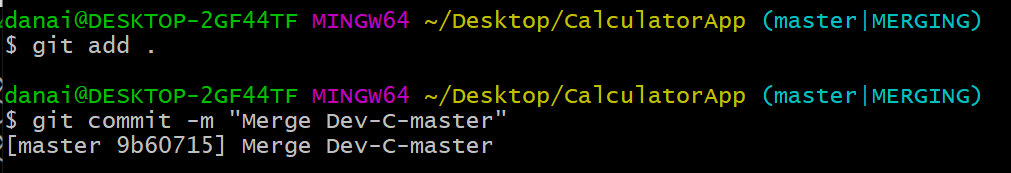
На този етап сме разрешили конфликтите, възникнали от сливането на двата клона. За да ги добавим трябва да ги commit-нем.



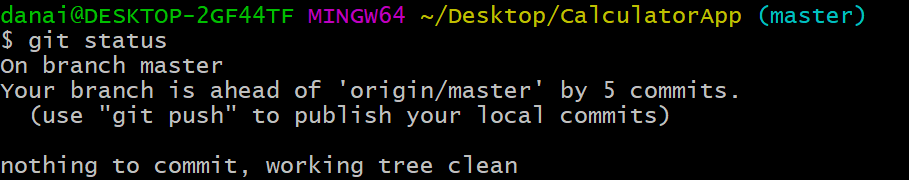
Време е да слеем и последния клон Dev-C с главния:



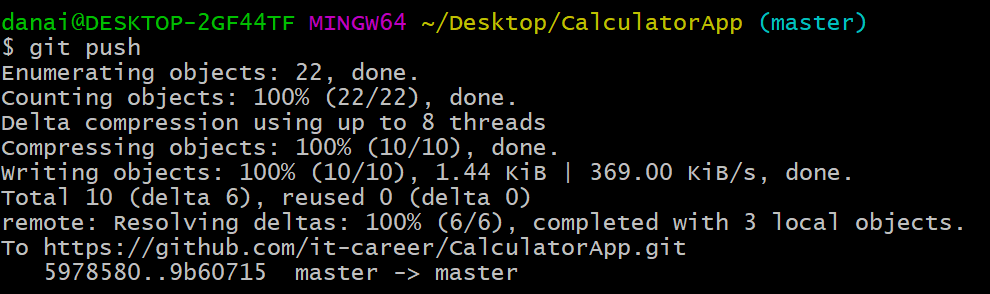
Отново автоматичното сливане е невъзможно. По аналогичен път разрешете възникналите конфликти в кода. След това ще довършим сливането:



Ако на този етап проверим статуса на проекта ще видим дали всичко е наред със сливанията на нашите клонове:



Ако всичко е наред и получим информация колко commit-а назад е GitHub репозиторито ни в сравнение с локалното, всичко което остава е да изравним двете версии като качим промените.



С това приключва нашата работа по този проект. Към този момент в GitHub репозиторито ни разполагаме с проект, съдържащ цялата функционалност, която имплементирахме. За да тествате дали и в локалното репозитори разполагате с пълната функционалност просто пуснете проекта и вижте дали всеки метод работи, както се очаква.