

دانشگاه صنعتی امیرکبیر
(پلی تکنیک تهران)

دانشکده ریاضی و علوم کامپیووتر

برنامه‌سازی پیشرفته و کارگاه

Git

استاد درس

دکتر مهدی قطعی

استاد دوم

بهنام یوسفی مهر

نگارش

سیدآرمان حسینی، سانیا عزتی، صالح ملازاده و محمدرضا شیخ‌الاسلامی

بهار ۱۴۰۳

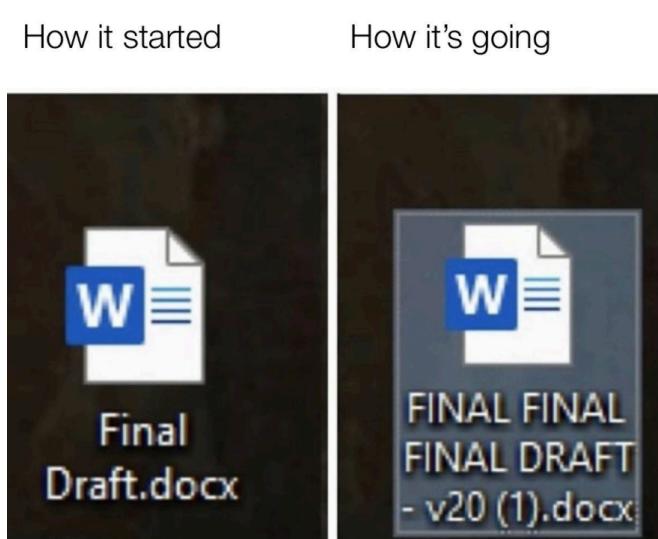
فهرست

4	مقدمه
6	اطمینان از نصب صحیح گیت
7	آشنایی اولیه با shell
7.....	ویندوز
8.....	لینوکس و macOS
11	کردن git configure
12.....	ایجاد ریپوی گیت
15.....	کردن تغییرات در گیت stage
18.....	کردن تغییرات commit
21	قواعد commit ها
21.....	کامنت‌گذاری درست
21.....	تغییرات مرتبط در یک کامیت
22.....	تست و چک کردن تغییرات قبل از کامیت
23	تاریخچه ریپو
24gitignore فایل
26	برنجها.....
26.....	شروع به کار
30	ساخت برج
33.....	چرا برجها
34	سناریوی ۱
34	سناریوی ۲
36.....	نمونهٔ یه ساختار خوب برای branch ها
38	مرج کردن

39.....	حذف برنج‌ها
40.....	conflict‌ها و برطرف کردن‌شون
40	ایجاد conflict
43	ایجاد conflict (ولی این‌بار برای واقعی)
48	بررسی تغییرات
51.....	آشنایی با remote branch‌ها
51	GitHub
54	GitHub Push کردن تغییرات‌تون به
55.....	درست کردن یک access token
59	clone و fetch دستور
60.....	یک کار راحت‌تر
61.....	push دستور
61	آماده‌سازی
63.....	آشنایی با push
65.....	دربیافت تغییرات برنج جدید توی ریپوی tyler
66	pull request ایجاد
69	pull دستور
69.....	یک خطای رایج
71.....	منابع بیشتر
72	چه چیزی یاد گرفتیم؟

مقدمه

شما روی یه پروژه کار می‌کنید، ساعت‌ها روش وقت می‌ذارین و نهایتاً به یه فایل final می‌رسین. بعد از یه مدت می‌فهمین که «عه، توی final یه تایپو هست»، تایپو رو برطرف می‌کنین و به (2)final می‌رسین. یه هفته بعد رئیس زنگ می‌زنه و می‌گه که «این (2)final که فرستادی بهمان جاش غلطه»، شما هم بهمان جای (2)final رو درست می‌کنید تا به (2)final final برسین. بعد از یه مدت همتیمی‌تون که تازه یادش اومند که پروژه‌ای وجود داره می‌یاد که اسمش رو اول پروژه بنویسه، ولی اشتباهی روی (2)final می‌نویسه و به (3)final می‌رسه، که با (2)final final کلی فرق می‌کنه، بعد اونو برای رئیس می‌فرسته که «اینم فایل تصحیح شده پروژه ما» و رئیس هم که می‌بینه هیچی عوض نشده، شروع می‌کنه به غر زدن.



همین مشکلات توی دنیای برنامه‌نویسی هم وجود دارن، و برای حلشون ابزارهای Version Control به وجود اومدن که معروفترینشون Git. این ابزارها کمک می‌کنن یه تیم چند ده، چند صد یا حتی چند هزار نفره از برنامه‌نویس‌ها بتونن سال‌ها روی یه پروژه کار کنن، بدون این که مجبور بشن فایلی به اسم finalfinalfinalv20ISwearToGodThisIsTheFinalOneFinal درست کنن! حتی تدریس‌یارهای خودتون هم از Git برای مدیریت محتواي کلاستون استفاده می‌کنن.

توی این داکیومنت، ما می‌خوایم با گیت آشنایی بشهیم. گیت، ابزاری بسیار معروف، بسیار گستردۀ و اگر راستش رو بخواید، گاهی بسیار رو اعصابه که توی حدودا همه شرکت‌های برنامه‌نویسی استفاده می‌شه. احتمالاً بعد از خوندن این داک، یه کم طول می‌کشه تا دستتون به Git عادت کنه، ولی نگران

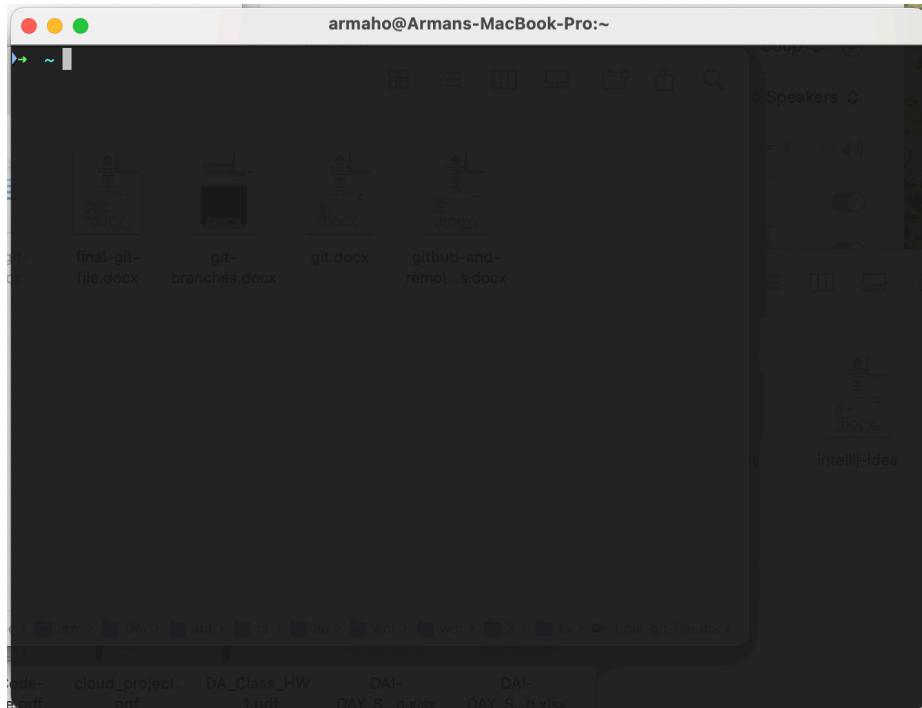
نباشید، کمک راه می‌افتد. ندارید هم طولانی بودن این داک شما رو بترسونه، چون بخش خوبی از اون اسکرین‌شات و خروجی کدهاست.

یادگیری Git اوایل یه کم سخت و پرستادندازه، و احتمالاً به یه سری خطای عجیب و غریب برمی‌خورید که حتی خود لینوس توروالدز هم شاید ندونه چطوری باید حلشون کنه! ولی نگران نباشید. توی این مسیر، حتماً از تدریس‌یارها کمک بگیرید، توی گوگل سرج کنید و از ChatGPT درباره مشکلاتی که بهشون برمی‌خورید سؤال بپرسید. در نهایت، Git برای همه دردرسراحتون یه راه حل داره، پس با خیال راحت جلو ببرید. در انتهای، باید بگیم که **شما توی پروژه‌های آینده این درس، لازمه که خیلی جاها از گیت استفاده کنید، پس مهمه که اون رو کامل همینجا یاد بگیرین.**

حوالستون باشه که پیش‌فرض ما اینه که شما، از داک tools که توی جلسه صفر بهتون داده شده، ابزارهای لازم برای git رو نصب کردین و یه اکانت github ساختین. اگر نکردین، لطفا برگردین و این کارها رو بکنید.

اطمینان از نصب صحیح گیت

اول از همه، باید مطمئن بشین که Git روی سیستمتون نصب شده و درست کار می‌کنه. اگه از مک یا لینوکس استفاده می‌کنین، ترمینال (Terminal) رو باز کنین، و اگه ویندوزی هستین، پاورشل (PowerShell) رو اجرا کنین. به این صفحه‌ای که باز می‌شه، shell می‌گن. ظاهرش هم احتمالاً چیزی شبیه اینه:



توی این صفحه، دستور زیر رو بزنین (حوالتون باشه که \$ رو نزنید!):

```
$ git --version
```

اگر خروجی‌تون شبیه به خروجی زیر نبود، یه جای کار می‌لنگه. به داک tools از جلسهٔ صفر برگردین یا یکی از تدریس‌یارها رو خبر کنین:

```
git version 2.39.5 (Apple Git-154)
```

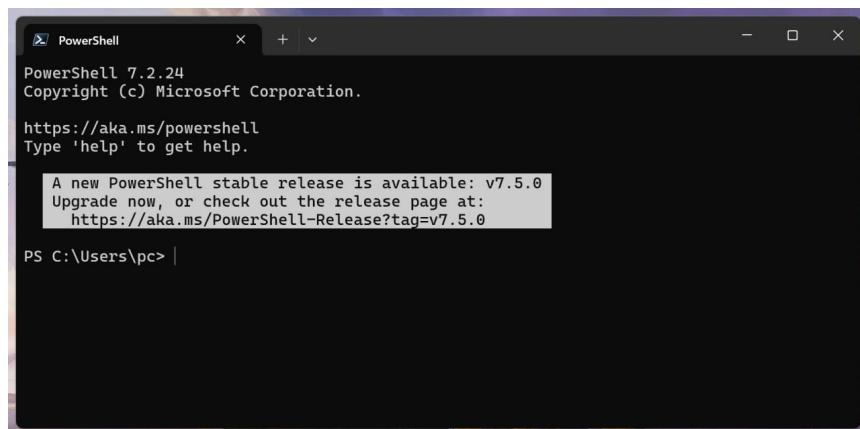
در غیر این صورت، shell‌تون رو باز نگه دارین. توی این داک به اون نیاز داریم.

آشنایی اولیه با shell

اول، لازمه که یه توضیح کوتاه راجع به shell ای دارین باهاش کار می‌کنین بهتون بدیم. شل لینوکس و macOS خیلی به هم شبیه‌ئه، اما شل ویندوز یه مقدار متفاوته.

ویندوز

صفحهٔ فعلی powershell برای شما همچین چیزیه:



می‌بینید که همچین چیزی روی صفحه‌تونه ولی بسته به اسم یوزری که روی لپتاپتون دارین، بخش `pc` اسم متفاوتی داره:

```
PS C:\Users\{your_username} >
```

این یعنی powershell آماده گرفتن دستورات شماست و اونها رو توی مسیر `C:\Users\{your_username}` اجرا می‌کنه. ه این مسیر، اصطلاحاً home directory می‌گن. حالا توی همین دایرکتوری، دستور زیر رو اجرا کنین:

```
$ ls
```

خروجی‌ای مثل خروجی زیر می‌بینید:

Directory: C:\Users\pc				
Mode	LastWriteTime	Length	Name	
----	-----	-----	-----	---
d-r--	12/18/2024 6:27 PM		Contacts	
d-r--	2/13/2025 9:31 PM		Desktop	
d-r--	12/18/2024 6:27 PM		Documents	
d-r--	2/13/2025 7:23 PM		Downloads	
d-r--	12/18/2024 6:27 PM		Favorites	

d-r--	12/18/2024	6:27 PM	Links
d-r--	12/18/2024	6:27 PM	Music
d-r--	6/14/2022	8:11 AM	OneDrive
d-r--	12/18/2024	6:27 PM	Pictures
d-r--	12/18/2024	6:27 PM	Saved Games
d-r--	12/18/2024	6:27 PM	Searches
d---	10/19/2024	8:00 PM	Tracing
d-r--	12/18/2024	6:27 PM	Videos

اگر یه مقدار به این خروجی توجه کنید، می‌بینید که دستور `ls` (که مخفف `list` است) تمام دایرکتوری‌ها و فایل‌هایی که توی `C:\Users\{your_username}` بود رو برآتون نشون داده. حالا دستور زیر رو اجرا کنید:

```
$ cd Documents
```

دستور `cd`, که مخفف `change directory` است، شما رو به دایرکتوری `C:\Users\{your_username}\Documents` می‌بره. می‌بینید که خط بعدی به همچین چیزی تغییر کرده:

```
PS C:\Users\{your_username}\Documents >
```

و اگر الان باز `ls` بگیرین، این بار محتوای پوشیده `Documents` رو می‌بینید. حالا می‌توانید با دستور زیر، به دایرکتوری قبلی‌تون برگردین:

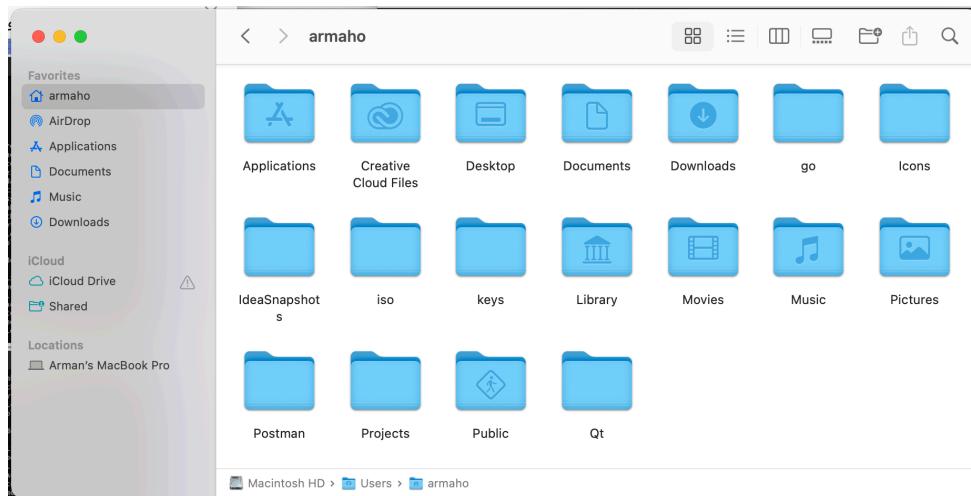
```
$ cd ..
```

لینوکس و macOS

توی لینوکس یا macOS، شما همچین صفحه‌ای پیش روتون دارین. چون من `zsh` دارم ممکنه ظاهر این صفحه برای شما با مال من فرق کنه، ولی شبیه همینه:



این صفحه هنوز خیلی خالیه، ولی اون بالا یه ~ می‌بینین. این نماد نشون‌دهنده home directory شماست. همون جایی که تو ش پوشه‌هایی مثل Desktop، Documents، Videos و Downloads و بقیه چیزایی که همیشه باهاشون سروکار دارین، قرار دارن. یعنی home شما اینجاست:



الآن، shell‌تون آماده گرفتن دستورات شما و اجرашون توی دایرکتوری home. دستور زیر رو توی اون اجرا کنید:

```
$ ls
```

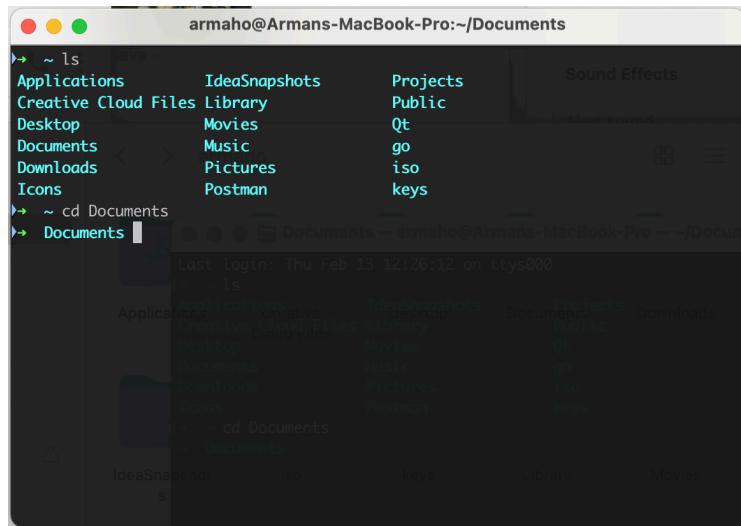
خروجی‌ای مثل خروجی زیر می‌بینید:

Applications	IdeaSnapshots	Projects
Creative Cloud Files	Library	Public
Desktop	Movies	Qt
Documents	Music	go
Downloads	Pictures	iso
Icons	Postman	keys

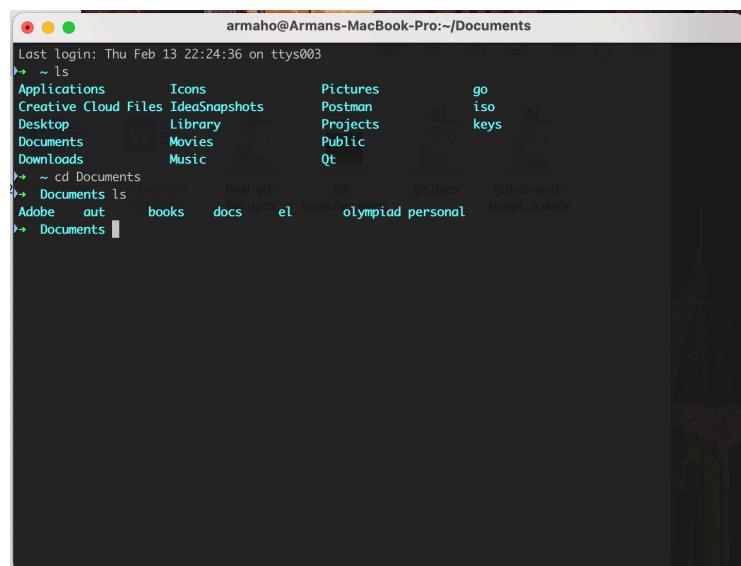
می‌بینید که دستور ls، لیستی از تمام دایرکتوری‌ها و فایل‌هایی که توی home بود رو بهتون برگردونید. حالا با استفاده از دستور زیر، وارد دایرکتوری Documents بشین:

```
$ cd Documents
```

می‌بینید که علامت ~ جای خودش رو به Documents می‌ده و این یعنی شما، با موفقیت وارد پوشۀ Documents شدین:



اگر اینجا هم `ls` بزنید، اینبار لیستی از دایرکتوری‌ها و فایل‌های توی `Documents` می‌بینید:



با استفاده از دستور زیر، به home برگردین:

```
$ cd ..
```

git configure کردن

قبل از استفاده از git، لازمه که بعضی تنظیمات اون رو عوض کنید.

اول از همه، با دستور زیر ایمیل خودتون رو تنظیم کنین. خوبه که این ایمیل رو، همون ایمیلی بذارین که باهاش اکانت گیت‌هابتون رو ساختین:

```
git config --global user.email "molioo1298@gmail.com"
```

بعدش، یه اسم برای خودتون تنظیم کنین:

```
git config --global user.name "Jesper"
```

نهایتا، لازمه که editor که git ازش استفاده می‌کنه رو تنظیم کنید. این ادیتور به شکل دیفالت روی vim تنظیم شده که ممکنه در دفعات اول استفاده برآتون راحت نباشه. اگر از ویدوز استفاده می‌کنید اوون رو روی Notepad تنظیم کنید:

```
$ git config --global core.editor "notepad.exe"
```

و اگر از macOS استفاده می‌کنید، اوون رو رویTextEdit تنظیم کنید:

```
$ git config --global core.editor "open -W -n -aTextEdit"
```

نهایتا، اگر از لینوکس استفاده می‌کنید، می‌توانید بسته به distro اون رو تنظیم کنید، ولی خب وقتی لینوکس دارین بهتره که یه خورده vim یاد بگیرین (=))

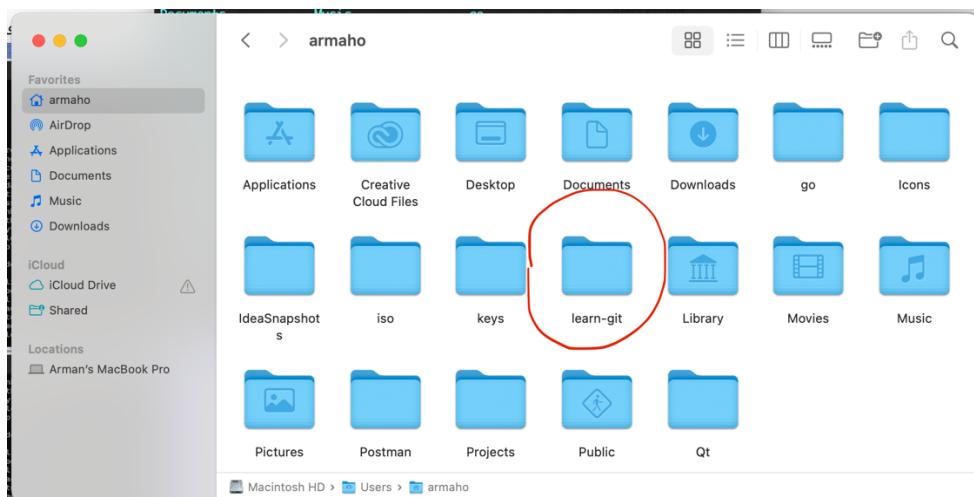
ایجاد ریپوی گیت

اول از همه، باید یه دایرکتوری جدید بسازیم تا پروژه‌مون رو توی اون قرار بدم. توی بخش اول، یاد گرفتیم که شل چطور کار می‌کنه و دوتا دستور مهم اون، یعنی `ls` و `cd` رو هم یاد گرفتیم. حالا قبل از هر چیزی، مطمئن بشین که شل شما توی `home directory` تنظیم شده و نه توی یه دایرکتوری دیگه.

بعد از اون، دستور زیر رو وارد کنید. `mkdir`، که مخفف `make directory` است، توی مسیر فعلی شل براتون یه دایرکتوری به اسم `learn-git` ایجاد می‌کنه:

```
$ mkdir "learn-git"
```

شما بعد از اجرای این دستور، می‌تونید این دایرکتوری رو توی `home` تون ببینید:



با استفاده از دستور `cd`، به این دایرکتوری بروی:

```
$ cd learn-git
```

اگر الان `ls` بزنید، می‌بینید که هیچ چیزی توی این دایرکتوری نیست.

حالا بیاین اولین دستور گیت‌مون رو اجرا کنیم (فقط تیکه `git init` رو بنویسید):

```
$ git init
Initialized empty Git repository in /Users/armaho/learn-git/.git/
```

باید یه خروجی مشابه اون چیزی که بالا دیدید، ببینید. با استفاده از این دستور، ما دایرکتوری `learn-git` رو به یه `repository` تبدیل کردیم. با به اختصار `repo`، به پروژه‌ای گفته می‌شه که با

استفاده از Git به روزرسانی می‌شود. شما می‌توانید توی هر repo تمامی دستورات Git را اجرا کنید.
اگر الان دستور زیر را اجرا کنید:

```
$ ls -a
.
.. .
.git
```

الآن می‌بینید که یه دایرکتوری به اسم .git ایجاد شده. دلیل این که از -a با دستور ls استفاده کردیم اینه که دایرکتوری git. یه دایرکتوری مخفی هست و با ls عادی نمایش داده نمی‌شود. این دایرکتوری شامل همه چیزهایی هست که Git برای مدیریت پروژه‌مون نیاز دارد. ما کار زیادی با این دایرکتوری نداریم و نباید بهش دست بزنیم، ولی هر وقت جایی git دیدیم، بدونید که با یه ریپو گیت سروکار داریم.

حالا، دستور زیر را اجرا کنید:

```
$ git status
On branch main
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

دستور git status به شما وضعیت فعلی ریپو را نشون می‌دهد. چون هنوز هیچ فایل جدیدی اضافه نکردید، بنابراین وضعیت خاصی برای گزارش دادن نیست. اولین خط به شما می‌گوید که روی برنج main هستید (که بعداً توی این داک با برنج‌ها آشنا می‌شید). خط دوم بهتون می‌گوید که تا الان هیچ کامیتی نداشتید (که با کامیت هم توی ادامه این داک آشنا می‌شید). و خط سوم هم بهتون می‌گوید که هیچ فایلی برای کامیت وجود نداره.

داخل learn-git، یه فایل مثل hello.txt ایجاد کنید و توش بنویسید "Hello, Git!". بعدش دوباره git status بزنید:

```
$ git status
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    hello.txt
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

می‌بینید که توی hello.txt Untracked files ظاهر می‌شه. این فایل‌ها، فایل‌هایی‌ان که گیت تغییرات‌شون رو دنبال نمی‌کنه. توی بخش بعد، می‌خوایم این فایل رو stage کنیم.

کردن تغییرات در stage

هر فایلی که توی گیت تغییر می‌دیم، نهایتاً با یه کامیت برای همیشه توی ذهن گیت می‌مونه. به همین دلیل قبل از انجام کامیت، باید تغییرات‌مون رو به دقت بررسی کنیم. توی گیت، Staging Area به شما این امکان رو می‌ده که دقیقاً همین کار رو انجام بدید. این فضا جاییه که فایل‌های انتخاب شده برای کامیت بعدی توش قرار می‌گیرن. می‌تونید بهش به عنوان یه فضای موقت نگاه کنید که تغییرات آماده کامیت شدن رو برای شما نگه می‌داره.

برای stage کردن hello.txt، از دستور زیر استفاده کنین:

```
$ git add hello.txt
```

حالا اگر دوباره git status بزنید، با خروجی زیر مواجه می‌شید:

```
$ git status
On branch main
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   hello.txt
```

می‌بینید که فایل hello.txt، به جای این که توی Untracked files باشه، توی Changes to be committed ئه. این بخش، مختص تغییراتیه که قراره commit بشن. دقت کنید که توی خروجی‌ش یه چیز بامزه هم بهتون گفته: git

```
(use "git rm --cached <file>..." to unstage)
```

گیت داره بهتون یاد می‌ده که برای خروج این فایل از Staging Area (یا به عبارتی، unstage کردنش) می‌تونید از دستور زیر استفاده کنید:

```
$ git rm --cached hello.txt
```

اگر بعد از این دستور، git status بزنید، می‌بینید که این فایل مثل قبل، stage نشده‌ست:

```
$ git status
On branch main
No commits yet
Untracked files:
```

```
(use "git add <file>..." to include in what will be committed)
    hello.txt

nothing added to commit but untracked files present (use "git add" to track)
```

همچنین، با استفاده از دستور زیر، می‌توانید تمام تغییرات stage نشده تمام فایل‌های ریپوتوون رو، کنید بدون این که لازم باشه اسم دونه دونه اون فایل‌ها رو بنویسید:

```
$ git add -A
```

حالا که فایل hello.txt رو stage کردیں، دوباره اون رو تغییر بدین. این بار داخلش به جای "Hello, Git" بزنید: "Hello there, Git!"

```
$ git status

On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   hello.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:  hello.txt
```

می‌بینید که فایل hello.txt هم یک بار به عنوان یه فایل جدید توی بخش Changes to be committed او مده، و هم یک بار دیگه به عنوان فایلی که تغییر کرده توی Changes not staged for commit. در واقع، نسخه‌ای از hello.txt که شما stage کردید، همون نسخه‌ایه که داخلش نوشته‌ید "Hello, Git" و نه نسخه‌ای که تغییرش دادید. برای این که نسخه جدیدترش رو stage کنید، لازمه که دوباره دستور زیر رو اجرا کنید:

```
$ git add hello.txt
```

خیلی خوبه که به خروجی‌ها و دستورات پیشنهادی‌ای که گیت بهتون می‌ده توجه کنیں و اون‌ها رو بخونین و نسبت بهشون کنجکاو باشین. مثلاً توی خروجی قبلی، شما یک دستور جدید می‌بینید:

```
(use "git restore <file>..." to discard changes in working directory)
```

که گیت بهتون برای توصیف این دستور می‌گه "to discard changes in working directory". یعنی اگر شما دستور زیر رو قبل از add کردن تغییراتتون می‌زدین:

```
$ git restore hello.txt
```

باعث می‌شدین تغییرات stage نشده hello.txt از بین برن و داخلش دوباره "Hello, Git" باشه.

خروجی‌های Git بین ابزارهای برنامه‌نویسی واقعاً دقیق و مفیدن و معمولاً پیشنهادهایی که می‌ده، خیلی کارسازن. حتی وقتی که به یه خطاب‌خوردید، Git بهتون دستوراتی پیشنهاد می‌کنه که می‌تونید ChatGPT برای رفع اون خطاب ازشون استفاده کنید. علاوه بر این، در مواقعي که با خطاب مواجه می‌شید، و گوگل هم می‌تونن یار خوبی برای شما باشن.

کردن commit تغییرات

بعد از stage کردن تغییرات‌تون، لازمه برای این که اون‌ها توی حافظه گیت بموون، commit شون بکنیم. برای این کار، اول از همه یه git status بگیرید تا مطمئن باشین همه تغییرات‌تون stage شدن:

```
$ git status

On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   hello.txt
```

حالا که می‌بینیم همه تغییرات stage شدن، اون‌ها رو commit می‌کنیم:

```
$ git commit -m "add a simple hello.txt file"

[main (root-commit) 499871f] add a simple hello.txt file
  1 file changed, 1 insertion(+)
  create mode 100644 hello.txt
```

جلوی دستور git commit ما یه "add a simple hello.txt file" -m نوشته‌یم. به این توضیحات، commit message می‌گن. این توضیحات خیلی مهم هستن و باید خلاصه و دقیق باشن تا کاری که توی اون کامیت انجام شده رو به خوبی توصیف کنن. شما می‌تونستید هر جمله‌ای جای "add a "simple hello.txt file" بنویسید تا مسیح کامیت شما باشه، ولی این مسیح باید به‌طور واضح توضیح بدنه که چی توی کامیت تغییر کرده. اصلاً فکر کردن به یه commit message خوب می‌تونه بخش قابل توجه‌ای از روز شما رو به خودش اختصاص بده!



```
% git add .
% git commit -m '!'
```

حالا اگر با `git log`، تاریخچه repo مون رو بررسی کنیم، می‌توانیم commit مون رو ببینیم:

```
$ git log

commit 499871fd2f48ce02c862569993804a18f11229db (HEAD -> main)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Fri Feb 14 00:06:05 2025 +0330

    add a simple hello.txt file
```

توی این `log`، می‌توانید ببینید که از اسم و ایمیلی که توی بخش «`git config`» کردن استفاده شده و امضای شما، پای این کامیته!

اگر بعد از زدن `git log`، شلتون یه صفحه جدید باز کرد و نتونستین توش دستور جدیدی بزنین، با زدن دکمه Q از اون صفحه بیرون بیایین.

یک فایل C توی `learn-git` به اسم `main.c` درست کنید و محتواش رو مثل کد زیر بنویسین:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    puts("Hello from our git repo!");
    return EXIT_SUCCESS;
}
```

همچنین، یه فایل `goodbye.txt` هم درست کنید و توش بنویسید:

```
Goodbye, Git :(
```

حالا، اگر `git status` بگیرین، با دو فایل `stage` نشده رو به رو می‌شید:

```
$ git status

On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    goodbye.txt
    main.c

nothing added to commit but untracked files present (use "git add" to track)
```

هر دوی اونها را `stage` کنید:

```
$ git add -A
$ git status

On branch main
```

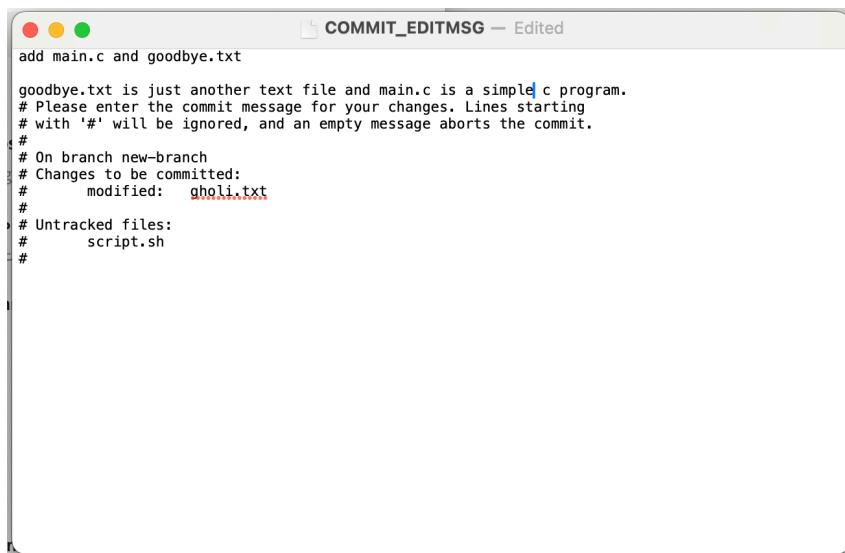
```
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
  new file:   goodbye.txt
  new file:   main.c
```

حالا یه commit جدید برای تغییراتتون بزنید. این بار از `m`- استفاده نکنید:

```
$ git commit

hint: Waiting for your editor to close the file...
```

اینجا، Git از ادیتوری که توی مرحله «`git config`» کردن `configure` مشخص کردید استفاده می‌کنه تا `commit` رو از شما بگیره. شما می‌تونید چند خط پیام به گیت بدید. خط اول به عنوان تیتر پیام در نظر گرفته می‌شه و باقی خطوط به عنوان بدنه اون پیام. همچنین، خطوطی که با `#` شروع می‌شن برای گیت مهم نیستن و می‌تونید اونا رو نادیده بگیرید. من پیام زیر رو برای کامیت خودم مشخص کردم:



بعد از این که پیام کامیت رو نوشتیں، با بستن صفحه ادیتورتون، گیت کامیت رو نهایی می‌کنه (اگر از استفاده می‌کنین، حتما با استفاده از `Command + Q`، یا دکمه `Quit` از taskbar، به شکل macOS کامل `TextEdit` رو ببندین). بعد از این کار، `commit` شما انجام می‌شه و می‌تونید با `git log` اون رو ببینید:

```
$ git log

commit fb32626f45903dbd77a68aebb949c22ddb6de36b (HEAD -> main)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Fri Feb 14 00:40:08 2025 +0330
```

```

add main.c and goodbye.txt

goodbye.txt is just another text file and main.c is a simple c program.

commit 499871fd2f48ce02c862569993804a18f11229db
Author: Arman Hosseini <armanhosseini8787@gmail.com>
Date:   Fri Feb 14 00:06:05 2025 +0330

add a simple hello.txt file

```

دقت کنید که با هر کامیت، صرفا فایل‌هایی که توی Staging Area بودن کامیت می‌شن و باقی فایل‌ها توی این کامیت نخواهند بود. پس شما می‌تونید یه سری از فایل‌هاتون رو توی یه کامیت عوض کنین و باقی‌شون رو توی یه کامیت دیگه.

قواعد commit

کامنت‌گذاری درست

همون‌طور که قبلاً گفتیم، یکی از مهم‌ترین اصول یه کامیت خوب اینه که پیام خیلی واضح و خوبی داشته باشه که دقیقاً توضیح بده چه کاری توی اون کامیت انجام شده. خیلی وقت‌ها ممکنه لازمه که چند دقیقه به مانیتورتون خیره بشین تا بتونید یه کامنت خوب بنویسید.

توی پیام کامیت، حتماً از ساده‌ترین فرم افعال استفاده کنید. مثلًاً اگر فایل hello.txt رو اضافه کردید، کامنت‌هایی مثل "added hello.txt" یا "adding hello.txt" مناسب نیستن. بهتره از خود فعل به شکل ساده، یعنی "add hello.txt" استفاده کنید. البته این مورد ممکنه بسته به جایی که دارید کار می‌کنید و convention های اون، متفاوت باشه.

تغییرات مرتبط در یک کامیت

هر کامیت باید شامل تغییرات مرتبط با یک کار باشه. مثلًاً کامیتی که قبلاً داشتیم و همزمان هم main.c و هم goodbye.txt رو به ریپوی ما اضافه کردیم، چندان کامیت خوبی نبود، چون که همزمان دو تغییر نامربوط به هم رو پوشش می‌داد. بهتر بود که این تغییرات در قالب دو کامیت مختلف انجام می‌شد.

تست و چک کردن تغییرات قبل از کامیت

کدتون رو قبل از کامیت، تست کنید و مطمئن بشید که کار می‌کنه. همچنین تغییراتی که stage شدن رو قبل از انجام کامیت بررسی کنید.

تاریخچه ریپو

همون‌طور که قبلاً تر فهمیدید، شما می‌توانید با دستور `git log`، تاریخچه commit‌های ریپوتون رو ببینید:

```
$ git log

commit fb32626f45903dbd77a68aebb949c22ddb6de36b (HEAD -> main)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Fri Feb 14 00:40:08 2025 +0330

    add main.c and goodbye.txt

    goodbye.txt is just another text file and main.c is a simple c program.

commit 499871fd2f48ce02c862569993804a18f11229db
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Fri Feb 14 00:06:05 2025 +0330

    add a simple hello.txt file
```

کنار هر commit، یه کد عجیب غریب طولانی می‌بینید. مثلاً کد کامیت آخر ما توی این مثال، `fb32626f45903dbd77a68aebb949c22ddb6de36b` کامیته. این کد انقدر خاصه که فقط ۷ حرف اولش هم برای تمایز دادن این کامیت از سایر کامیت‌ها کافیه و شما خیلی وقت‌ها به جای کد کامل این کامیت، فقط چند حرف اولش رو نیاز دارین.

علاوه بر این، می‌توانید با استفاده از فلگ `--oneline`، حالت خلاصه‌تری از `log` ببینین:

```
$ git log --oneline

fb32626 (HEAD -> main) add main.c and goodbye.txt
499871f add a simple hello.txt file
```

.gitignore فایل

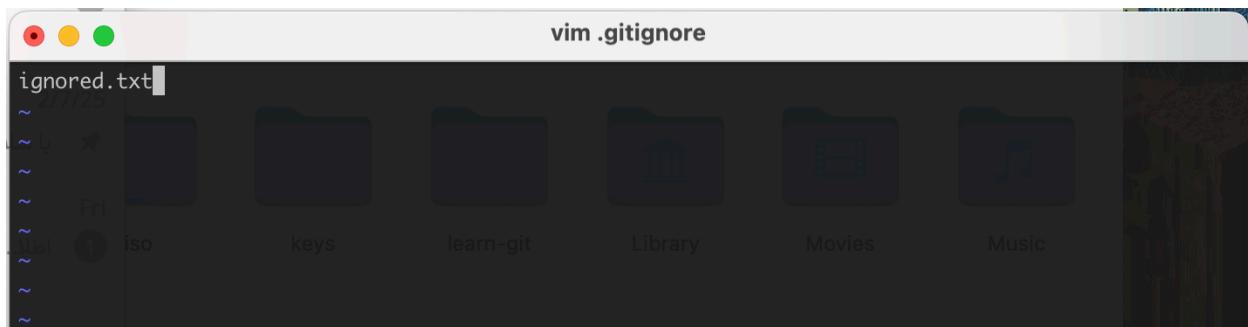
یک فایل جدید به ریپوتوون به اسم ignored.txt اضافه کنید و توش بنویسید: git status بزنید تا اون رو ببینید: "This file will be ignored."

```
$ git status

On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    ignored.txt

nothing added to commit but untracked files present (use "git add" to track)
```

حالا یه فایل به اسم ignored.txt. به ریپوتوون اضافه کنین، و توی خط اول اون بنویسید و ذخیرهش کنید:



دوباره git status بگیرین:

```
$ git status

On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

یه چیزی کمه، نه؟! اگر توجه کنید، دیگه فایل ignored.txt رو نمی‌بینید! انگار که گیت، دیگه تغییرات اون رو دنبال نمی‌کنه و بهش اهمیت نمی‌ده.

فایل .gitignore، فایل خیلی خاصیه. با استفاده از اون، می‌تونید به گیت بگید که فایل‌ها یا دایرکتوری‌های مختلف رو ایگنور کنه و تغییراتشون رو دنبال نکنه. تغییراتتون رو add و commit کنید:

```
$ git add -A

$ git commit -m "add .gitignore"

[main 4e0eff9] add .gitignore
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

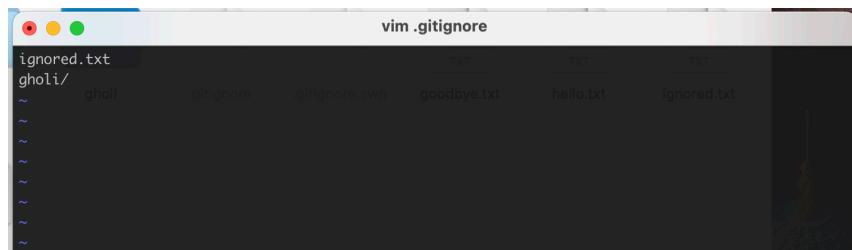
حالا، دایرکتوری‌ای به اسم `gholi` توی ریپوتوون درست کنید، و توی اون دو فایل `1.txt` و `2.txt` رو ایجاد کنید. بعد از این کار، اگر `git status` بزنید با خروجی زیر مواجه می‌شید:

```
$ git status

On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    gholi/

nothing added to commit but untracked files present (use "git add" to track)
```

حالا، به `.gitignore` خط زیر رو اضافه کنید. با اضافه کردن این خط، به `git` می‌گید که کل دایرکتوری `gholi` رو ایگنور کنه:



اگر دوباره `git status` بزنید، می‌بینید که `git` دیگه تغییرات دایرکتوری `gholi` رو دنبال نمی‌کنه:

```
$ git status

On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   .gitignore

no changes added to commit (use "git add" and/or "git commit -a")
```

تغییراتتون رو `add` و `commit` کنید:

```
$ git add -A

$ git commit -m "update .gitignore"

[main 8bf25ec] update .gitignore
 1 file changed, 1 insertion(+)
```

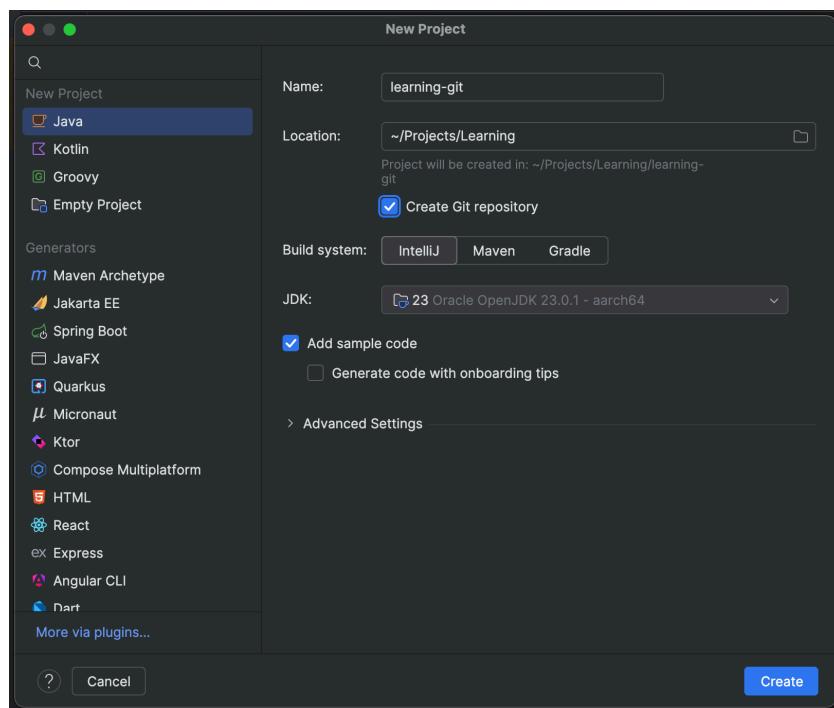
برنج‌ها

حالا که با بعضی از دستورات و مفاهیم Git آشنا شدیم، می‌توانیم راجع به شاخه‌ها (branches) توی اون صحبت کنیم. Branch یکی از مهم‌ترین ویژگی‌های Git هست که باعث می‌شه دهها، صدها یا حتی هزاران برنامه‌نویس بتونن همزمان روی پروژه‌های مشترکی مثل Mozilla، Linux و خیلی از پروژه‌های دیگه کار کنن.

برنج‌ها به یک تیم از برنامه‌نویس‌ها اجازه می‌دن تا بدون تاثیرگذاری روی کار هم‌دیگه، همزمان یک code base را تغییر بدن، فیچر جدید اضافه کنن، خطاهای را رفع کنن، یا هر کار دیگه‌ای انجام بدن. هر برنامه‌نویس قبل از شروع کدنویسی، یک برنج از main می‌گیره، کارش رو روی برنج جدید انجام می‌ده، و در نهایت هم با merge کردن، تغییراتش رو روی برنج main منتقل می‌کنه. اگر این جمله گیجتون کرد، بباید از اول با برنج‌ها آشنا بشیم.

شروع به کار

برای شروع به کار، اولش یه پروژه جدید توی IntelliJ به اسم learning-git ایجاد کنید. تیک Create Git repository رو هم بزنید:



با زدن این تیک، خود IntelliJ برای پروژه‌تون یه ریپوی git درست می‌کنه، یه فایل .gitignore. تو ش می‌ذاره و کارهای اولیه رو انجام می‌ده، تا شما بتونین راحت‌تر با git کار کنین. مثلاً اگر به فایل .gitignore که برآتون ساخته شده یه نگاه بندازین، می‌بینید که خیلی از دایرکتوری‌هایی که IDE مختلف برای خودشون درست می‌کنن و ربطی به پروژه‌تون نداره اون‌جا هست:

The screenshot shows the IntelliJ IDEA interface. In the Project tool window on the left, there's a tree view of a project named 'learning-git'. Inside 'src' > 'Main', there's a '.gitignore' file. The right-hand editor pane displays the contents of this file, which is a standard .gitignore template with many patterns to ignore temporary files and build artifacts from various IDEs.

```

1  ### IntelliJ IDEA ####
2  out/
3  !**/src/main/**/out/
4  !**/src/test/**/out/
5
6  ### Eclipse ####
7  .apt_generated
8  .classpath
9  .factorypath
10 .project
11 .settings
12 .springBeans
13 .sts4-cache
14 bin/
15 !**/src/main/**/bin/
16 !**/src/test/**/bin/
17
18 ### NetBeans ####
19 /nbproject/private/
20 /nbbuild/
21 /dist/
22 /nbdist/
23 /.nb-gradle/
24
25 ### VS Code ####
26 .vscode/
27
28 ### Mac OS ####
29 .DS_Store

```

لطفاً به انتهای فایل .gitignore، خط زیر رو اضافه کنید تا فایل‌های .iml هم ایگنور بشن، و گرنه در ادامه داک یه خورده اذیت می‌شین:

*.iml

کد زیر، که تو ش مجموع ارقام عدد ورودی رو به دست می‌اریم، یه کد نمونه‌ست برای این که branch را یاد بگیریم. این کد رو توی Main.java کپی و پیست کنید و اجراس کنید تا مطمئن بشین درست کار می‌کنه:

```

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        var scn = new Scanner(System.in);
        var input = scn.nextInt();

        System.out.printf("Sum of digits: %d", sumOfDigits(input));
    }

    public static int sumOfDigits(int number) {
        var sum = 0;
    }
}

```

```

        while (number > 0) {
            sum += number % 10;
            number /= 10;
        }

        return sum;
    }
}

```

وقتی که اولین commit مون رو انجام بدیم، اول از همه، با دستور `cd` به دایرکتوری پروژه‌تون برین و بعد، `git status` بزنید:

```

$ git status

On branch main

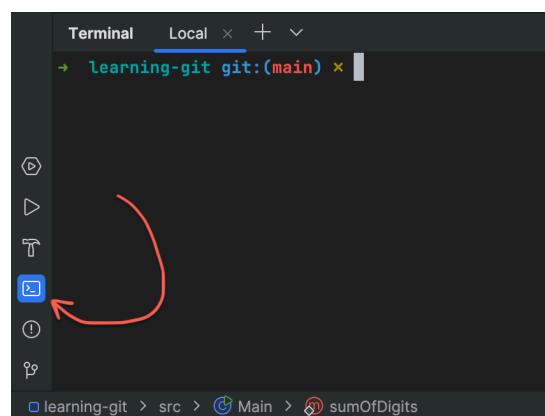
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .gitignore
    new file:   learning-git.iml
    new file:   src/Main.java

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:  .gitignore
    modified:  src/Main.java

```

یک فیچر خوب IntelliJ اینه که توی خودش یه shell داره تا کار شما راحت‌تر باشه. با زدن دکمه `terminal`، می‌تونید به اون دسترسی پیدا کنید:



بعد از زدن این دکمه، صفحه shell زیر IntelliJ برآتون باز می‌شود. همون‌طور که می‌بینید، شبیه همون shell ایه که قبل باهاش کار می‌کردیم، چون راستش دقیقاً همونه! اگر اینجا git status بزنید می‌بینید که خروجی‌ای شبیه به خروجی قبل بهتون می‌دهد:

```

Terminal Local + 
→ learning-git git:(main) ✘ git status
On branch main

No commits yet

Changes to be committed:
(use "git rm --cached <file>..." to unstage)
  new file:   .gitignore
  new file:   learning-git.iml
  new file:   src/Main.java

Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
  modified:   .gitignore
  modified:   src/Main.java

```

برای راحتی کار، در ادامه داک و توی پروژه‌هاتون می‌توانید از این شل استفاده کنید.

توی خروجی git status می‌بینید که تغییرات‌تون توی فایل .gitignore و Main.java هنوز stage نشده‌اند. برای این که همه تغییرات stage بشن، از دستور زیر استفاده می‌کنیم:

```
$ git add -A
```

توی خروجی git status بعدی، می‌بینید که همه تغییرات stage شده‌اند:

```

$ git status
On branch main
No commits yet
Changes to be committed:
(use "git rm --cached <file>..." to unstage)
  new file:   .gitignore
  new file:   learning-git.iml
  new file:   src/Main.java

```

لطفاً با استفاده از دستور زیر، فایل learning-git.iml را unstaged کنید تا گیت برای همیشه ایگنورش کنه:

```
$ git rm --cached learning-git.iml
```

اگر دوباره git status بزنید، خروجی زیر رو می‌بینید:

```
On branch main
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   .gitignore
    new file:   src/Main.java
```

حالا تغییرات رو commit کنید:

```
$ git commit -m "initial commit"
```

حالا باید با زدن git status، خروجی زیر رو ببینید تا مطمئن بشین که همه چیز commit شده:

```
On branch main
nothing to commit, working tree clean
```

جزو چیزهایی که بهترین راه یادگیری اونها، ور فتن باهاشون و دیدن دستوراتشونه. branch پس بریم تا اولین branch مون رو بسازیم.

ساخت برنچ

در ابتدا، دستور زیر رو توى دایرکتوری پروژه‌تون اجرا کنید:

```
$ git branch
```

این دستور، تمامی فعالیت‌های repo رو نشون می‌ده. طبیعتاً، شما باید فقط یک برنچ به اسم main یا master ببینین:

```
* main
```

اگر terminal‌تون توى يه صفحهٔ جدید، مثل صفحهٔ زیر برنچ‌هاتون رو نشون داد، می‌توانید با زدن Q از این صفحه خارج بشین:

A screenshot of a terminal window titled "Local". The title bar also includes "Terminal", a close button ("x"), and a plus sign ("+"). Below the title bar, there is a list of icons on the left side, each with a small description below it. The icons include: a green square with "main", a tilde (~), a blue square with a double-headed arrow (D), a blue square with a right-pointing arrow (R), a blue square with a downward-pointing arrow (T), a blue square with a left-pointing arrow (L), a blue square with a question mark (!), and a blue square with a double question mark (!!). At the bottom right of the terminal window, there is a small box containing the text "(END)".

حالا می‌خوایم یه برنج جدید به اسم support-long ایجاد کنیم. توی این برنج، می‌خوایم یه سری تغییر توی کدمون بدیم تا تابع sumOfDigits از ورودی‌های long پشتیبانی کنه. برای ایجاد برنج جدید، از دستور زیر استفاده می‌کنیم:

```
$ git branch support-long
```

حالا اگر دستور git branch رو بزنید:

```
$ git branch
```

با خروجی زیر مواجه می‌شید:

```
* main
  support-long
```

همون‌طور که می‌بینید، ما با موفقیت branch جدیدمون رو از روی main ساختیم و الان می‌تونیم توی لیست branch‌ها ببینیم. ولی هنوز توی این branch نیستیم، برای این که وارد این branch بشیم، از دستور checkout استفاده می‌کنیم:

```
$ git checkout support-long
```

حالا اگر git branch بزنید، خروجی زیر رو می‌بینید:

```
  main
* support-long
```

می‌بینید که ستاره کوچیکی که قبلاً کنار main بود، الان روی support-long شده. این یعنی ما با موفقیت، branch‌مون رو تغییر دادیم.

کدی که داخل این برنچه، دقیقاً شبیه کد main‌ است. در واقع، این برنچ، کپی کاملی از برنچ main‌ است، چون وقتی درستش کردیم، توی main بودیم. همیشه اگر توی برنچ a باشیم و برنچ b را درست کنیم، برنچ b کپی برنچ a خواهد بود. حالا کدی که توی Main.java بود رو، به کد جدید زیر تغییر بدین:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        var scn = new Scanner(System.in);
        var input = scn.nextLong();

        System.out.printf("Sum of digits: %d", sumOfDigits(input));
    }

    public static int sumOfDigits(long number) {
        var sum = 0;
        while (number > 0) {
            sum += (int) (number % 10);
            number /= 10;
        }

        return sum;
    }
}
```

تفاوت این کد با کد قبلی، صرفاً در اینه که این کد به جای int، قابلیت اینو داره که یک long ورودی بگیره. حالا به ترتیب git status بزنید تا مطمئن باشید که تغییراتتون توی گیت دیده می‌شه و خروجی زیر رو می‌بینید:

```
On branch support-long
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   src/Main.java

no changes added to commit (use "git add" and/or "git commit -a")
```

بعدش هم git add بزنید و دوباره git status بزنید تا مطمئن بشین که تغییراتتون stage شدن:

```
On branch support-long
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   src/Main.java
```

نهایتاً هم، تغییرات رو کامیت کنین:

```
$ git commit -m "add support for long numbers"
[support-long 3f48bc3] add support for long numbers
  1 file changed, 3 insertions(+), 3 deletions(-)
```

اگر الان `git log` بزنید، با همچین خروجی‌ای مواجه می‌شید و می‌فهمید که تغییرات‌تون اعمال شدن:

```
commit 3f48bc3d5c29522283dd4a8c811a72fc78feb4e8 (HEAD -> support-long)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Tue Feb 11 10:07:51 2025 +0330

    add support for long numbers

commit 97ea957914201c480912c24718e352abe42f8e72 (main)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Mon Feb 10 12:30:17 2025 +0330

    initial commit
```

حالا، بباید یه نگاه به برنج `main` بندازیم. این `checkout` رو کنید:

```
$ git checkout main

Switched to branch 'main'
```

فایل `Main.java` رو نگاه کنید. می‌بینید که این فایل، هیچ تغییری نکرده و هنوز هم فقط اعداد `int` رو پشتیبانی می‌کنه! با زدن `git log`، می‌تونید ببینید که `commit` مرتبط با این تغییر هم وجود نداره و فقط `commit` اولیه‌تون رو می‌بینید:

```
commit 97ea957914201c480912c24718e352abe42f8e72 (HEAD -> main)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Mon Feb 10 12:30:17 2025 +0330

    initial commit
```

این که شما می‌تونید کد یه `branch` رو عوض کنید، بدون این که کد `branch` را دیگه تون عوض بشه، از قابلیت‌های اصلی ایه که `git` به شما ارائه می‌ده! با `checkout` کردن دوباره `support-long`، می‌بینید که تغییرات شما بر می‌گردن. شما در ادامه داک، با استفاده از دستور `git merge`، تغییرات برنج `support-long` رو به `main` منتقل می‌کنید.

چرا برنج‌ها

احتمالاً برآتون سوال شده که چرا اصلاً باید از `branch` استفاده کنیم. ارزش این قابلیت، وقتی مشخص می‌شه که با یک اپلیکیشن واقعی‌تر سرو کله می‌زنیم. توی هر دوی سناریوهایی که در ادامه بهتون توضیح می‌دم، می‌تونید کاربردهایی واقعی از `branch` را ببینید. واقعیت اینه که بدون برنج‌ها، کار کردن توی یه تیم برنامه‌نویسی خیلی سخت‌تر از چیزی می‌شد که فکر می‌کنین.

سناریوی ۱

فرض کنید قلی توی یه تیم برنامه‌نویسی در شرکت تپسی کار می‌کنه. یک روز رئیسش بهش می‌گه که قابلیت گرفتن تاکسی دو مقصد رو به سیستم اضافه کنه. قلی هم شروع می‌کنه به کد زدن و تمام روز رو مشغول کار بر روی این ویژگی جدید می‌گذره تا بتونه خیلی زود این امکان رو برای مشتری‌ها فراهم کنه. اما قلی به برنج‌های Git اعتقادی نداره و همه کارهاش رو روی برنج main انجام می‌ده.

انتهای روز، رئیس دوباره میاد بالای سر قلی و می‌گه که آب دستشه بذاره زمین (!) چون سیستم به خطاب خورده و هیچ‌کس نمی‌تونه تاکسی بگیره! حالا قلی باید سریع باگ رو برطرف کنه، ولی کدی که تا الان برای تاکسی دو مقصد زده رو چیکار کنه؟ آیا باید انقدر `Ctrl + Z` بزنه تا همه کدهاش پاک بشن و بتونه به کد اولیه برسه و خطاب رو برطرف کنه؟ از طرفی، قلی باید خیلی سریع باگ فعلی سیستم رو حل کنه و از طرف دیگه هم کل روز کد زده و نمی‌تونه از کدهاش دل بکنه!

قلی می‌تونست که در ابتدا، به جای این که کارش رو روی main انجام بده، یه branch جدید مثل `do-maghsadeh` ایجاد کنه و کارهاش رو اون‌جا انجام بده، وقتی رئیسش با خطاب او مد بالای سرش، خیلی سریع برنج main (که دست نخورده باقی مونده) رو `checkout` کنه تا بتونه خطای مشتری‌ها رو زود برطرف کنه و همزمان، تغییراتش هم توی برنج `do-maghsadeh` حفظ بشه. توی شکل زیر، می‌تونید تصویری از این حالت ببینید:



توی این visualization، که توی دنیای گیت خیلی معروفه، می‌تونید ببینید که برنج `do-maghsadeh` وقتی که قلی کارش رو شروع کرده، از برنج main جدا شده.

سناریوی ۲

قلی و ممد که هر دو در شرکت تپسی مشغول به کار هستن، به تازگی درگیر پیاده‌سازی فیچرهای جدیدی شدن که رئیس ازشون خواسته. قلی چند روزه که روی فیچر «تاکسی دو مقصد» کار می‌کنه و

امروز، رئیس به ممد گفته که روی فیچر جدیدی به نام «هم‌سفر» کار کنه. اما متاسفانه، هر دو هنوز به مفهوم برنج‌ها در Git اعتقادی ندارن و تصمیم دارن کارشون رو مستقیماً روی برنج main پیش ببرن.

قلی به خوبی کامیت می‌کنه و کارش رو به چند قسمت معنادار تقسیم کرده و با پایان هر قسمت، تغییراتش رو کامیت می‌کنه. اما مشکل اینجاست که وقتی ممد می‌رمه تا کارش رو شروع کنه، تغییرات قلی رو توی کدهای برنج main می‌بینه. این تغییرات هنوز به درستی کار نمی‌کنن و باعث می‌شن که اپ تیپسی به کلی بالا نیاد. بنابراین، ممد نمی‌تونه هیچ کاری رو شروع کنه.

ممد که از وضعیت ناامید شده، می‌رمه بالای سر قلی و ازش می‌خواهد که هر چه سریع‌تر کدهاش رو درست کنه تا بتوانه به موقع فیچر هم‌سفر رو تحويل بد. اما قلی که هنوز تحت استرس رفع خطاهایی که داشته، حالش خوب نیست و با ممد بحث می‌کنه و می‌گه که نمی‌تونه حالا حالاها کدش رو درست کنه.

اگر قلی، از ابتدا کارش رو روی برنج جدیدی به اسم do-maghsadeh انجام می‌داد، تغییراتش توی main دیده نمی‌شد و ممد می‌تونست از main، برنج جدیدی مثل hamsafar بگیره و کارش رو شروع کنه، بدون این که نیازی به بحث و دعوا با قلی باشه. نهایتاً هم وقتی development این دو فیچر تموم شد و به خوبی تست شدن، هر دو کدهاشون رو روی برنج main مرج می‌کردن تا فیچرها به دست مشتری برسه. تصویر زیر، این حالت رو نشون می‌ده:

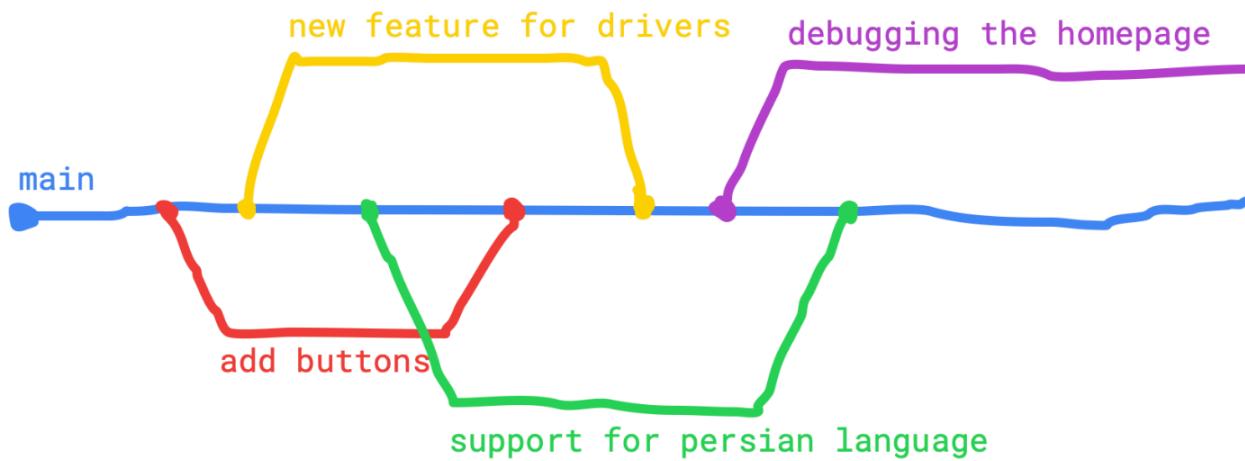


توی این تصویر، می‌تونیم ببینیم که هر دو برنج main، hamdafar و do-maghsadeh که هر دو برنج گرفته شدن و بعد از این که کار اون‌ها تموم شده، توی main مرج شدن و تغییراتشون به main منتقل شده.

نمونهٔ یه ساختار خوب برای branch‌ها

حالا که با branch‌ها آشنا شدیم، خوبه که یه ساختار استانداردتر رو برای اون‌ها بررسی کنیم. با توجه به این که شما قراره بعضی پروژه‌های آینده‌تون رو روی git و به شکل گروهی پیش ببرین، خوبه که از این ساختار الهام بگیرین تا به مشکلات زیادی نخورین.

هر نرم‌افزاری، یه برنج main به عنوان برنج اصلی‌ش داره. این برنج، عموماً شامل یه نسخه stable و تست شده از اون نرم‌افزاره. به هیچ وجه نباید روی خط main، مستقیم commit بکنین! اگر بناست تغییری توی main داده بشه، باید یه برنج از اون بگیرین، تغییرات رو روی یک branch که از main گرفته شده انجام بدین، روی همون برنج تستش کنین، و وقتی که مطمئن شدین که تغییراتتون باگی ندارن، اون‌ها رو توی main مرج کنین:



شکل بالا branches مختلفی می‌باشد. همواره main، برنج رو نشون می‌دیم. برنج stable و تست شده است و کار می‌کنیم. اولین برنچی که گرفتیم، برای اضافه کردن یه سری دکمه به برنامه است. این دکمه‌ها رو توی برنچ مون اضافه می‌کنیم، تستشون می‌کنیم، وقتی فهمیدیم کار می‌کنن، اون‌ها رو به main اضافه می‌کنیم. باقی برنچ‌ها هم به شکلی مشابه درست می‌شن و بعداً توی main مرج می‌شن.

حتی خیلی از شرکت‌ها، به جای این که برنچ‌هاشون رو مستقیم روی main مرج کنن، روی خط دیگه‌ای مثل main تا development می‌برن. حتی از این هم استabilیلتر باشه. ولی شما لازم نیست این کار رو بکنین، بعدتر در اولین کارهای جدی‌تون، یادش می‌گیرین.

تنها کاری که لازمه توی AP و هر موقع دیگه‌ای انجام بدین، اینه که هیچ وقت روی main کامیت نکنین! مگر اولین کامیت برنامه‌تون، اون هم چون هنوز برنچ دیگه‌ای توی ریپو‌تون ندارین. شما فقط می‌تونید به main مرج کنید، حتی برای تغییرات کوچیک. خیلی از شرکت‌ها، دسترسی همه کارکنان غیر از یکی-دوتا از مدیران رو به main کلا می‌بندن.

مرج کردن

وقتی کارتون توی یه برنج تموم شد، می‌تونید با `merge` کردن اون برنج توی برنج‌های دیگه، تغییراتش رو به برنج‌های دیگه منتقل کنید. مثال قبلی‌مون و برنج‌های `main` و `support-long` رو به خاطر بیارین. اول از همه، با دستور `git branch support-long` برنج `support-long` باشید:

```
main
* support-long
```

با `log` git، چک کنید که `commit` آخرتون، مربوط به اضافه کردن قابلیت مرتبه با اعداد `long` بوده:

```
commit 3f48bc3d5c29522283dd4a8c811a72fc78feb4e8 (HEAD -> support-long)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Tue Feb 11 10:07:51 2025 +0330

    add support for long numbers

commit 97ea957914201c480912c24718e352abe42f8e72 (main)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Mon Feb 10 12:30:17 2025 +0330

    initial commit
```

حالا برنج `main` رو `checkout` کنید بريد، و با استفاده از دستور زير، برنج `support-long` رو داخل اون `checkout` کنید:

```
$ git merge support-long

Updating 97ea957..3f48bc3
Fast-forward
  src/Main.java | 6 +-----
  1 file changed, 3 insertions(+), 3 deletions(-)
```

حالا، توی برنج `main` دستور `log` git رو بزنید:

```
commit 3f48bc3d5c29522283dd4a8c811a72fc78feb4e8 (HEAD -> main, support-long)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Tue Feb 11 10:07:51 2025 +0330

    add support for long numbers

commit 97ea957914201c480912c24718e352abe42f8e72
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Mon Feb 10 12:30:17 2025 +0330

    initial commit
```

می‌بینید که commit مرتبط با اعداد long، توی main هم دیده می‌شه! با دیدن فایل Main.java، می‌تونید چک کنید که تغییرات کدها واقعاً توی برنج main اعمال شدن. به این عملیات، merge کردن می‌گن.



حذف برنج‌ها

اگر الان دستور git branch را اجرا کنید، خروجی‌ای شبیه خروجی زیر می‌بینید:

```
* main
  support-long
```

همون‌طور که می‌بینید، برنج support-long اضافه‌ست و دیگه کاربردی نداره، چون تغییراتش توی main مرج شده. اگر این برنج‌ها زیاد بشن، یه خوردۀ برآتون دردسر درست می‌کنن. با استفاده از دستور زیر، می‌تونید برنج‌های قدیمی را پاک کنید:

```
$ git branch -d support-long
Deleted branch support-long (was d5a71e9).
```

خوبه که بعد از اتمام کارتون با یک برنج، با استفاده از این دستور اوون رو حذف کنید تا لیست برنج‌هاتون خالی بمونه.

conflict ها و بروز آنها

ممد و قلی رو به خاطر بیارین، فرض کنیں که توی فرآیند کدنویسی برای فیچرهای جدید تپسی، هر دوی اونها یک تکه از فایل Main.java رو عوض کردن. حالا git باید از کجا بفهمه که کدوم کد، کد درستی برای این فایله؟ کد ممکنه ممد و قلی، به conflict بخورن و یه خوردده اذیت بشن!

بیاین تا با بررسی کد مجموع ارقام، ببینیم که conflict ها چجوری رقم می‌خورن و چطور می‌شه اونها رو درست کرد.



conflict ایجاد

ربیس گفته که ما، با شروع برنامه هیچ پیامی به کاربر نشون نمی‌دیم که بدونه باید یه عدد بهمون ورودی بده و خوبه قبل ورودی گرفتن "Enter a number:" رو پرینت کنیم. همزمان به این هم غر زده که اسم ورودی تابع sumOfDigits رو به جای number بذاریم که متدهای تری داشته باشیم. ما هم برای هر کدوم از این کارها، یک branch از main می‌گیریم:

```
$ git branch fix-message

$ git branch fix-parameter-name
```

برنج fix-message برای پرینت کردن پیام مناسب برای کاربر و برنج fix-parameter-name برای درست کردن اسم پارامتر ورودی sumOfDigits. اول بیاین پیاممون به کاربر رو درست کنیم. برنج مربوط به این کار رو checkout کنید:

```
$ git checkout fix-message
Switched to branch 'fix-message'
```

حالا، کد زیر رو به متدهای main، توی جای مناسب اضافه کنید:

```
System.out.println("Enter a number:");
```

تغییرات رو بررسی کنید و اگر کدتون درست بود، تغییرات رو add و commit کنید تا نهایتاً، کامیتی مثل این رو توی log اتون ببینید:

```
commit d5a71e9ec3242a27ae972896c1f267795912e90a (HEAD -> fix-message)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Tue Feb 11 11:52:40 2025 +0330

    add input message
```

حالا، بعد از این که کارمون توی این برنج تموم شد، به برنج fix-parameter-name می‌ریم:

```
$ git checkout fix-parameter-name
Switched to branch 'fix-parameter-name'
```

اگر به Main.java نگاه کنید، می‌بینید که خطی که "Enter a number" رو چاپ می‌کرد رو نمی‌بینید. حالا، اسم ورودی تابع sumOfDigits رو عوض کنید، طبیعتاً با چیزی که از داک "IntelliJ Tools" یادتونه، باید بتونید به راحتی این کار رو انجام بدید:

```
public static int sumOfDigits(long n) {
    var sum = 0;
    while (n > 0) {
        sum += (int) (n % 10);
        n /= 10;
    }

    return sum;
}
```

تغییراتتون رو بررسی کنید و اگر درست بودن، اونها رو کامیت کنید تا به همچین کامیتی بررسی‌شون:

```
commit 1f6ab68a77eaf63b4a394c49703bd36f3c81c7b1 (HEAD -> fix-parameter-name)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Tue Feb 11 11:57:24 2025 +0330

    shorten parameter name
```

بعد از این که همه تغییرات رو دادین، به برنج main بربین:

```
$ git checkout main
Switched to branch 'main'
```

حالا، برج `fix-message` رو `merge` کنین:

```
$ git merge fix-message
Updating 3f48bc3..d5a71e9
Fast-forward
 src/Main.java | 2 ++
 1 file changed, 2 insertions(+)
```

بعد از اون، برج `fix-parameter-name` رو `merge` کنین. با یه صفحه، مثل صفحه‌ای که برای ایجاد کامنت کامیت‌هاتون می‌بینید مواجه می‌شید. اگر خواستین یه پیام دیگه به جای "Merge branch 'fix-parameter-name'" بنویسین ولی می‌تونین با همین پیام هم پیش بربین.

نهایتاً، به همچین خروجی‌ای می‌رسید:

```
$ git merge fix-parameter-name
Auto-merging src/Main.java
Merge made by the 'ort' strategy.
 src/Main.java | 8 ++++++-
 1 file changed, 4 insertions(+), 4 deletions(-)
```

چی شد؟ خروجی که خوب به نظر می‌رسه. چرا با این که توی هر دو برج‌مون فایل `Main.java` رو عوض کردیم، به مشکلی نخوردیم؟ اگر `Main` رو نگاه کنید، می‌بینید که به طرز جالبی هم ":"Enter a number" کوتاه شده! فایل `sumOfDigits` کوتاه شده! فایل `Main.java` تون، آلان به همچین شکلیه:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        var scn = new Scanner(System.in);

        System.out.println("Enter a number:");
        var input = scn.nextLong();

        System.out.printf("Sum of digits: %d", sumOfDigits(input));
    }

    public static int sumOfDigits(long n) {
        var sum = 0;
        while (n > 0) {
            sum += (int) (n % 10);
            n /= 10;
        }
    }
}
```

```

        return sum;
    }
}

```

به نظر می‌آید که در مورد `merge`‌ها، گیت تا جایی که بتوانه هوشمندانه عمل می‌کنه! گیت، وقتی می‌بینه که موقع مرج دو برنج با هم، فایلی توی هر دو تغییر کرده، تلاش می‌کنه تا اون‌ها را `auto-merge` کنه. اگر به خروجی‌ای که موقع اجرای کامند `merge fix-parameter-name` دیدین دقیق‌تر کنین، رد پای این `auto-merge` را می‌بینید:

```

Auto-merging src/Main.java
Merge made by the 'ort' strategy.
src/Main.java | 8 ++++++-
1 file changed, 4 insertions(+), 4 deletions(-)

```

به خطوط زیر توی خروجی توجه کنین:

```

Auto-merging src/Main.java
Merge made by the 'ort' strategy.

```

می‌بینید که `git`، فایل `src/Main.java` را، با استفاده از استراتژی‌ای به اسم `ort`، `auto-merge` کرده و شما لازم نیست تا با `merge` این دو سر و کله بزنین. حالا بیاین ببینیم که آیا می‌توانیم انقدر گیت رو بپیچونیم تا بالآخره نتونه `merge` را انجام بده و به `conflict` بخوریم؟

ایجاد conflict (ولی این‌بار برای واقعی)

بیاین دوتا برنج درست کنیم و توی هر دوی اون‌ها، خط زیر رو توی برنامه‌مون عوض کنیم تا ببینیم که آیا گیت می‌توانه اون‌ها را `auto-merge` کنه یا نه:

```
System.out.printf("Sum of digits: %d", sumOfDigits(input));
```

برنج‌های `fix-output-1` و `fix-output-2` رو از `main` باشه که برای این کار، برنج فعلی‌تون حتما باید `main` باشه:

```

$ git branch fix-output-1
$ git branch fix-output-2

```

برنج `fix-output-1` رو `checkout` کنید و `printf` رو با `println` جایگزین کنین:

```
System.out.println("Sum of digits: " + sumOfDigits(input));
```

تغییرات رو commit کنیں و به برنج 2 fix-output بینید. باید مجددا خط printf را به همون شکل قبلی بینید. این بار، پیام داخل printf را مثل شکل زیر عوض کنید تا خروجی دقیق‌تری به کاربر نشون بدیم:

```
System.out.printf("Sum of digits in your number: %d", sumOfDigits(input));
```

کدتون رو توی این برنج هم کامیت کنید و بعد، برای انجام مرج به main بردید. اول، 1- fix-output را مرج کنید:

```
$ git merge fix-output-1
Updating 0deaf11..53bf66b
Fast-forward
src/Main.java | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

می‌بینید که تغییرات 1- fix-output به راحتی با main مرج می‌شون و متده printf به println تغییر پیدا می‌کنه. حالا، تلاش کنید تا 2- fix-output رو هم مرج کنید:

```
$ git merge fix-output-2
Auto-merging src/Main.java
CONFLICT (content): Merge conflict in src/Main.java
Automatic merge failed; fix conflicts and then commit the result.
```

خروجی دستوری که وارد کردین رو ببینید. اگر اون رو بخونین، می‌بینید که merge تون به خوده و auto-merge هم ناموفق بوده. اگر الان git status بزنید، با خروجی زیر مواجه می‌شینید:

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   src/Main.java

no changes added to commit (use "git add" and/or "git commit -a")
```

می‌بینید که src/Main.java، توی بخش جدیدی به اسم Unmerged paths دیده می‌شه. این بخش، برای فایل‌هایی که توی عملیات مرج به conflict خوردن. همچنین اگر الان git log بگیرین می‌بینید که هیچ کدام از commit‌های برنج 2 fix-output توانی لگتون نیست.

خب، تبریک! شما به اولین conflict تو نیستید با بررسی فایل Main.java، این رو از نزدیک ببینید. این فایل آن به همچین شکلی در اومده:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        var scn = new Scanner(System.in);

        System.out.println("Enter a number:");
        var input = scn.nextLong();

<<<<< HEAD
        System.out.println("Sum of digits: " + sumOfDigits(input));
=====
        System.out.printf("Sum of digits in your number: %d",
sumOfDigits(input));
>>>>> fix-output-2
    }

    public static int sumOfDigits(long n) {
        var sum = 0;
        while (n > 0) {
            sum += (int) (n % 10);
            n /= 10;
        }

        return sum;
    }
}
```

تیکه‌ای که conflict داشته، اون تکه عجیب و غریب اون وسطه:

```
<<<<< HEAD
        System.out.println("Sum of digits: " + sumOfDigits(input));
=====
        System.out.printf("Sum of digits in your number: %d",
sumOfDigits(input));
>>>>> fix-output-2
```

گیت با تغییر این فایل و اضافه کردن چند خط text، به شما می‌گه که این بخش از کد توی fix با تغییر این فایل و اضافه کردن چند خط text، به شما می‌گه که این بخش از کد توی output-2 به شکل زیر اومده:

```
System.out.printf("Sum of digits in your number: %d", sumOfDigits(input));
```

در حالی که توی HEAD (که اینجا همون main نه) به شکل زیره:

```
System.out.println("Sum of digits: " + sumOfDigits(input));
```

و از شما می‌خواهد این conflict را درست کنید. خوشبختانه، اینجا درست کردن conflict کار راحتیه. تمام این ۶ خط را حذف کنید و کدشون رو با کد زیر جایگزین کنید:

```
System.out.println("Sum of digits in your number: " + sumOfDigits(input));
```

این خط کد، هم پیام درست‌تر برنج 2-fix-output را دارد و هم از printf استفاده می‌کند. به عبارتی، این خط نمایان گر تغییرات هر دو برنج در کنار همه. به خاطر این که merge مون به خورد، لازمه که بعد رفع اون‌ها، توی یه commit جدا git status رو کامل کنیم. دوباره git status بزنید:

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   src/Main.java

no changes added to commit (use "git add" and/or "git commit -a")
```

همون‌طور که می‌بینید، Unmerged paths هنوز توی Main.java هست. حالا که conflict‌های اون رو برطرف کردیم، با git add stage، اون رو کنید:

```
$ git add -A
$ git status
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
  modified:   src/Main.java
```

حالا، تغییرات خودتون رو با پیام مناسب کامیت کنید. این پیام باید مثل متن زیر باشد، کلا متن کامیت‌های مرتبط با merge را همیشه به این شکل می‌ذارن:

```
$ git commit -m "merge 'fix-output-2' into 'main'"
[main 0f8ac7a] merge 'fix-output-2' into 'main'
```

اگر حالا git log بزنید، هم commit‌های برنج 2-fix-output را می‌بینید، هم commit‌هایی که برای آن داشتیں: merge

```
commit 0f8ac7af0359825fba84562565457644013347ef (HEAD -> main)
Merge: 53bf66b 01f5dc0
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Tue Feb 11 14:58:45 2025 +0330

    merge 'fix-output-2' into 'main'

commit 01f5dc0ebd51ce196dd2a14f6e0e71b529ae55e9 (fix-output-2)
Author: Arman Hosseini <armanhosseini878787@gmail.com>
Date:   Tue Feb 11 12:27:31 2025 +0330

    better output message
```

بررسی تغییرات

درسته که commit git status، فایل‌هایی که عوض شدن رو بهتون نشون می‌ده. ولی لازمه که قبل از این که هر فایل چه تغییری کرده رو هم بدونید. اینجا از یکی از ابزارهای IntelliJ برای این کار استفاده می‌کنیم.

یک برنج جدید به اسم product-of-digits درست کنید و checkout کش کنید. اگر می‌خوايد بعد از ساختن یه برنج درجا checkout کنید، می‌توانید به جای دو دستور زیر:

```
$ git branch product-of-digits
$ git checkout product-of-digits
```

از دستور زیر استفاده کنید تا هر دو کار، همزمان انجام بشه:

```
$ git checkout -b product-of-digits
```

توى این برنج، می‌خوایم برنامه‌مون رو عوض کنیم تا به جای مجموع ارقام، ضرب اون‌ها رو نشون بد. برای این کار، برنامه رو تغییر بدید:

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        var scn = new Scanner(System.in);

        System.out.println("Enter a number:");
        var input = scn.nextLong();

        System.out.println("Product of digits in your number: " +
productOfDigits(input));
    }

    public static int sumOfDigits(long n) {
        var sum = 0;
        while (n > 0) {
            sum += (int) (n % 10);
            n /= 10;
        }

        return sum;
    }

    public static int productOfDigits(long n) {
        if (n == 0) {
            return 0;
        }
    }
}
```

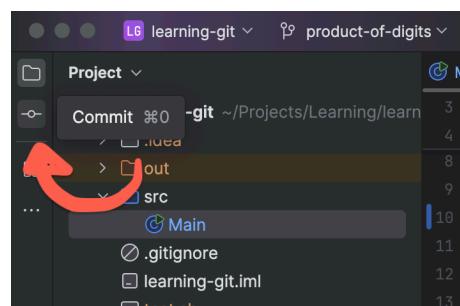
```

var product = 1;
while (n > 0) {
    product *= (int)(n % 10);
    n /= 10;
}

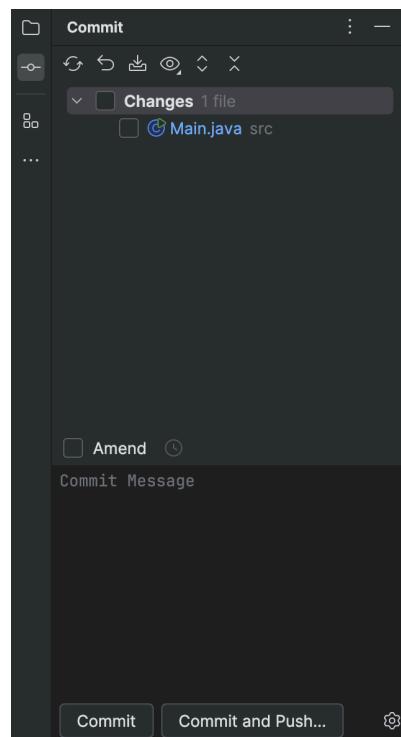
return product;
}
}

```

توی کد جدید، علاوه بر متدهای `sumOfDigits`، متدهای به اسم `productOfDigits` هم داریم که حاصل ضرب ارقام عدد `n` رو برمی‌گردونه. کد جدیدتون رو تست کنید. وقتی دیدین که درست کار کرد، با `git commit` تغییراتتون رو `stage` کنید. برای دیدن تغییراتتون، به بخش `IntelliJ Commit` توی `IntelliJ` بروین:

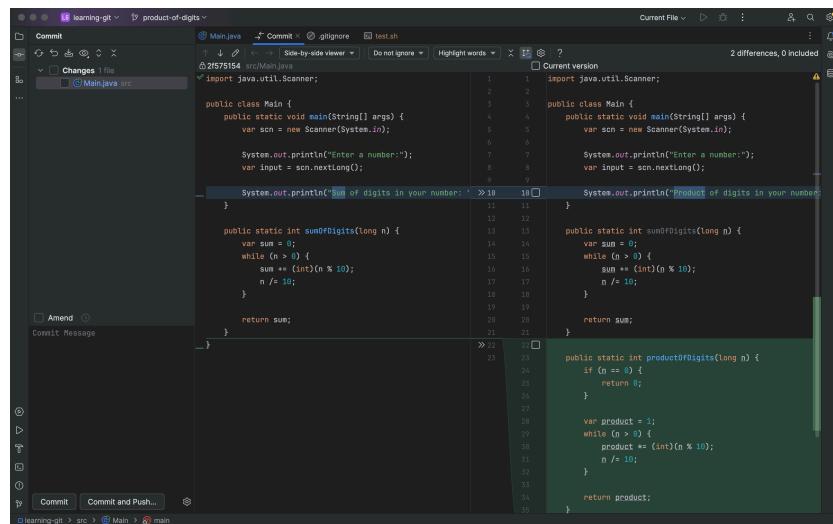


با باز کردن این بخش، پنلی که `IntelliJ` برای `commit` کردن کدهاتون بهتون می‌ده باز می‌شه:



لر IntelliJ به عنوان یک IDE خیلی خوب، خیلی از امکانات گیت رو بدون استفاده از shell در اختیارتون می‌ذاره. توی یه داک دیگه، یه خورده اونها رو بررسی می‌کنیم تا ببینید که چطور می‌شه ازشون استفاده کرد و باهاش کارهاتون راحتتر می‌شه.

توی پنلی که باز شد، روی Main.java کلیک کنید تا صفحه زیر رو ببینید:



```

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        var scn = new Scanner(System.in);

        System.out.println("Enter a number:");
        var input = scn.nextLong();

        System.out.println("Sum of digits in your number: " + sumOfDigits(input));
        System.out.println("Product of digits in your number: " + productOfDigits(input));
    }

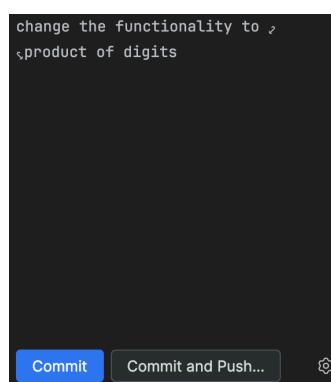
    public static int sumOfDigits(long n) {
        var sum = 0;
        while (n > 0) {
            sum += (int)(n % 10);
            n /= 10;
        }
        return sum;
    }

    public static int productOfDigits(long n) {
        if (n == 0) {
            return 0;
        }

        var product = 1;
        while (n > 0) {
            product *= (int)(n % 10);
            n /= 10;
        }
        return product;
    }
}

```

می‌بینید که IntelliJ چقدر خوب تغییراتتون رو قبل از کامیت نشونتون می‌ده تا بررسی‌شون کنید. کدهایی که اضافه کردین یا تغییر دادین دقیقاً مشخصن. بعد از بررسی کدهاتون، می‌تونید توی بخش Commit Message یه پیام برای کامیتتون بنویسید، فایل‌هایی که می‌خواین commit کنید رو مشخص کنید و بعد با زدن دکمه Commit، اونها رو کامیت کنید:

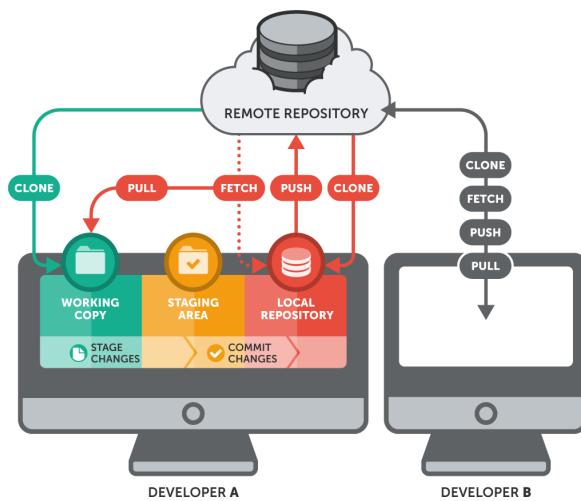


این کار توی shell با دستور git diff انجام می‌ده که خروجیش به اندازه IntelliJ تمیز نیست، ولی اگر خواستین می‌تونین خودتون یه مقدار راجع بهش بخونین و یاد بگیرین، منابع لازم برash توی بخش «منابع بیشتر» انتهای این داک او مده.

آشنایی با remote branch‌ها

اگر یادتون باشه، یکی از مهمترین چیزهایی که راجع به گیت گفتیم، این بود که شما و همکارهاتون می‌تونید روی یه پروژه به شکل تیمی کار کنید، ولی تا حالا هر کاری کردین، فقط و فقط روی لپتاپ خودتون بوده. به ریپوی که روی لپتاپ خودتون باشه، ریپوی محلی یا local می‌گن.

منطقاً شما نمی‌تونید کد یه پروژه بزرگ رو، از یه ریپوی رندوم روی لپتاپ‌تون کنترل کنید! این جاست که مفهوم ریپوی remote مطرح می‌شه؛ یه ریپو روی سرورهای شرکت، که کدها پروژه رو روی اون نگه می‌دارن تا همهٔ توسعه‌دهنده‌ها بتونن بهشون دسترسی پیدا کنن.

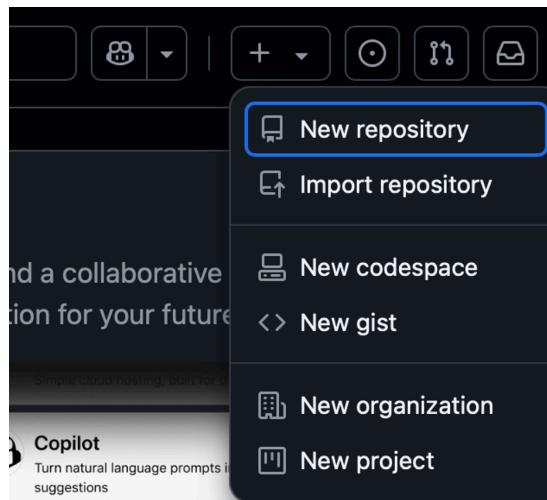


توی این بخش بررسی می‌کنیم که چطور می‌تونید از طریق یک ریپوی remote درست کنید و پروژه‌تون رو به اون وصل کنید.

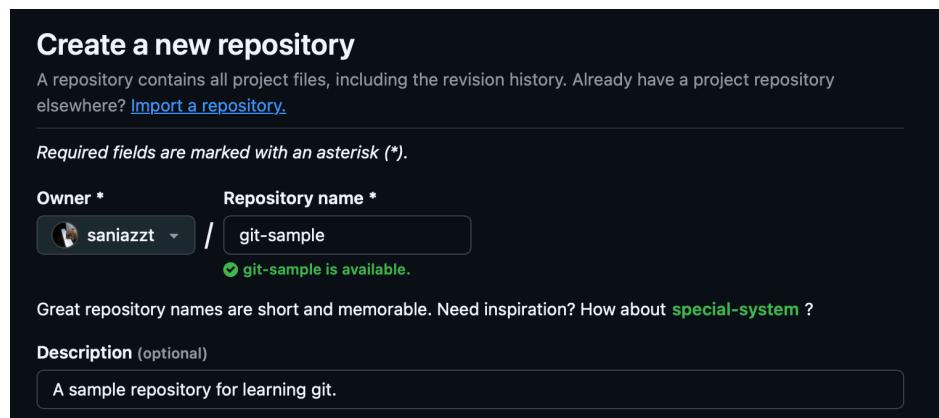
GitHub

گیت‌هاب یک پلتفرم برای به اشتراک گذاشتن کدهایی که از git استفاده می‌کنن. میلیون‌ها برنامه‌نویس کدهای خودشون رو اونجا به صورت متن‌باز (Open Source) به اشتراک می‌ذارن تا برنامه‌نویس‌های دیگه بتونن از اون کدها استفاده کنن و اون پروژه رو توسعه بدن. شما باید از داک tools که توی جلسهٔ صفر منتشر شد، یه اکانت توی GitHub درست کرده باشین، اگر نکردین به این داک برگردین و یه اکانت درست کنید. توی اکانت‌تون لاگین کنید.

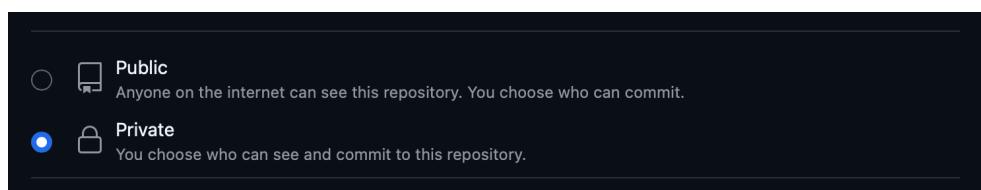
توى مرحله بعد باید يك مخزن يا repository برای پروژه خودتون بسازيد. از طریق زدن علامت + بالای صفحه، New repository رو انتخاب کنید:



حالا وقتشه مشخصات repository رون رو وارد کنید. اسم اوون رو بذارين، مىتونيد يك توضیح کوتاه هم درباره پروژه‌تون توى قسمت Description بنویسيد:

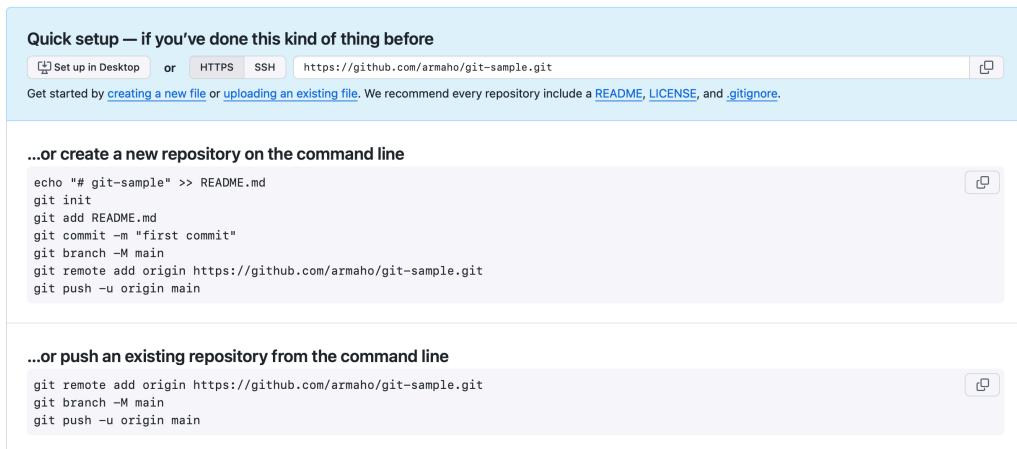


ريپوی شما میتونه public یا private باشه. اگر public باشه همه میتونن اوون رو ببینن و اگر private باشه فقط افرادی که شما مشخص میکنید میتونن. البته توى هر دو حالت، شما تغیین میکنید که کی میتونه ریپوتون رو عوض کنه. فعلا نوع repository رو بگذاريد:



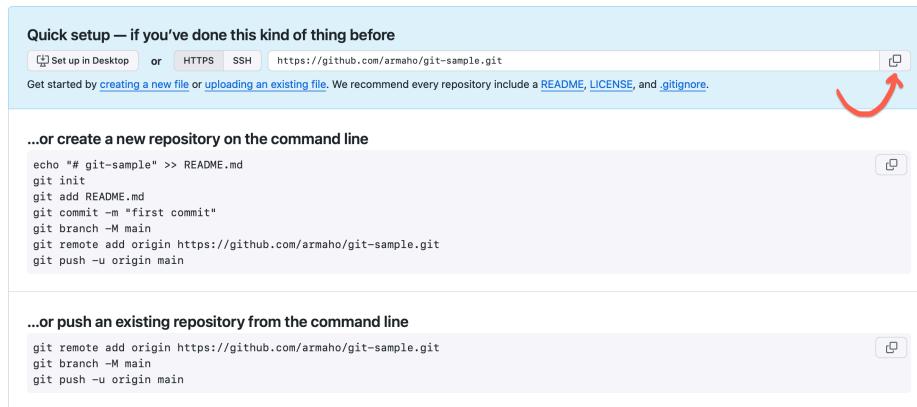
امکان اضافه کردن فایل README برای توضیحات بیشتر، فایل `.gitignore` (که یکم قبل با اون آشنا شدیم) و License که مجوز استفاده از کد شما هست هم توی این مرحله وجود دارد، اما فعلاً با این بخش‌ها کاری نداریم و تیک‌هاشون رو بردارین و اون‌ها رو روی `none` بذارین. حالا روی دکمه سبز `Create repository` در پایین صفحه کلیک کنید.

تبریک می‌گم! شما اولین repository گیت‌هاب خودتون رو ساختید:



GitHub کردن تغییراتتون به Push

از بخش زیر، با زدن دکمه مشخص شده، لینک repo تون روی گیت‌هاب رو کپی کنید (حوالاستون باشه که لینک HTTPS رو کپی کنید نه SSH):



به پروژه learning-git که توی IntelliJ توسعه داده بودیم برگردین، می‌خوایم کدهایی که توی اون زدیم رو به GitHub مون ببریم. دستور زیر رو توی ترمینال تون اجرا کنید و به جای [url]، لینکی که کپی کردیم رو قرار بدین:

```
$ git remote add origin [url]
```

دستورات `git remote`، دستورات مرتبط با repo ریموت پروژه شما هستن. توی دستور بالا، ما از `git` خواستیم که یه ریپوی `remote` جدید، به اسم `origin` و با یک لینک مشخص، به ریپوی شما اضافه کنه تا بتونید تغییراتتون رو بهش اضافه کنید. الان با دستور زیر، می‌توانید لیست ریپوهای ریموتن رو ببینید:

```
$ git remote -v
origin  https://github.com/saniazzt/git-sample (fetch)
origin  https://github.com/saniazzt/git-sample (push)
```

حالا از هر برنچی که هستید با دستور زیر به برنج `main` برگردید:

```
$ git checkout main
```

و با زدن دستور `git status` مطمئن بشید که تغییری روی برنج `main` ندارین. حالا با دستور `push` تمام commit‌ها و فایل‌ها رو از روی برنج `main` به `origin` منتقل کنید. توی این مرحله، لازمه که یوزرنیم و پسورد گیت‌هابتون رو بعد از زدن دستور زیر وارد کنید:

```
$ git push origin main
```

به خطای خور دین، نه؟

```
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/get-started/getting-started-with-
git/about-remote-repositories#cloning-with-https-urls for information on
currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com/armaho/git-sample.git/'
```

درست کردن یک access token

بیاین متن خطای خود را با هم بخونیم. داره می‌گه که ریموت شما، یعنی گیت‌هاب، از ۱۳ آگوست ۲۰۲۱ دیگه اجازه استفاده از پسورد برای کارهایی مثل push یا pull را بهتون نمی‌ده و به همین دلیل، authentication یعنی هم بهتون داده که تو ش می‌توانید شیوه‌های مختلف authentication یعنی موفق نبوده. یه لینک یعنی این لینک برید، متن زیر رو می‌بینید:

Cloning with HTTPS URLs

The `https://` clone URLs are available on all repositories, regardless of visibility. `https://` clone URLs work even if you are behind a firewall or proxy.

When you `git clone`, `git fetch`, `git pull`, or `git push` to a private remote repository using HTTPS URLs on the command line, Git will ask for your GitHub username and password. When Git prompts you for your password, enter your personal access token. Alternatively, you can use a credential helper like [Git Credential Manager](#). Password-based authentication for Git has been removed in favor of more secure authentication methods. For more information, see [Managing your personal access tokens](#).

این متن بهتون می‌گه که:

"When Git prompt you for your password, enter your personal access token"
ولی خب، شما که personal access token ندارید. اشکالی نداره، چون اینجا نوشته که:
"For more Information, see Managing your personal access tokens"

روی این کلیک کنید تا به یه صفحه جدید برید که به شما یاد می‌ده چجوری token بسازید.
راهنمای بخش Creating a personal access token (classic) را دنبال کنید. توی بخش scope، همه تیک‌ها رو بزنین تا این توکن به همه چیز دسترسی داشته باشه و بعدا سر این موضوع اذیت

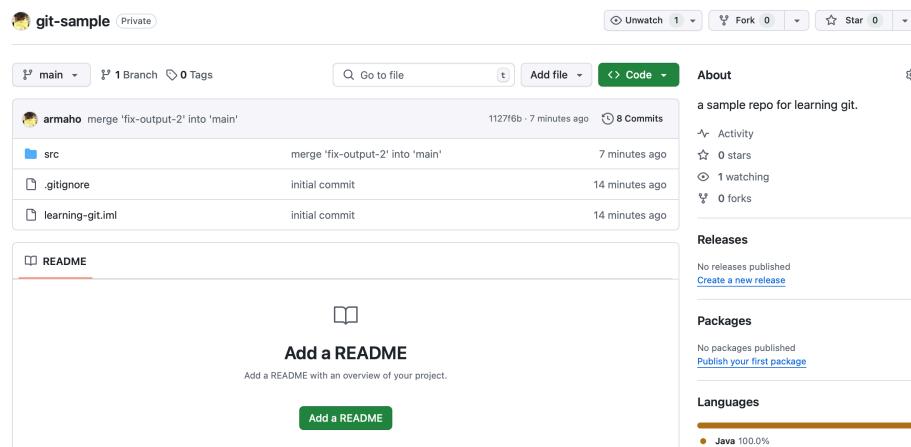
نشین. اون متنی که نهایتاً کپی می‌کنید، همون access token است که باید به جای پسوردتون وارد کنید. حواستون باشه که اونو یه جا نگه دارین چون دیگه بهتون نشون داده نمی‌شه.

دوباره دستور زیر رو اجرا کنید و این بار به جای پسورد، access token رو وارد کنید:

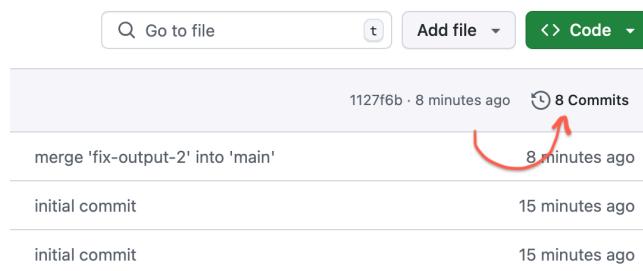
```
$ git push origin main

Username for 'https://github.com': armanhosseini878787@gmail.com
Password for 'https://armanhosseini878787@gmail.com@github.com':
Enumerating objects: 34, done.
Counting objects: 100% (34/34), done.
Delta compression using up to 8 threads
Compressing objects: 100% (26/26), done.
Writing objects: 100% (34/34), 2.99 KiB | 340.00 KiB/s, done.
Total 34 (delta 16), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (16/16), done.
To https://github.com/armaho/git-sample.git
 * [new branch]      main -> main
```

موفق شدین! ریپوتون رو توی گیت‌هاب رفرش کنید:



می‌بینید؟ کدتون الان فقط روی کامپیوتر خودتون نیست، بلکه یک کپی از اون، برای هر کس دیگه‌ای، توی یک remote repo روی گیت‌هاب هم وجود داره. روی commits کلیک کنید:



تمام تاریخچه ریپوتوون اینجا مشخصه:

Commits

All users All time

Commits on Feb 18, 2025

Commit	Author	Date	Message
1127f6b	armaho	10 minutes ago	merge 'fix-output-2' into 'main'
64ae6df	armaho	11 minutes ago	better output message
639ff53d	armaho	11 minutes ago	use println instead of printf
267911a	armaho	11 minutes ago	Merge branch 'fix-parameter-name'
fe36996	armaho	13 minutes ago	shorten parameter name
ca0caf8	armaho	14 minutes ago	add input message
5883389c	armaho	15 minutes ago	add support for long numbers
8f93354	armaho	17 minutes ago	initial commit

اگر روی یکی از کامیت‌ها کلیک کنید، می‌توانید تغییراتی که توی کدتون داده هم ببینید:

Commit 619f53d

armaho committed 12 minutes ago

use println instead of printf

main

1 parent 267911a commit 619f53d

src/Main.java

1 file changed +1 -1 lines changed

```

    7 7     System.out.println("Enter a number:");
    8 8     var input = scn.nextLong();
    9 9
   10 10    System.out.printf("Sum of digits: %d", sumOfDigits(input));
   11 11    System.out.println("Sum of digits: " + sumOfDigits(input));
   12 12
   13 13    public static int sumOfDigits(long n) {

```

اگر حتی یه خورده بیشتر دقت کنید، می‌بینید که ریپوی remote، حتی برنج‌هایی مثل main رو هم توی خودش داره. این یعنی می‌توانید در آینده، باقی branch‌هاتون هم به این ریپو بفرستین:

git-sample Private

main 1 Branch 0 Tags

Switch branches/tags

Find or create a branch...

Branches Tags

main default

View all branches

README

merge 'fix-output-2'

initial commit

initial commit

یه چیز خیلی خوب راجع به گیت اینه که عموما متن‌های خطا و داکیومنتیشن‌هاش خیلی کاملن! خوبه که وقتی به خطایی می‌خورین، چیزی خراب می‌شه یا هر اتفاق دیگه‌ای برآتون می‌افته، حتما متن خطا رو کامل بخونین. دیدین که این‌جا، ما صرفا با خوندن متن خطا، و نه هیچ کمک دیگه‌ای، تونستیم مشکل personal access token‌ها رو برطرف کنیم.

دستور `clone` و `fetch`

باید این بار به جای این که یه ریپوی local درست کنیم و بعد از روی اون، ریپوی remote را بسازیم، از روی ریپوی remote، local رو ایجاد کنیم. کنار دایرکتوری learning-git، که برای پروژه جاواتون درست کردیم، یه دایرکتوری دیگه درست کنید به اسم learning-fetch رو باز کنید و با استفاده از `cd`، به این دایرکتوری جدید بربین.

وقتی که مسیر shell را تنظیم کردیم، دستور `git init` رو وارد کنید تا یه ریپوی خالی ایجاد بشه:

```
$ git init
Initialized empty Git repository in /Users/armaho/Projects/Learning/learning-
fetch/.git/
```

به ریپوی جدید، یه ریپوی remote به اسم origin اضافه کنید که url اش، url ریپوی گیت‌هاستونه:

```
$ git remote add origin [url]
```

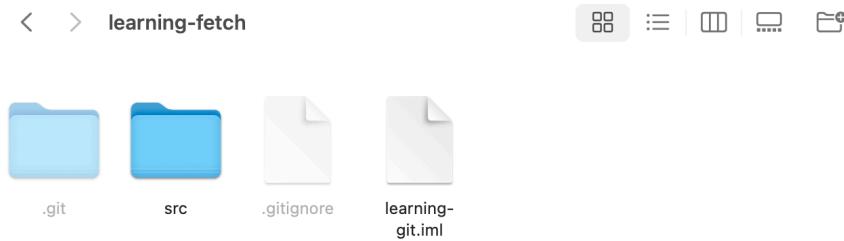
دستور `fetch` رو اجرا کنید. با اجرای این دستور، چیزهایی مثل branch، tagها و باقی چیزای ریپوی origin (که همون ریپوی گیت‌هاستونه) گرفته می‌شوند. با این دستور در ادامه بیشتر کار می‌کنید:

```
$ git fetch origin main
```

حالا دایرکتوری learning-fetch رو باز کنید. چی می‌بینید؟ درسته، با یک فولدر خالی مواجه می‌شید! نکته اینجاست که ریپوی local شما بعد از دستور `fetch` آپدیت نمی‌شوند. برای آپدیت شدن و ادغام کردن کدهای remote با کدهای local باید دستور `merge` زیر رو وارد کنید. این دستور، برنج main که توی ریپوی origin هست رو با main ریپوی local شما merge می‌کنه:

```
$ git merge origin/main
```

حالا دوباره به فولدرتون برگردید، بالاخره می‌بینید که فایل‌هاتون اضافه شدند:



یک کار راحت‌تر

کارهای بالا سخت بودن، نه؟ این که fetch چرا اون جوری کار می‌کنه، یا این که مرجش چرا یه جوری بود؟

برای این که دیگه لازم نباشه این کارها رو بکنید، با استفاده از دستور `cd`، به دایرکتوری پدر-`learning-fetch` برویم (دایرکتوری پدر `learning-fetch`، اونیه که ما `learning-fetch` رو تووش ساختیم). بعدش دستور زیر رو اجرا کنید و طبق معمول، به جای `[url]`، آدرس ریپوو `remote`تون رو بدمی:

```
$ git clone [url]

Cloning into 'git-sample'...
remote: Enumerating objects: 34, done.
remote: Counting objects: 100% (34/34), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 34 (delta 16), reused 34 (delta 16), pack-reused 0 (from 0)
Receiving objects: 100% (34/34), done.
Resolving deltas: 100% (16/16), done.
```

می‌بینید که یه دایرکتوری جدید هم اسم ریپوی گیت‌هابتون ایجاد شد. اگر توی اون رو ببینید، شامل تمام فایل‌هاییه که توی ریپوتون درست کرده بودیم و دیگه لازم نیست کلی دستور برای دریافت یه repo بنویسید.

دستور push

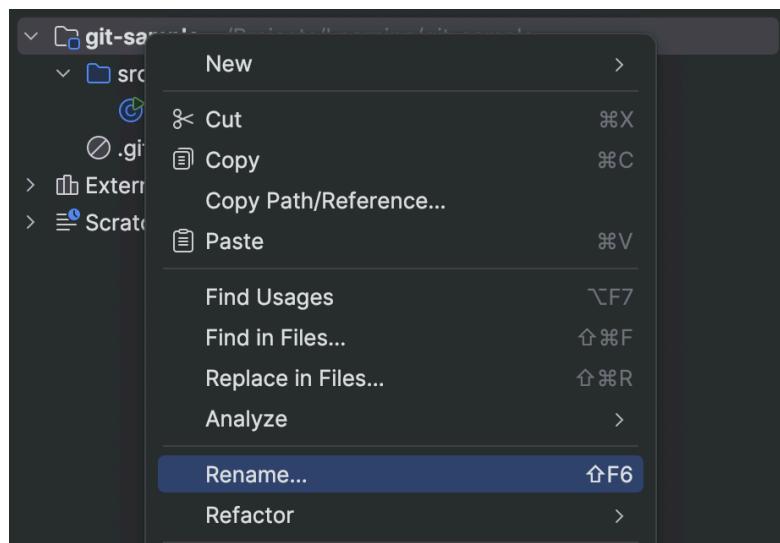
آماده‌سازی

الآن شما دو تا دایرکتوری دارین، یکی learning-git، همون که از اول درست کرده بودین و باهاش کار کردین، و دومی git-sample، همونی که از ریپوی remote‌تون تو بخش قبل کلون کردین. حالا می‌خوایم که شما، خودتون رو به جای دو تا اعضاي يه تیم برنامه‌نویسي، جک و تایلر بذارین.

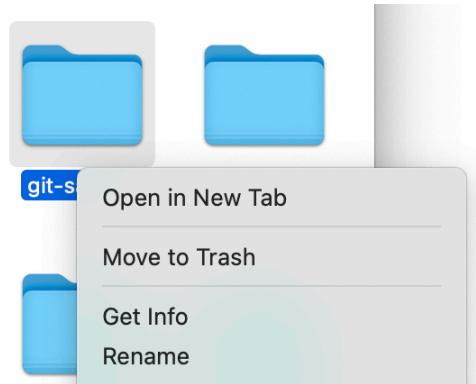
I have two sides



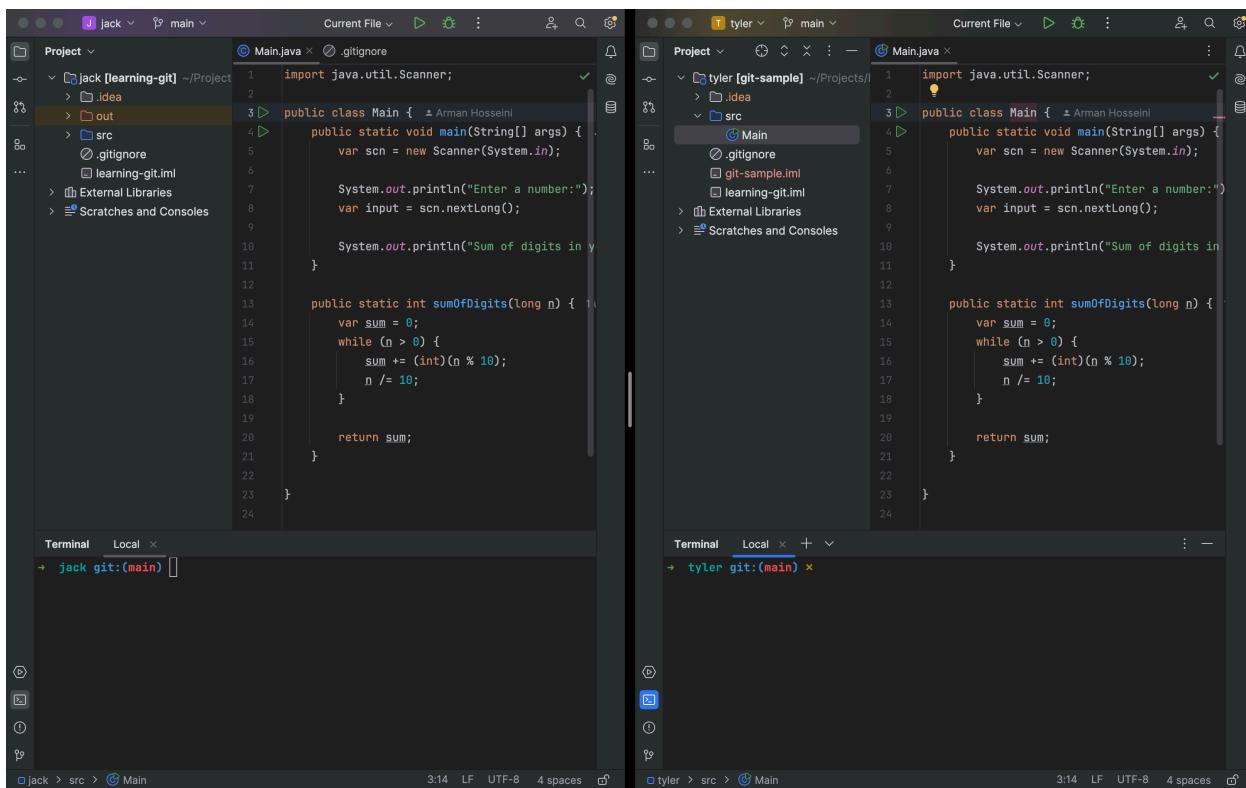
لطفا با استفاده از IntelliJ، اسم پروژه learning-git رو به jack و اسم دایرکتوری git-sample رو به tyler تغییر بدین تا بتونیں این دو دایرکتوری رو راحت‌تر از هم تشخیص بدین:



حتما این کار رو با IntelliJ انجام بدین، و گرنه ممکنه IntelliJ پروژه‌تون رو به اسم‌های قدیمی‌شون بشناسه. IntelliJ فقط اسم پروژه‌هاتون رو عوض می‌کنه و بعدش لازمه خودتون دستی اسم دایرکتوری‌های git-sample و learning-git عوض کنید:



نهایتاً IntelliJ تون رو ببندین. لطفاً دو تا صفحهً IntelliJ برای این دو پروژه کنار هم باز کنین، یکی برای پروژهٔ git-sample و اون یکی برای learning-git. توی هر دو هم shell داخلی IntelliJ رو باز کنید که کارتون راحت‌تر بشه:



آشنایی با push

اگر دقت کنید، توی کدی که الان جلوتونه، متده productOfDigits وجود نداره. علتش اینه که ما هیچ وقت، برنج product-of-digits رو توی main مرج نکردیم و تغییرات اون هیچ‌گاه به main نیومد. ولی شما می‌تونید این برنج رو، همچنان توی ریپوی jack ببینید:

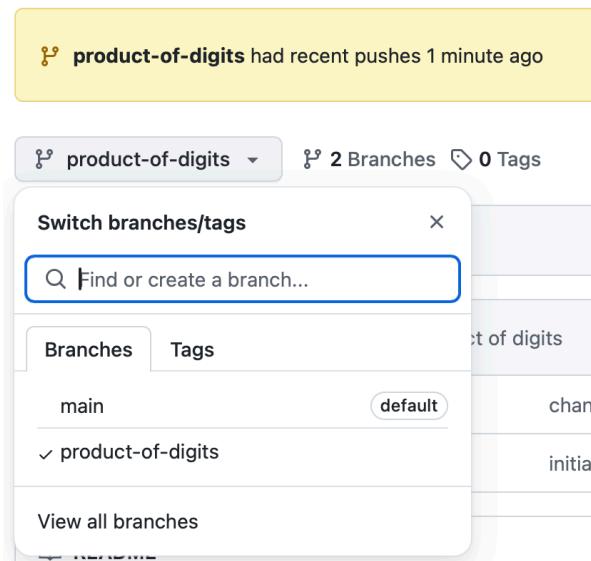
```
fix-message
fix-output-1
fix-output-2
fix-parameter-name
* main
  product-of-digits
```

با استفاده از دستور زیر، این برنج رو به ریپوی remote اضافه کنید:

```
$ git push origin product-of-digits

Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 471 bytes | 471.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'product-of-digits' on GitHub by visiting:
remote:     https://github.com/armaho/git-sample/pull/new/product-of-digits
remote:
To https://github.com/armaho/git-sample.git
 * [new branch]      product-of-digits -> product-of-digits
```

به ریپوی remote روی گیت‌هاب بین. اگر الان برنج‌ها رو ببینید، می‌تونید هم توی لیست برنج‌ها ببینید:



برنامه شما هنوز یه مشکل داره، اون هم این که productOfDigits به ازای اعداد منفی، خروجی غلطی نشون می‌ده:

```
Enter a number:  
-123  
Product of digits in your number: 1
```

برای تصحیح اون، کد productOfDigits رو توی ریپو جک به کد زیر تغییر بدین:

```
public static int productOfDigits(long n) {  
    if (n == 0) {  
        return 0;  
    }  
  
    if (n < 0) {  
        n = -1 * n;  
    }  
  
    var product = 1;  
    while (n > 0) {  
        product *= (int) (n % 10);  
        n /= 10;  
    }  
  
    return product;  
}
```

حالا، تغییرات جدید رو commit کنید. سپس دوباره با دستور زیر، تغییرات رو push کنید:

```
$ git push origin product-of-digits
```

حالا، می‌تونید تغییرات جدید رو هم روی برج github product-of-digits ببینید.

دریافت تغییرات برنج جدید توی ریپوی tyler

حالا به ریپوی tyler برین و اونجا git branch بزنین. میبینید که توی ریپوی tyler، هنوز فقط یه برنج وجود داره: main

```
* main
```

تايلر چطور میتونه توی ریپوی local خودش، تغییرات جک رو روی برنج product-of-digits ببینه و بررسی کنه؟ برای اين کار، اول توی ریپوی تايلر git fetch بزنید:

```
$ git fetch
remote: Enumerating objects: 11, done.
remote: Counting objects: 100% (11/11), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 8 (delta 2), reused 8 (delta 2), pack-reused 0 (from 0)
Unpacking objects: 100% (8/8), 796 bytes | 79.00 KiB/s, done.
From https://github.com/armaho/git-sample
 * [new branch]      product-of-digits -> origin/product-of-digits
```

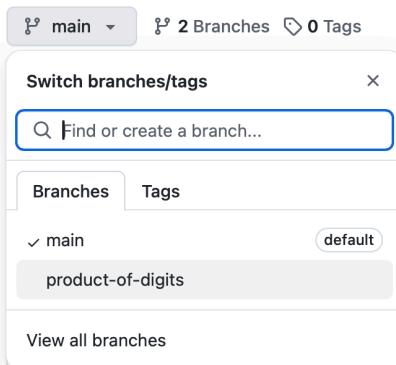
توی خروجی میبینید که یه برنج جدید، به اسم product-of-digits توی ریپوی تايلر ایجاد شد، که معادل checkout origin/product-of-digits رو اگر الان، توی ریپوی تايلر کنید، تغییرات جک رو اونجا میبینید.

ایجاد pull request

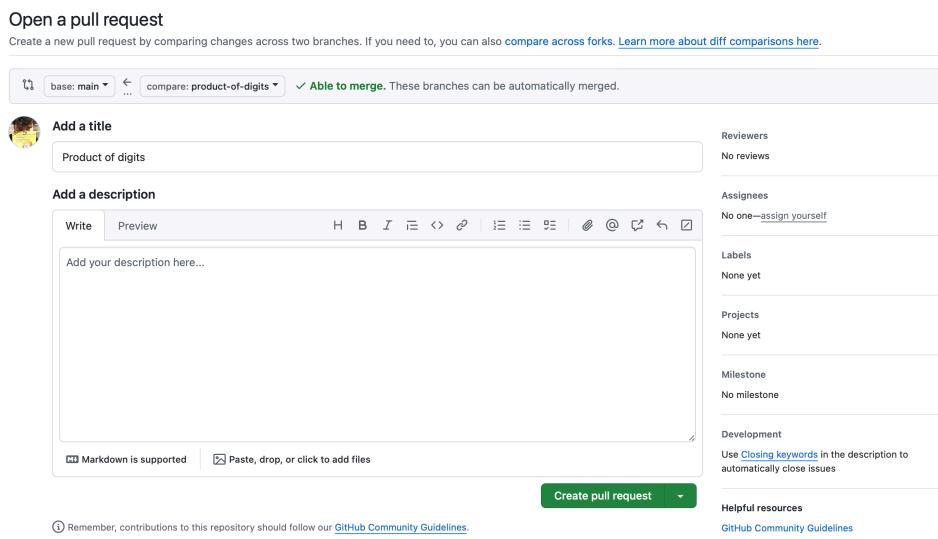
همون طور که گفتیم، شما همین وقت نباید روی برج main کامیت بکنین. بعضی از شرکت‌ها، حتی دسترسی push کردن روی main هم فقط به یکی دو تا از ادمین‌هاشون می‌دان. علتش اینه که توی یه پروژه برج main، عموماً خطیه که کد نهایی و تست شده اون پروژه توشه، به خاطر همین همه تغییرات باید قبل از این که روی برج main برن بررسی بشن.

یکی از راه‌هایی که می‌شه به این بررسی‌ها نظم داد، اینه که برای درخواست‌های merge به خط main، pull request ثبت کرد. بعدها یکی از کسانی که توی شرکت دسترسی داره، این pull request رو می‌کنه و اگر به نظرش بدون مشکل بود، اجازه ادغام اون برج به main رو می‌ده.

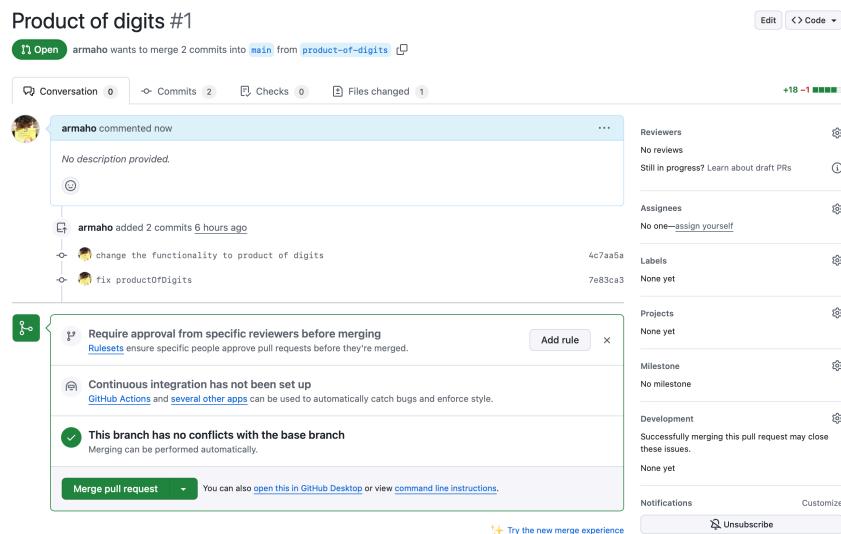
برای این که توی مثال قبلی، main رو داخل product-of-digits مرج کنیم، می‌تونیم از یه pull request استفاده کنیم. برای این کار به ریپوتون توی github یه سر بزنید و اون رو روی برج product-of-digits استارت کنید:



در بخش contribute، گزینه open pull request رو بزنین:



توی اینجا، می‌تونید یک اسم و توضیحات مناسب برای pull request انتخاب کنید. توی یک شرکت بزرگ، از سمت راست می‌تونید reviewerها (مثلاً رئیستون)، assignees (مثلاً همکارهاتون) و سایر چیزها رو هم انتخاب کنید. نهایتاً، دکمه Create pull request رو بزنید:



توی این صفحه، یه سر به تب‌های commits و files changed بزنید، چه چیزایی می‌بینید؟

بعد از بررسی تغییرات، دکمه Merge pull request رو بزنید و مرج رو با کامنتی مناسب تایید کنید. بعد از این کار، اولین pull request‌تون رو انجام دادین و تغییرات توی main مرج شدن. چون برج product-of-digits دیگه به دردی نمی‌خوره، گیت‌هاب بهتون پیشنهاد می‌ده که اون رو پاک کنید:



Pull request successfully merged and closed

You're all set—the `product-of-digi...` branch can be safely deleted.

Delete branch

اگر دوست داشتید، بکنید! حالا اگر توی گیت‌هاب به برج main بردید، می‌بینید که تغییرات جدید توی اون اعمال شده.

دستور pull

به دو صفحهٔ IntelliJ که کنار هم داشتین برگردن و خودتون رو جای jack بذارین. برنج main رو checkout کنید. می‌بینید که main هنوز قدیمیه و تغییرات product-of-digits رونداره. چطور می‌تونید اون رو آپدیت کنید و تغییرات برنج main توی origin رو بگیرین؟

شما با استفاده از git pull، می‌تونید دقیقاً همین کار رو بکنید! این دستور ظاهرش شبیه دستور push‌هه. اون رو اجرا کنید:

```
$ git pull origin main
From https://github.com/armaho/git-sample
 * branch           main      -> FETCH_HEAD
Updating 260148f..a610e32
Fast-forward
  src/Main.java | 19 ++++++-----+
  1 file changed, 18 insertions(+), 1 deletion(-)
```

بعدش فایل Main.java رو چک کنید تا ببینید که آپدیت شده.

یک خطای رایج

حالا، به IntelliJ که برای تایلر درست کردین یه سر بزنید. الان، به عنوان تایلر، خط زیر رو توی جای مناسبی از برنامه‌تون اضافه کنید تا قبل شروع کار، به کاربر سلام بده:

```
System.out.println("Hello there!");
```

تغییراتتون رو روی main کامیت کنید (شما می‌دونید که نباید این کار رو بکنید، ولی تایلر به دلایل آموزشی نمی‌دونه!). بعد از این کار، تلاش کنید تا با استفاده از دستور pull، تغییرات برنج main رو توی origin دریافت کنید:

```
$ git pull origin main
From https://github.com/armaho/git-sample
 * branch           main      -> FETCH_HEAD
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only     # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a
```

```
default
hint: preference for all repositories. You can also pass --rebase, --no-
      rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

می‌بینید که نمی‌شه این کار رو کرد و از fatal که ته خروجی می‌بینید، می‌شه فهمید که خطأ خوردین! بیاین مثل همیشه، با هم متن خطأ را بخونیم. گیت به شما می‌گه که:

"You have divergent branches..."

منظورش اینه که برنج main ای که توی ریپوی local تایلر، یه سری کامیت جدید برای خودش داره، و برنج main ای که توی origin هست هم یک سری کامیت جدید برای خودش، به همین خاطر این دو برنج (واگرا) شدن و گیت نمی‌دونه که کدام main، درسته!

گیت در ادامه بهتون می‌گه که:

"... and need to specify how to reconcile them. You can do so by running one of the following commands sometime before your next pull "

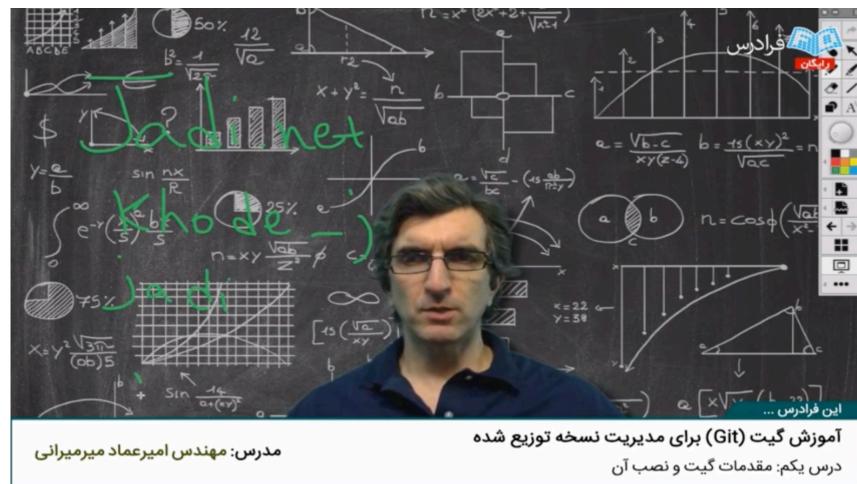
گیت ازتون می‌خواهد که یه شیوه برash مشخص کنید که بدونه چطور باید با برنج‌های divergent برخورد کنه. در ادامه بهتون یه سری دستور می‌ده که با اجرا کردن اون‌ها، می‌توانید این شیوه رو مشخص کنید. اگر دقت کنید، دستور اول به گیت می‌گه که تلاش کنه تا دو برنج divergent رو merge کنه:

```
$ git config pull.rebase false # merge
```

شما هم با اجرای این دستور، این شیوه رو برای گیت مشخص کنید. توی pull بعدی، می‌توانید ببینید که گیت، تغییرات برنج‌های main تایلر و origin/main رو merge می‌کنه. pull کنید تا این موفقیت رو ببینید. دقت کنید که چون شما برای pull کردن، برنج origin/main و main رو merge می‌کنید، ممکنه که اینجا هم به merge conflict بخورین و مجبور شین مثل قبل اون‌ها رو برطرف کنید.

منابع بیشتر

اگر هر جایی از مباحثی که توی این داک گفته شد رو به خوبی متوجه نشدین، می‌تونین از منبعی که خیلی‌ها با اون git رو یاد گرفتن، یعنی [دوره جادی توی فرادرس](#) اون رو دوباره بخونین.



همچنین، اگر خیلی حس شجاعت می‌کنین، می‌تونین یه نگاه به [داک رسمی گیت](#) هم بندازین!

چه چیزی یاد گرفتیم؟

ما توی این داک، یه دور به شما یاد دادیم که چطور می‌تونید از گیت استفاده کنید. این مهارت، هم توی کار آینده‌تون و هم توی ادامه دانشگاه خیلی به دردتون می‌خوره، چون حدودا همه شرکتا از git برای هماهنگ کردن پروژه‌هاشون استفاده می‌کنن. شما یاد گرفتین که:

- گیت چیه، اصلا Version Control به چه دردی می‌خوره.
- commit و stage کردن یعنی چی و چطور می‌شه باهاشون کار کرد.
- برنج‌ها چی‌ان و کجا به درد می‌خورن.
- مرج کردن به چه مفهومه و چطور می‌شه conflict‌ها رو رفع کرد.
- چطور می‌شه با یه remote repository کار کرد.

کار کردن با گیت، در ابتدا ممکنه برای شما یه خوردہ پر فراز و نشیب باشه. حتما توی گوگل سرج کنید راجع به مشکلاتی که بهشون برمی‌خورین، از تدریس‌یارهاتون بپرسین یا از ChatGPT بخوابین که مشکل رو بهتون توضیح بده.