

Git과 GitHub

PM을 위한 협업의 언어

코드를 작성하지 않아도 변경의 흐름을 이해하고,
이슈와 Pull Request를 통해 개발팀과 더 정확하게 협업하는 법

2회

진행 횟수

120분

회차별 시간

실습형

진행 방식

대상	범위
Git을 처음 접하는 PM	Git 기초부터 GitHub 협업까지
실습	완료 기준
PM 문서와 요구사항 변경	PR을 읽고 판단할 수 있음

이 교재를 사용하는 법

각 회차는 **개념** → **PM 관점** → **실습** → **복습** 순서로 진행한다. 명령어를 외우기보다 실행 전후에 무엇이 바뀌는지 설명하는 데 초점을 둔다.

PM 관점은 제품 의도와 출시 조건을, **확인할 결과**는 실습의 완료 기준을, **개념 체크**는 수업 마무리 질문을 뜻한다.

Git과 GitHub를 처음 접하는 PM도 요구사항에서 Pull Request까지의 흐름을 한 번에 이어볼 수 있도록 구성했다.

- Git과 GitHub의 차이를 자신의 말로 설명한다.
- `status`, `diff`, `commit`, 브랜치의 역할을 이해한다.
- Issue와 Pull Request에서 PM이 확인할 항목을 구분한다.
- 병합, 배포, 릴리스가 같은 말이 아님을 설명한다.

차례

00	시작 전 준비	3
01	Git: 변경을 안전하게 기록하기	4
실습	PM 문서로 첫 커밋과 브랜치 만들기	6
02	GitHub: 변경을 함께 검토하기	11
실습	Issue에서 Pull Request까지	15
부록	용어, 명령어, 복습 문제	20

준비

시작 전 준비

설치 문제로 수업 시간을 잃지 않도록 참가자는 아래 항목을 미리 확인한다. 진행자는 실습용 비공개 저장소 하나를 준비한다.

0.1 참가자 준비 체크

1	GitHub 계정	GitHub 계정 을 만들고 브라우저에서 로그인한다.
2	Git 설치	Git 공식 다운로드 에서 설치한 뒤 터미널에서 <code>git --version</code> 을 실행한다.
3	텍스트 편집기	VS Code, Antigravity 또는 평소 쓰는 편집기를 준비한다. 워드프로세서는 사용하지 않는다.
4	사용자 정보	아래 명령으로 커밋에 기록될 이름과 이메일을 설정한다.

```
git config --global user.name "내 이름"
git config --global user.email "GitHub에 등록한 이메일"
git config --global --list
```

개인정보와 인증 주의

회사 정책에 따라 공개 커밋에는 GitHub 비공개 이메일을 사용할 수 있다. 비밀번호, API 키, 고객 데이터, 인증서 파일은 저장소에 올리지 않는다.

제1장 · 1회차 120분

Git: 변경을 안전하게 기록하기

Git을 거대한 명령어 모음이 아니라 “변경을 확인하고, 선택하고, 의미 있는 단위로 기록하는 도구”로 이해한다.

1.1 오늘의 완료 기준

- Git과 GitHub의 역할 차이를 설명할 수 있다.
- 작업 폴더, 스테이징 영역, 커밋의 관계를 그림으로 설명할 수 있다.
- 문서 변경을 확인하고 첫 커밋을 만든다.
- 별도 브랜치에서 변경한 뒤 기본 브랜치로 병합한다.

시간	구간	진행 내용
00-10	오프닝	왜 PM에게 버전 관리가 필요한지 사례로 시작
10-25	개념	Git과 GitHub, 저장소와 버전 이력
25-45	멘탈 모델	작업 폴더 → 스테이징 → 커밋 → 원격 저장소
45-65	첫 저장소	저장소 생성, 상태 확인, 첫 커밋
65-70	휴식	질문 정리 및 환경 문제 확인
70-95	변경 실습	문서 수정, diff 확인, 선택적 커밋
95-112	브랜치	기능 브랜치 생성과 병합
112-120	복습	핵심 오해 점검과 1회차 과제

1.2 Git과 GitHub는 무엇이 다른가

Git은 내 컴퓨터에서 변경 이력을 관리하는 도구이고, GitHub는 그 이력을 팀과 공유하고 검토하는 협업 서비스다.

구분	Git	GitHub
정체	분산 버전 관리 시스템	Git 저장소를 호스팅하는 협업 플랫폼
주요 장소	내 컴퓨터의 작업 폴더와 저장소	웹에 있는 원격 저장소
주요 기능	변경 추적, 커밋, 브랜치, 병합	Issue, Pull Request, 리뷰, 권한, 자동 검사
인터넷	대부분의 기록 작업은 없어도 가능	공유와 협업에 필요
PM 관점	무엇이 언제 바뀌었는지 확인	왜 바꾸고 누가 검토했는지 연결

PM 관점

GitHub 화면만 보더라도 Git의 커밋과 브랜치 개념을 알아야 변경 범위, 리뷰 상태, 병합 가능 여부를 정확히 읽을 수 있다.

1.3 Git의 네 단계

Git은 “저장” 버튼 하나로 모든 것을 처리하지 않는다. 현재 수정한 내용 중 어떤 것을 다음 기록에 포함할지 한 번 더 선택하도록 만들어져 있다.



행동	뜻	비유
파일 저장	편집한 내용을 디스크에 저장	원고 파일을 저장함
<code>git add</code>	다음 기록에 포함할 변경을 선택	이번 발행분에 넣을 문단을 고름
<code>git commit</code>	선택한 변경의 스냅샷과 설명을 기록	버전 번호와 변경 이유를 남김
<code>git push</code>	내 커밋을 원격 저장소에 전송	팀이 볼 수 있는 서가에 올림

꼭 알아야 할 기본 용어

Repository	프로젝트 파일과 변경 이력이 함께 저장되는 공간. 줄여서 repo라고 한다.
Commit	특정 시점의 변경 스냅샷. 작성자, 시각, 설명, 고유 식별값을 가진다.
Branch	기본 흐름을 건드리지 않고 별도의 변경을 진행하는 가벼운 작업선이다.
Diff	두 상태 사이에서 어떤 줄이 추가·삭제·수정되었는지 보여주는 비교 결과다.
Merge	한 브랜치의 변경을 다른 브랜치에 합치는 작업이다.
HEAD	현재 내가 보고 작업 중인 커밋 또는 브랜치를 가리키는 표식이다.

1.4 실습: PM 문서로 첫 커밋 만들기

코드 대신 제품 요구사항 문서를 사용한다. 실습 시나리오 “Acorn Shop 결제 완료 경험 개선” 이다.

실습 1. 저장소와 최초 버전

20분

1. 실습 폴더를 Git 저장소로 만든다.

```
mkdir pm-git-lab
cd pm-git-lab
git init -b main
git status
```

`git init -b main`이 동작하지 않는 구버전에서는 `git init` 후 `git branch -M main`을 실행한다.

2. 편집기에서 `product-brief.md` 파일을 만들고 아래 내용을 저장한다.

```
# 결제 완료 경험 개선

## 문제
고객이 결제 완료 후 주문 상태를 명확히 이해하지 못한다.

## 목표
결제 완료 화면에서 주문 번호와 다음 행동을 안내한다.

## 성공 기준
- 결제 완료 문의 20% 감소
- 주문 상세 이동률 측정
```

3. 변경 상태를 확인하고 첫 커밋을 만든다.

```
git status
git add product-brief.md
git diff --staged
git commit -m "docs: add checkout completion brief"
git log --oneline
```

확인할 결과

`git status`에 “working tree clean”이 표시되고, `git log --oneline`에 첫 커밋이 한 줄로 보이면 완료다.

실습 2. 변경을 보고 선택해서 기록하기

25분

1. 성공 기준에 “결제 완료 화면 이탈률 측정”을 추가하고 파일을 저장한다.
2. 바뀐 줄을 확인한다.

```
git status
git diff
```

3. 변경을 스테이징한 뒤, 실제 커밋될 내용만 다시 확인한다.

```
git add product-brief.md
git diff --staged
git commit -m "docs: add checkout exit metric"
```

PM 관점

좋은 커밋 메시지는 파일 이름보다 변경의 목적을 말한다. “문서 수정” 보다 “결제 완료 이탈 지표 추가” 가 나중에 의사결정 이력을 찾기 쉽다.

실습 3. 별도 브랜치에서 안전하게 변경하기

17분

1. 기능 브랜치를 만든다.

```
git switch -c feature/checkout-copy
git branch
```

2. **product-brief.md**의 목표 아래에 “주문 확인 이메일 발송 시점을 안내한다.”를 추가한다.

3. 변경을 커밋한다.

```
git add product-brief.md
git commit -m "docs: clarify order email timing"
```

4. 기본 브랜치로 돌아가 변경을 합친다.

```
git switch main
git merge feature/checkout-copy
git log --oneline --graph --decorate --all
```

초보자 안전선

이 과정에서는 `git reset --hard`, 강제 `push`, `rebase`를 사용하지 않는다. 무엇을 잃을 수 있는지 모르는 명령은 실행 전에 현재 상태와 복구 방법을 확인한다.

1회차 핵심 정리

- Git은 내 컴퓨터에서 변경 이력을 기록하고, GitHub는 그 이력을 팀과 공유하고 검토한다.
- `git add`는 다음 기록에 담을 변경을 고르고, `git commit`은 선택한 변경을 설명과 함께 남긴다.
- 브랜치는 기본 흐름을 건드리지 않고 변경을 시도하는 별도 작업선이다.

1회차 복습 질문

1. 파일 저장과 커밋은 무엇이 다른가?
2. `git add` 를 실행하는 이유는 무엇인가?
3. 브랜치를 만들면 기본 브랜치의 파일이 즉시 바뀌는가?
4. `git diff` 와 `git diff --staged` 는 무엇을 비교하는가?

1회차 복습 답안

1. 저장은 파일을 디스크에 쓰는 것이고, 커밋은 선택한 변경을 설명과 함께 Git 이력에 기록하는 것이다.
2. 수정한 내용 중 다음 커밋에 포함할 변경을 선택하기 위해서다.
3. 아니다. 브랜치는 별도 작업선이며 병합하기 전까지 기본 브랜치에는 반영되지 않는다.
4. `git diff` 는 아직 스테이징하지 않은 변경, `git diff --staged` 는 다음 커밋에 들어갈 변경을 보여준다.

회차 사이 15분 과제

새 브랜치에서 `release-notes.md` 를 만들고 커밋한 뒤, `git log --oneline --graph --decorate --all` 결과를 가져온다.

제2장 · 2회차 120분

GitHub: 변경을 함께 검토하기

Issue에서 문제를 정의하고, 브랜치에서 변경한 뒤, Pull Request를 통해 검토하고 병합하는 협업 흐름을 실습한다.

2.1 오늘의 완료 기준

- 로컬 저장소와 원격 저장소의 관계를 설명한다.
- Issue, Branch, Commit, Pull Request가 어떻게 연결되는지 설명한다.
- PR의 네 가지 핵심 화면에서 필요한 정보를 찾는다.
- PM 관점의 리뷰 의견을 남기고 병합 여부를 판단한다.

시간	구간	진행 내용
00-10	복습	1회차 용어를 변경 흐름에 다시 배치
10-25	원격 저장소	clone, fetch, pull, push와 origin
25-45	GitHub Flow	Issue → Branch → Commit → PR → Review → Merge
45-65	Issue와 브랜치	문제 정의, 완료 조건, 작업선 만들기
65-70	휴식	작 역할과 저장소 권한 확인
70-95	PR 실습	PR 작성, diff 읽기, 줄 단위 코멘트
95-110	리뷰와 병합	수정 요청, 재검토, merge와 충돌
110-120	정리	배포와 릴리스 구분, 최종 복습

2.2 로컬과 원격을 연결하는 네 동작

동작	무엇을 하는가	PM이 보는 상황
<code>clone</code>	원격 저장소 전체를 내 컴퓨터에 처음 복제	프로젝트에 처음 참여할 때
<code>fetch</code>	원격의 새 이력만 가져오고 내 작업에는 합치지 않음	팀 변경을 먼저 안전하게 확인할 때
<code>pull</code>	원격의 새 이력을 가져와 현재 브랜치에 반영	작업 전 최신 상태를 맞출 때
<code>push</code>	내 로컬 커밋을 원격 저장소로 전송	동료가 내 변경을 보거나 리뷰하게 할 때

origin은 무엇인가

원격 저장소 주소에 붙인 기본 별칭이다. 반드시 GitHub라는 뜻은 아니며, 팀에서 다른 이름을 사용할 수도 있다.

로컬 저장소를 새 GitHub 저장소에 연결하는 선택 실습

GitHub에서 README 없이 빈 저장소를 만든 뒤 화면에 표시되는 주소를 사용한다. 인증에서 시간이 걸리면 아래 실습 대신 GitHub 웹 방식으로 진행한다.

```
git remote add origin https://github.com/USERNAME/pm-git-lab.git
git remote -v
git push -u origin main
```

2.3 GitHub Flow: 변경을 제안하고 합의하는 흐름

브랜치는 안전한 작업 공간이고, Pull Request는 그 변경을 기본 브랜치에 합쳐도 되는지 팀에 묻는 제안서다.

1	→ 2	→ 3	→ 4
Issue	Branch + Commit	Pull Request	Review + Merge
문제, 목표, 완료 조건을 합의	변경을 분리하고 작은 단위로 기록	변경 이유와 결과를 함께 검토	피드백과 검사를 통과해 병합
Issue	버그, 기능, 아이디어, 할 일을 기록하고 논의하는 단위다. 좋은 이슈는 문제와 완료 조건을 명확히 한다.		
Pull Request	한 브랜치의 변경을 다른 브랜치에 병합하자는 제안이다. 설명, 커밋, 검사, diff, 리뷰가 한곳에 모인다.		
Review	의견만 남기거나, 승인하거나, 변경을 요청할 수 있다. 승인은 팀의 규칙과 역할에 따라 의미가 달라진다.		
Checks	테스트, 빌드, 품질 검사처럼 자동화된 검증 결과다. 초록색 표시가 제품 요구사항까지 보장하지는 않는다.		
Merge conflict	Git이 어느 변경을 최종본으로 선택해야 할지 안전하게 판단하지 못하는 상태다.		

PM 관점

Issue는 “무엇을 왜 해결하는가” , PR은 “실제로 무엇이 어떻게 바뀌는가”에 가깝다. 두 기록이 연결되어야 요구사항부터 구현 결과까지 추적할 수 있다.

2.4 PM은 Pull Request에서 무엇을 보는가

PM은 개발자의 코드 품질 판단을 대신하지 않는다. 제품 의도, 정책, 범위, 사용자 영향과 출시 조건이 변경 내용에 정확히 반영되었는지를 확인한다.

먼저 보는 네 화면

화면	확인할 것	PM 질문
Conversation	배경, 목적, 관련 Issue, 논의와 승인	왜 필요한 변경인지 설명되어 있는가?
Commits	변경이 어떤 단위로 쌓였는지	관련 없는 변경이 섞이지 않았는가?
Checks	테스트, 빌드, 자동 검사 결과	실패한 검사가 있는가? 필요한 수동 검증은 무엇인가?
Files changed	실제 추가·삭제·수정된 내용	요구사항과 다른 정책 또는 문구가 들어가지 않았는가?

PM 리뷰 체크리스트

1	문제와 목표	관련 Issue가 연결되어 있고 해결하려는 문제가 한 문장으로 설명되는가?
2	범위	이번에 바꾸는 것과 바꾸지 않는 것이 분리되어 있는가?
3	정책과 예외	권한, 상태, 국가, 채널, 회원 유형별 예외가 빠지지 않았는가?
4	완료 조건	수용 기준과 실제 변경이 대응하며, 테스트 결과나 화면 증거가 있는가?
5	데이터와 측정	필요한 이벤트, 로그, 지표가 포함되었고 개인정보 위험은 없는가?
6	영향 범위	기존 사용자, 연동 시스템, 운영팀, 고객 지원에 미치는 영향이 설명되는가?
7	출시 조건	배포 순서, 기능 플래그, 마이그레이션, 고객 공지가 필요한가?
8	실패 대응	문제가 생겼을 때 되돌리거나 비활성화할 방법과 판단자가 정해졌는가?

2.5 실습: Issue에서 Pull Request까지

두 명이 한 조가 되어 한 명은 변경 작성자, 한 명은 PM 리뷰어를 맡는다. 첫 실습을 마치면 역할을 바꾼다.

실습 1. 좋은 Issue로 시작하기

20분

GitHub 저장소에서 아래 구조로 Issue를 만든다.

배경

결제 완료 후 주문 번호와 다음 행동이 명확하지 않아 문의가 발생한다.

변경 목표

결제 완료 화면의 제목과 안내 문구를 명확하게 한다.

수용 기준

- 제목은 "주문이 완료되었습니다"로 표시한다.
- 주문 번호를 제목 아래에 표시한다.
- "주문 상세 보기" 버튼을 제공한다.
- 주문 상세 이동 이벤트를 기록한다.

제외 범위

- 결제 승인 로직은 변경하지 않는다.
- 주문 확인 이메일은 다음 작업에서 다룬다.

확인할 결과

구현 방법을 몰라도 문제, 사용자 결과, 완료 기준과 제외 범위가 읽히면 충분하다.

실습 2. 브랜치에서 변경하고 PR 열기

25분

1. Issue에서 브랜치를 만들거나 저장소의 Branch 메뉴에서 `feature/checkout-copy`를 만든다.
2. GitHub 웹 편집기에서 `product-brief.md`의 목표와 성공 기준을 수용 기준에 맞게 수정한다.
3. 커밋 메시지를 `docs: align checkout copy with acceptance criteria`로 작성한다.
4. 기능 브랜치에서 `main`으로 보내는 Pull Request를 만들고 아래 내용을 채운다.

변경 요약

결제 완료 화면의 제목, 주문 번호, 상세 이동 조건을 문서에 반영했습니다.

변경 이유

고객이 결제 완료 후 다음 행동을 이해하지 못하는 문제를 줄이기 위해서입니다.

확인 항목

- [] Issue의 수용 기준과 변경 내용을 비교했습니다.
- [] 제외 범위를 건드리지 않았습니다.
- [] 사용자에게 노출되는 문구를 확인했습니다.

위험 및 후속 작업

주문 상세 이동 이벤트 구현은 개발 PR에서 별도로 확인해야 합니다.

Closes #이슈번호

실습 3. PM 리뷰와 수정 반영

25분

1. 리뷰어는 Conversation에서 문제와 Issue 연결을 확인한다.
2. Files changed에서 실제 diff를 읽고 체크리스트 8개 항목을 점검한다.
3. 문구 또는 수용 기준 한 곳에 줄 단위 질문을 남긴다.
4. 작성자는 같은 브랜치에 수정 커밋을 추가한다. PR은 자동으로 업데이트된다.
5. 리뷰어는 수정 내용을 재확인하고 승인 또는 추가 변경 요청을 선택한다.

좋은 리뷰 문장

“이 문구가 틀렸습니다” 보다 “Issue의 수용 기준은 ‘주문이 완료되었습니다’ 입니다. 현재 문구와 다른 이유가 있을까요?” 처럼 근거와 확인 질문을 함께 남긴다.

미니 사례: 이 PR을 승인해도 될까?

Open · checks passed

결제 완료 화면 문구 개선 #42

Issue의 완료 조건: 제목을 “주문이 완료되었습니다”로 변경하고 주문 상세 이동 이벤트를 기록한다.

PR 설명: 결제 완료 화면 문구를 개선했습니다. 별도 테스트는 하지 않았습니다.

```
checkout/success-copy.md
```

- 결제가 완료되었습니다.
- + 주문이 성공적으로 접수되었습니다.

버튼: 주문 상세 보기

- + 장바구니 제목: 구매할 상품

1. Issue와 다른 문구가 사용된 이유는 무엇인가?
2. 장바구니 제목 변경은 이 PR의 범위에 포함되는가?
3. 주문 상세 이동 이벤트가 실제로 반영되었는가?
4. Checks가 통과했어도 어떤 검증 증거가 더 필요한가?

미니 사례 해설

현재 상태에서는 승인보다 변경 요청이 적절하다. 요구 문구와 구현 문구가 다르고, 관련 없는 장바구니 변경이 섞였으며, 이벤트 반영과 수동 테스트 증거가 없다. 자동 Checks 통과만으로 이 세 가지 제품 조건은 확인되지 않는다.

2.6 Merge, Deploy, Release는 같은 말이 아니다

용어	뜻	확인 질문
Merge	변경 이력을 대상 브랜치에 합침	기본 브랜치에 코드가 들어갔는가?
Deploy	실행 가능한 변경을 특정 환경에 배치	개발, 스테이징, 운영 중 어디에 배포했는가?
Release	사용자에게 기능을 제공하거나 공식 버전으로 묶음	사용자가 실제로 접근할 수 있는가?
Launch	공지, 교육, 영업, 운영까지 포함해 시장에 내놓음	고객과 운영 조직이 사용할 준비가 되었는가?

“병합됐으니 출시 완료”라고 말하지 않는다

팀의 배포 구조, 기능 플래그, 앱 심사, 데이터 마이그레이션에 따라 병합과 사용자 노출 사이에는 여러 단계가 있을 수 있다.

충돌이 생겼을 때 PM의 역할

같은 줄에 서로 다른 변경이 생기면 Git은 어느 쪽이 맞는지 결정하지 못한다. 구현 담당자가 충돌을 해결하더라도, 정책이나 문구의 최종 의미는 PM과 관련 책임자가 확인해야 한다.

1. 충돌한 파일과 요구사항을 확인한다.
2. “최신 변경”이 아니라 의도한 최종 상태를 결정한다.
3. 담당자가 충돌을 해결하고 새 커밋을 올린다.
4. PM이 최종 diff에서 제품 의미가 보존됐는지 다시 확인한다.

2회차 핵심 정리

- Issue는 “무엇을 왜 해결하는가”를, Pull Request는 “실제로 무엇이 어떻게 바뀌는가”를 보여준다.
- PM은 코드 품질을 대신 판단하지 않고, 제품 의도 · 정책 · 범위 · 출시 조건이 맞는지 확인한다.
- Merge는 이력 통합이며 Deploy, Release, Launch와 같은 말이 아니다.

부록

현장에서 다시 보는 cheatsheet

명령어를 외우기 위한 목록이 아니라, 현재 상태를 확인하고 다음 행동을 고르기 위한 참고표다.

A.1 기본 명령어

<code>git status</code>	현재 브랜치와 변경·스테이징 상태를 확인한다. 막히면 가장 먼저 실행한다.
<code>git diff</code>	아직 스테이징하지 않은 줄 단위 변경을 확인한다.
<code>git diff --staged</code>	다음 커밋에 실제로 들어갈 변경을 확인한다.
<code>git add FILE</code>	특정 파일의 현재 변경을 다음 커밋 대상으로 선택한다.
<code>git commit -m "MESSAGE"</code>	스테이징된 변경을 설명과 함께 기록한다.
<code>git log --oneline --graph --all</code>	전체 브랜치의 커밋 흐름을 간단한 그래프로 본다.
<code>git switch -c BRANCH</code>	새 브랜치를 만들고 그 브랜치로 이동한다.
<code>git switch BRANCH</code>	기존 브랜치로 이동한다.
<code>git merge BRANCH</code>	지정한 브랜치의 변경을 현재 브랜치에 합친다.
<code>git remote -v</code>	연결된 원격 저장소의 이름과 주소를 확인한다.
<code>git fetch</code>	원격의 새 이력을 가져오되 현재 브랜치에 합치지는 않는다.
<code>git pull</code>	원격의 새 이력을 가져와 현재 브랜치에 반영한다.
<code>git push -u origin BRANCH</code>	현재 브랜치를 처음 원격에 올리고 추적 관계를 설정한다.
<code>git restore --staged FILE</code>	파일 변경은 유지하면서 스테이징만 취소한다.

내용을 버리는 명령은 별도 확인

`git restore FILE` 은 커밋하지 않은 변경을 버릴 수 있다. `reset --hard` 와 강제 `push`는 공유 이력이나 로컬 변경을 잃게 할 수 있으므로 담당 개발자와 함께 처리한다.

A.2 자주 생기는 오해

오해	정확한 이해
커밋하면 GitHub에 올라간다	커밋은 로컬 기록이다. 원격 공유에는 <code>push</code> 가 필요하다.
PR은 코드 작성이 끝난 뒤에만 연다	초기 피드백이 필요하면 Draft PR로 진행 중 변경을 공유할 수 있다.
Checks가 초록이면 출시해도 된다	자동 검사는 정의된 항목만 검증한다. 정책, 문구, 운영 영향은 별도 확인이 필요하다.
Merge는 배포다	Merge는 이력 통합이다. 배포와 사용자 공개는 별도 과정일 수 있다.
충돌은 누군가 실수했다는 뜻이다	동시에 같은 부분을 다르게 바꾸면 자연스럽게 발생할 수 있다.
브랜치를 삭제하면 기록도 사라진다	병합된 커밋과 PR 기록은 남는다. 브랜치 이름만 정리하는 경우가 일반적이다.

A.3 최종 복습

1. Git과 GitHub의 차이를 한 문장씩 설명해보자.
 2. 파일 저장, 커밋, push는 각각 어디에 변화를 남기는가?
 3. Issue와 Pull Request는 각각 어떤 질문에 답해야 하는가?
 4. PM이 Files changed에서 확인해야 할 것은 무엇인가?
 5. Checks 통과와 제품 요구사항 충족은 왜 다른가?
 6. Merge 이후에도 출시 완료라고 단정할 수 없는 이유는 무엇인가?
 7. 충돌이 발생했을 때 “최신 버전” 만 선택하면 위험한 이유는 무엇인가?
 8. 비밀번호나 API 키를 잘못 커밋했다면 단순 삭제 커밋만으로 충분한가?
-

최종 복습 답안과 토론 기준

1. Git은 변경 이력을 관리하는 도구이고 GitHub는 Git 저장소를 공유 · 검토하는 협업 플랫폼이다.
 2. 저장은 작업 파일, 커밋은 로컬 Git 이력, push는 원격 저장소에 변화를 남긴다.
 3. Issue는 무엇을 왜 해결하는지, PR은 무엇이 실제로 어떻게 바뀌는지 답해야 한다.
 4. 요구사항과 실제 변경의 일치, 관련 없는 변경, 정책 · 문구 · 데이터 영향 등을 확인한다.
 5. 자동 Checks는 설정된 검사만 수행하므로 제품 의미와 운영 조건까지 보장하지 않는다.
 6. 별도 배포, 기능 플래그, 심사, 운영 준비가 남아 있을 수 있기 때문이다.
 7. 최신 변경이 반드시 합의한 정책 또는 올바른 최종 상태라는 보장이 없기 때문이다.
 8. 충분하지 않다. 이미 이력에 남고 외부에 노출되었을 수 있으므로 즉시 키를 폐기 · 교체하고 보안 담당 절차를 따라야 한다.
-

A.4 수업 종료 체크

✓	설명	Git과 GitHub, Commit과 Push, Merge와 Deploy의 차이를 말할 수 있다.
✓	상태 확인	<code>status</code> , <code>diff</code> , <code>log</code> 를 보고 현재 상황을 설명할 수 있다.
✓	흐름 이해	Issue에서 PR과 Merge까지의 흐름을 순서대로 설명할 수 있다.
✓	리뷰 판단	PM 체크리스트를 사용해 승인, 질문, 변경 요청 중 하나를 선택할 수 있다.

A.5 진행자 가이드

시작 전 준비 메모

- 실습용 저장소 이름은 `pm-git-lab` 으로 통일한다.
 - 2회차 짝 실습을 위해 참가자를 작성자와 리뷰어로 2명씩 묶는다.
 - 사내 GitHub 접근이나 인증이 불안정하면 2회차는 GitHub 웹 편집 방식으로 진행한다.
 - 터미널이 낯선 참가자에게는 철자보다 실행 전후의 상태 변화를 읽게 한다.
-

Git 멘탈 모델 설명 포인트

- 브랜치를 “복사된 폴더” 라고 설명하지 말고, 커밋을 가리키는 별도 작업선으로 설명한다.
 - 커밋은 자동 저장이 아니라 사람이 의미 있는 변경 묶음을 선택해 남기는 기록이다.
 - PM이 가장 자주 읽게 될 것은 `diff`, 커밋 메시지, PR 설명과 리뷰 상태다.
-

Pull Request 리뷰 설명 포인트

- 초록색 Checks는 자동 검사를 통과했다는 뜻이지, 제품 요구사항이 모두 맞다는 뜻은 아니다.
- PM 리뷰는 “코드가 예쁘다”가 아니라 “제품의 의도와 출시 조건이 충족된다”에 초점을 둔다.
- 모르는 구현은 단정하지 말고 “이 변경이 주문 상태 전환에도 영향을 주나요?” 처럼 영향 질문으로 확인한다.
- 한 줄 코멘트는 맥락이 정확할 때 사용하고, 범위 전체의 문제는 PR 전체 리뷰로 남긴다.

진행 전 최종 체크

- 실습 저장소 접근 권한과 인증 방법을 확인했다.
- 참가자별 작성자 · 리뷰어 역할을 정했다.
- 명령어 실습이 막힐 때 사용할 GitHub 웹 대체 흐름을 준비했다.
- 수업 종료 시 회수할 결과물과 복습 질문을 선택했다.

현장 메모

A.6 공식 참고 자료

- [Pro Git: Getting a Git Repository](#)
- [Pro Git: Recording Changes to the Repository](#)
- [Pro Git: Branches in a Nutshell](#)
- [GitHub Docs: Hello World](#)
- [GitHub Docs: GitHub Flow](#)
- [GitHub Docs: Pull Requests](#)
- [GitHub Docs: About Issues](#)
- [GitHub Docs: Merge Conflicts](#)

자료 기준

이 통합본은 Git 공식 문서와 GitHub 공식 문서를 기준으로 작성했다. GitHub 화면의 메뉴 명칭은 서비스 업데이트에 따라 달라질 수 있다.

내 업무에 연결하기

1. 이제 내 말로 설명할 수 있는 개념

2. 다음 Pull Request에서 먼저 확인할 것

3. 우리 팀에 바로 적용할 협업 규칙
