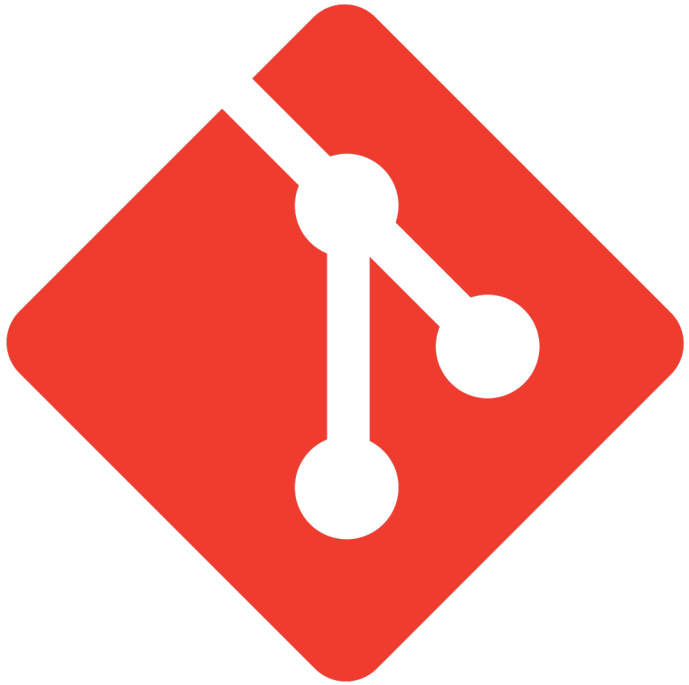
Lecture 12
Version control continued …

CS 5150, Spring 2026



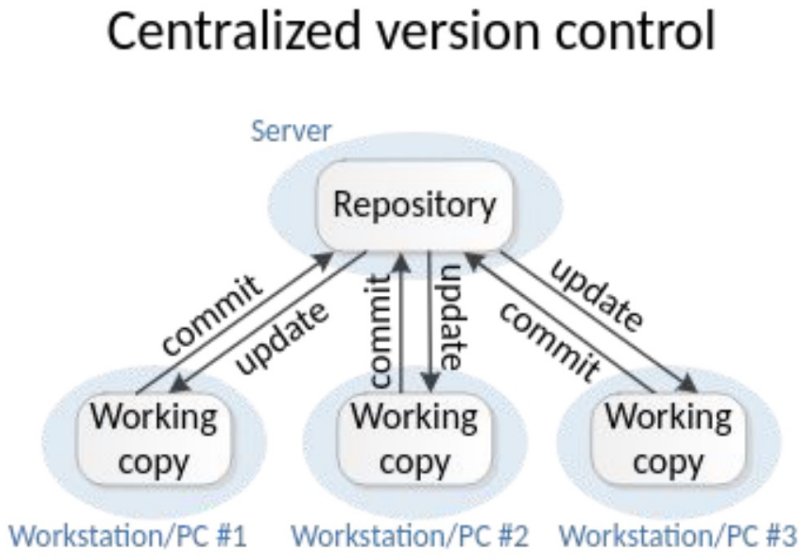
Administrative Reminders

- Exam 1: Mar 10. Syllabus up to Lecture 12 (Today, March 3)
- Midpoint Presentation: Mar 12/17
  - Sign up (if not signed up already)
- External talk: Mar 19 (See details on website)
- Quizzes (on canvas by today)
- Next class (Mar 5): Using AI tools for coding (instructions to follow)

Version Control Systems

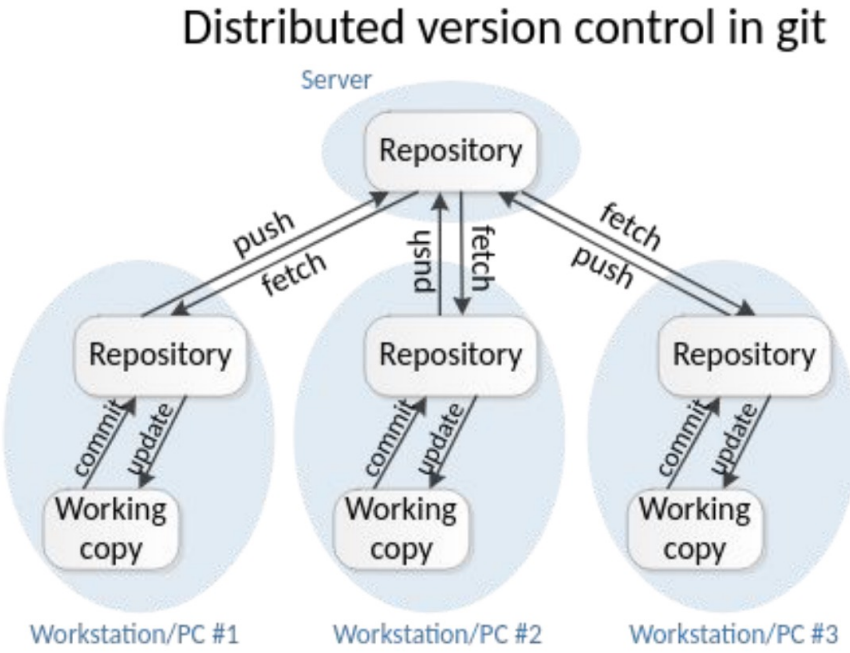
Centralized version control (the old way)

- One central repository
  - Stores a history of project versions
- Each user has a working copy
- A user commits file changes to the repository
- Committed changes are immediately visible to teammates who update
- Examples: SVN (Subversion), CVS
- Problems: Slow!, Single Point of Failure, Branching is expensive, …



Distributed version control (the new way)

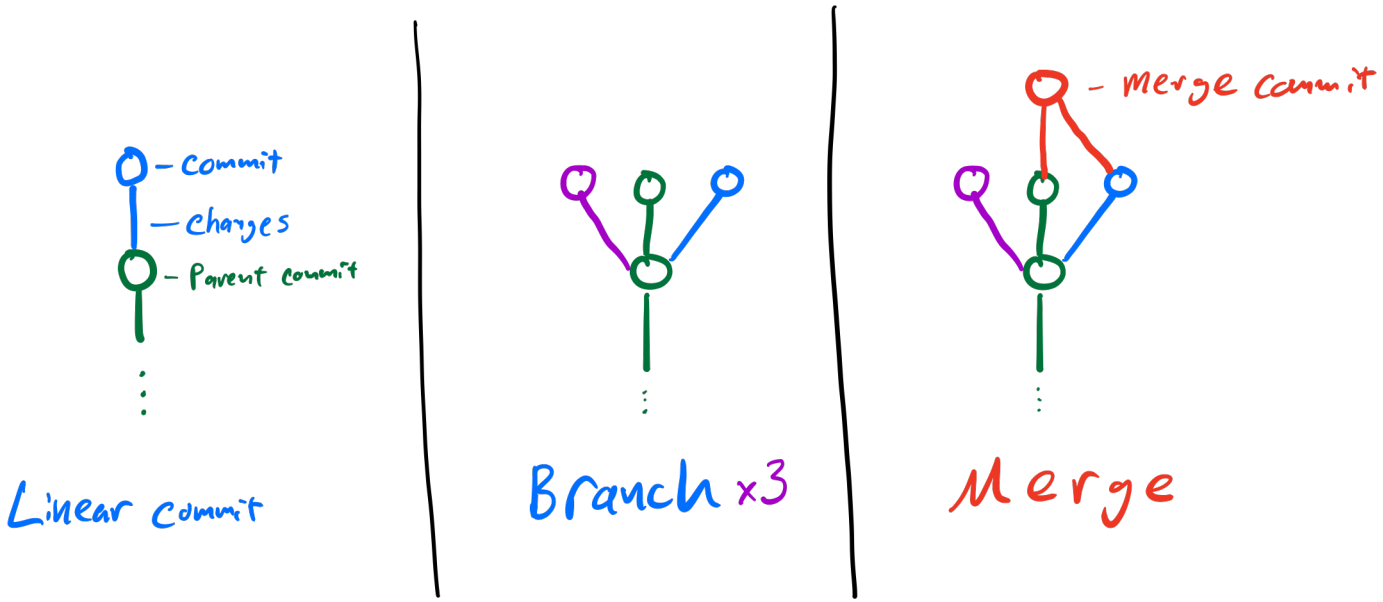
- Multiple copies of a repository. Each stores its own history of project versions.
- Each user commits to a local (private) repository
- All committed changes remain local unless pushed to another repository.
- No external changes are visible unless fetched from another repository.
- Examples: Git, Hg (mercurial)



Git VCS/SCM

- Created in 2005 by Linus Torvalds
  - Previous system (**bitkeeper**) took 30 seconds to apply a patch!
- Distributed VCS:
  - Geared for Speed, Data Integrity
  - Distributed non-linear workflows running 1000s of parallel branches on different machines
  - Lightweight branching (a branch is only reference to one commit)
- Maintains a “local” copy of entire repo on each machine
  - See “.git” folder
- Other VCS: SVN (centralized), Mercurial (slower), CVS

Commit diagrams



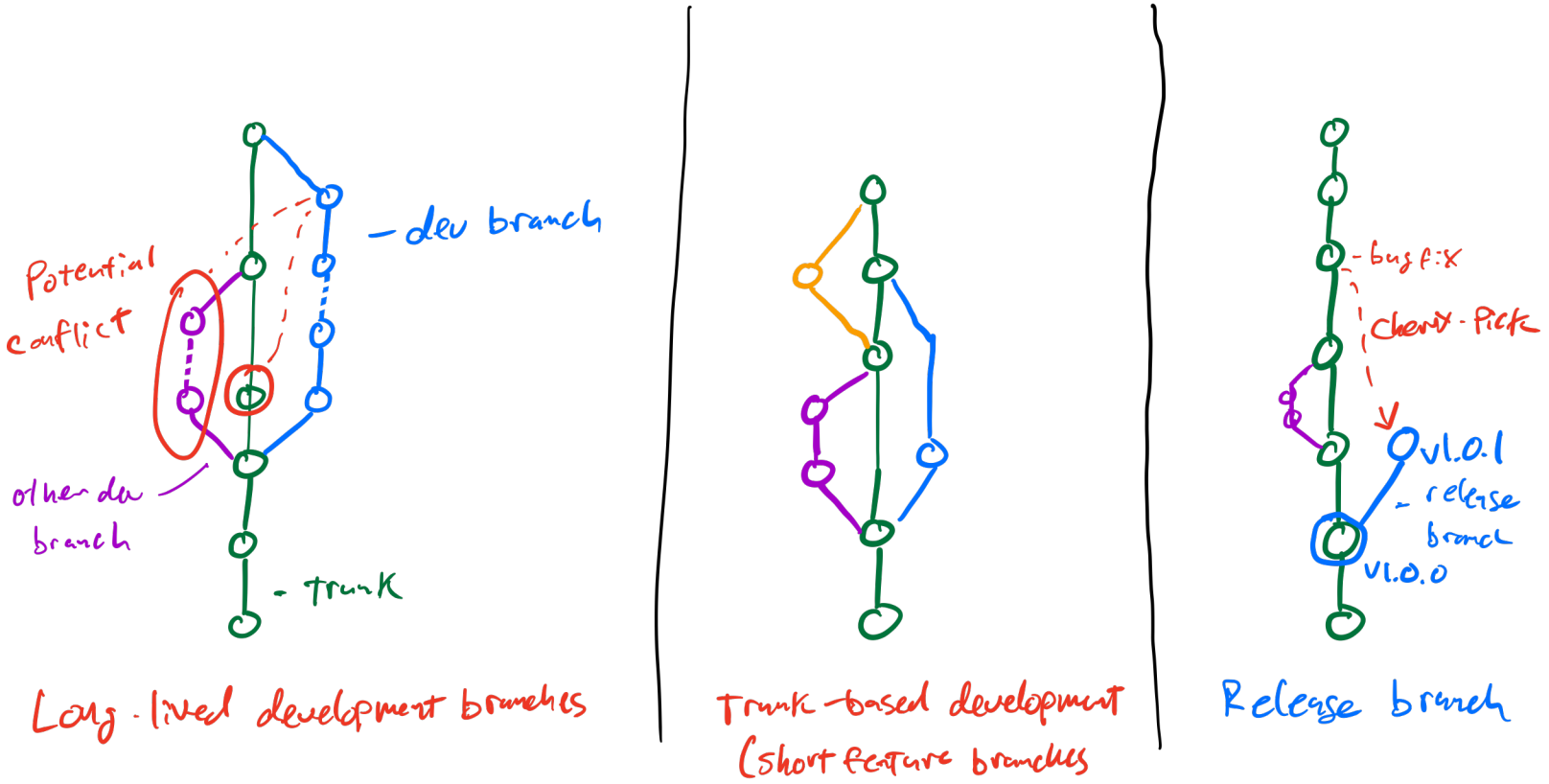
Interesting but important facts:

- Git does not store “files” or “diffs”, but snapshots of trees of blobs identified using hashes!
- Every object is identified by a cryptographic hash (or SHA)
- Immutability, integrity – history is tamper-proof!

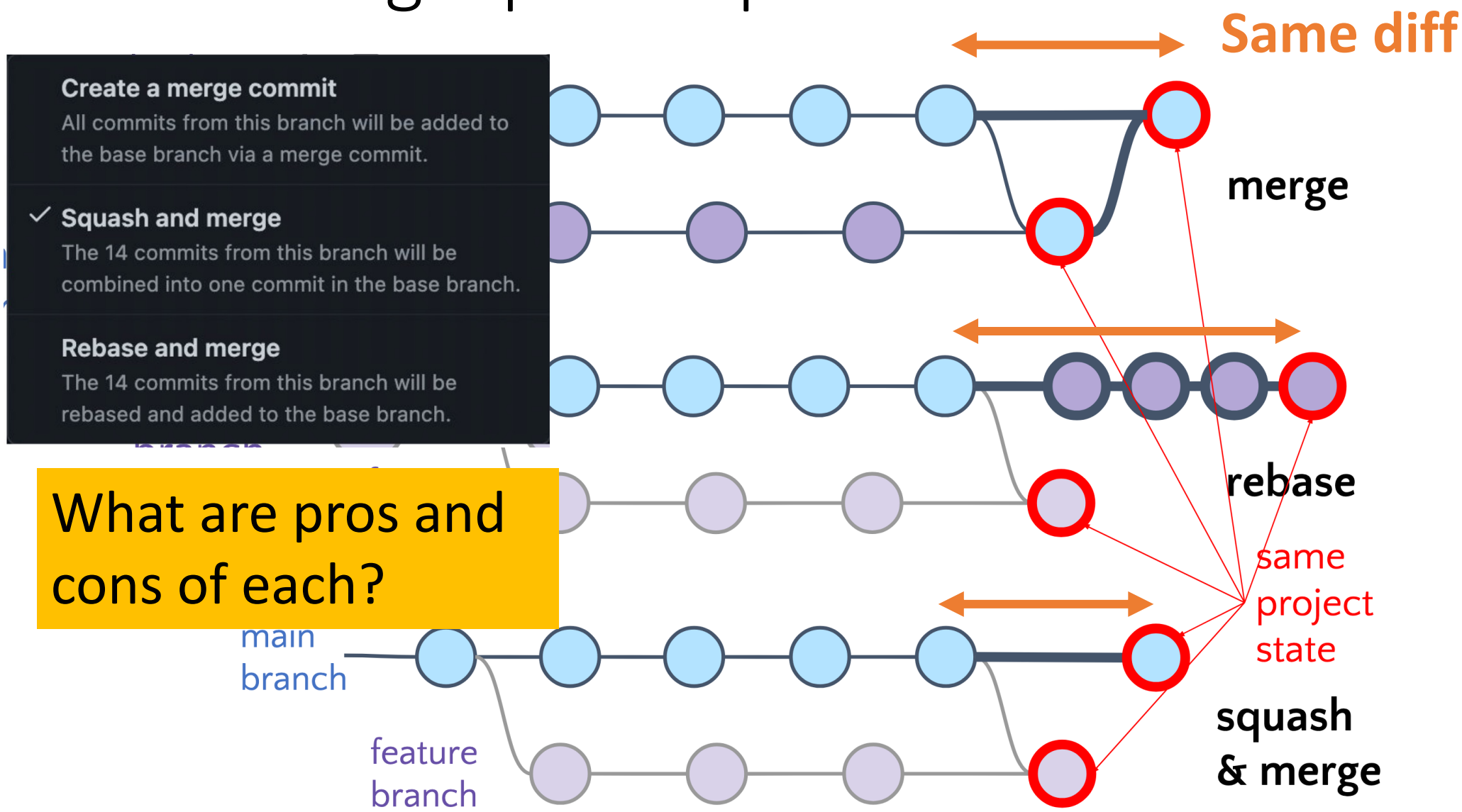
Branch management

- Trunk/master/main
  - Canonical latest version of "ready" code
  - Should be kept in buildable state
- Development branch
  - Long-lived branch for iterating on work in progress
  - Merged with trunk when "finished"
  - Problem: merges are painful; postponing them makes them more painful
  - Shift-left: "if it's painful, do it more often"
- Trunk-based workflow
  - Keep changes small (may queue in issue branch)
  - Merge immediately to trunk
  - Requires continuous testing
- Release branch
  - Tracks version of software released "in the wild" (think hardware products)
  - Provide stability
  - Cherry-pick bugfixes

Diagrams


Potential conflict
dev branch
other dev branch
trunk
Long-lived development branches
Trunk-based development (short feature branches)
bug fix
cherry-pick
v1.0.1
release branch
v1.0.0
Release branch

Resolving a pull request

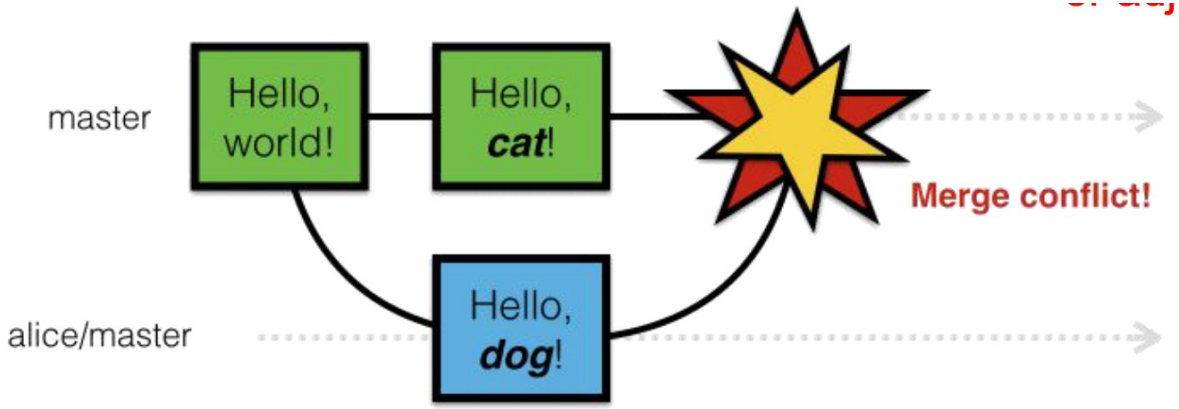


What are pros and cons of each?

Conflicts

Git's merge tools can make mistakes!

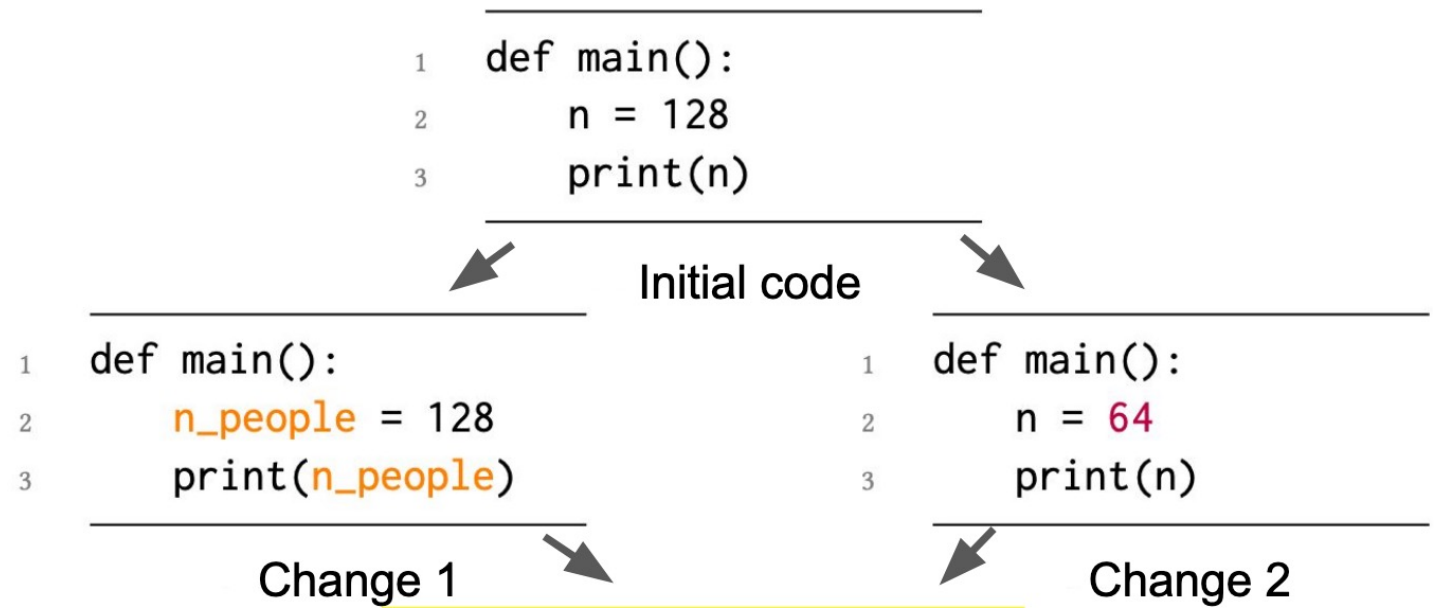
- When you run git merge, git attempts to retain all the changes from each branch
- A "conflict" arises when two users **change the same line (or adjacent lines)** of a file



- The person doing the merge needs to resolve the conflict by **manual editing**

Merge algorithm failure: unable to merge

- Line-by-line merge yields a conflict
- Inspection reveals they can be merged!

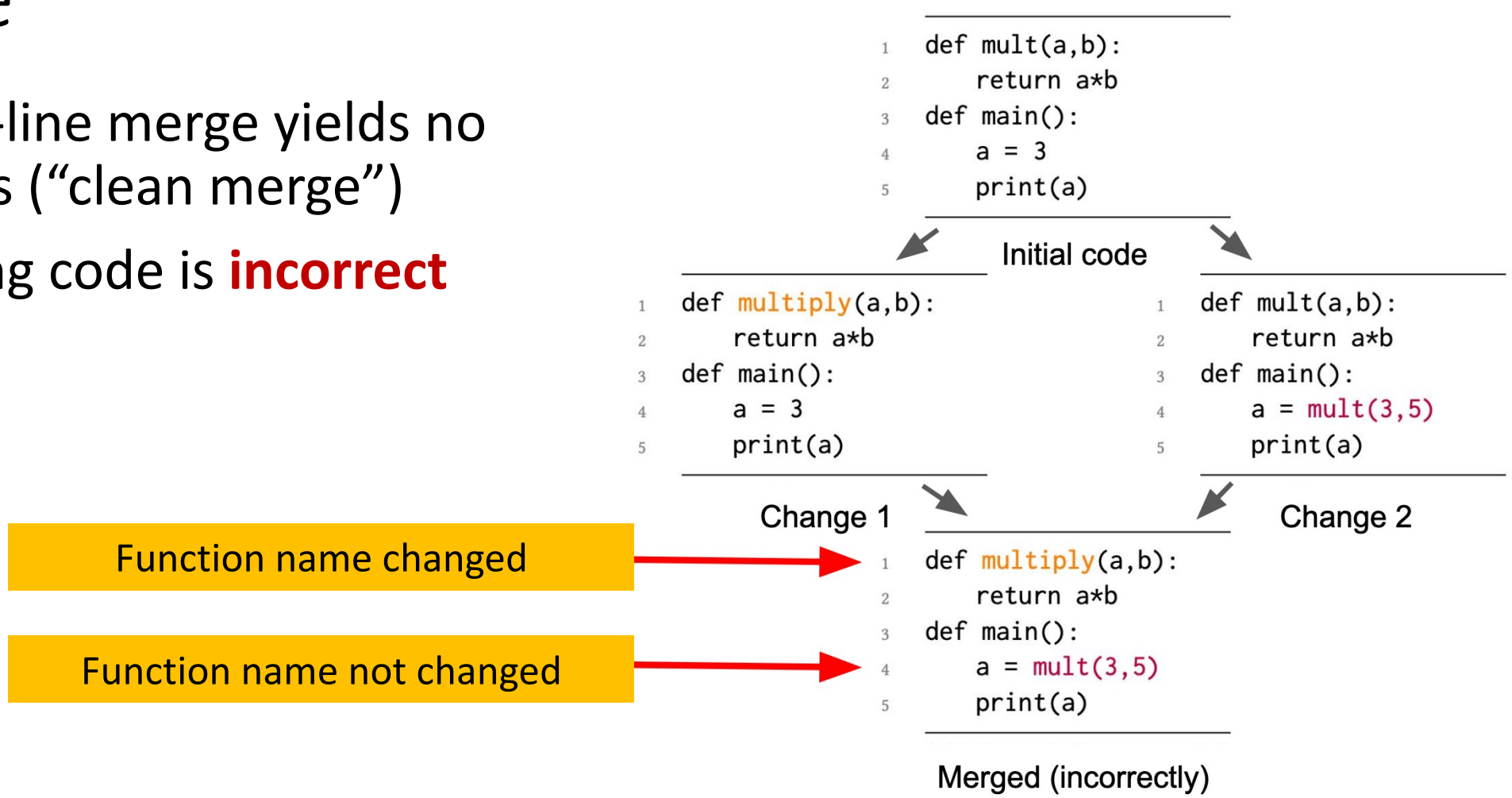


Works despite changes on same line

Git's output: "merge conflict"

Merge algorithm failure: clean, incorrect merge

- Line-by-line merge yields no conflicts (“clean merge”)
- Resulting code is **incorrect**



Wrong word substitution!

How to avoid merge conflicts?

- Sync with your teammates often
  - Pull often
  - Push as often as practical
    - Don’t destabilize the main branch (Don’t break the build!)
    - Use continuous integration: Automatic testing on each PR and push, even for branches
  - Avoid long-lived branches (make frequent, small pull requests)
- Commit often
  - Each commit should address one concept; tests should always pass
- Each merged branch should address one concern (Feature/fix)
- Don’t worry about commit history!
- Get changes in main via PR:
  - Squash and merge! What about rebase?

Make single-concern branches and commits

- They are easier to understand, review, merge, revert.
- Ways to achieve single-concern branches and commits:
  - Do only one task at a time: commit after each one
- Create a branch for each simulateneous task
  - Easier to share work with teammates
  - Single-concern branch ⇒ Single-concern commit on main
  - Requires a bit of bookkeeping to keep track of them all (but worth it)
  - Potential for merge conflicts
- Do multiple tasks in one branch ☹
  - Commit only specific files, or only specific parts of files (use staging)

Do not commit all files

Use a .gitignore file

Don’t commit:

- Binary files
- Log files
- Generated files
- Temporary files

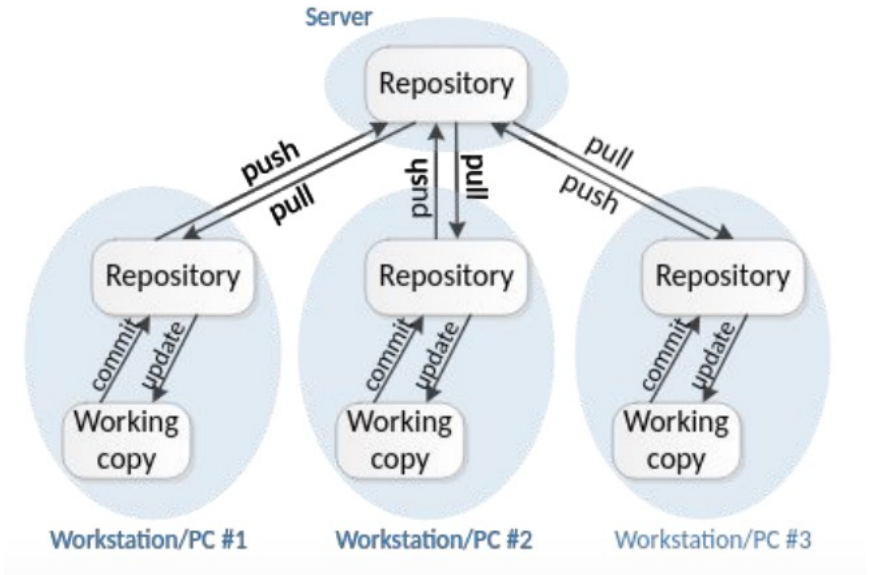
Committing would waste space and lead to merge conflicts

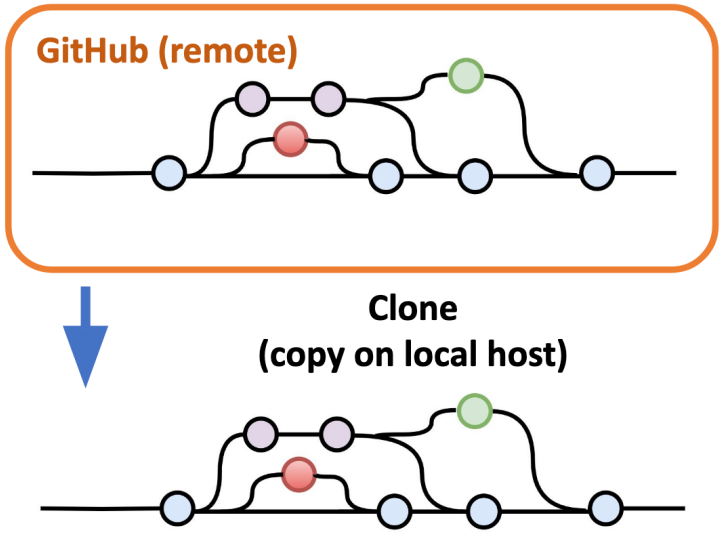
Plan ahead to avoid merge conflicts

- Modularize your work
  - Divide work so that individuals or subteams “own” parts of code
  - Other team members only need to understand its specification
  - Requires good documentation
- Communicate about changes that may conflict
  - Examples (rare!): reformat whole codebase, move directories, rename fundamental data structures

Cloning

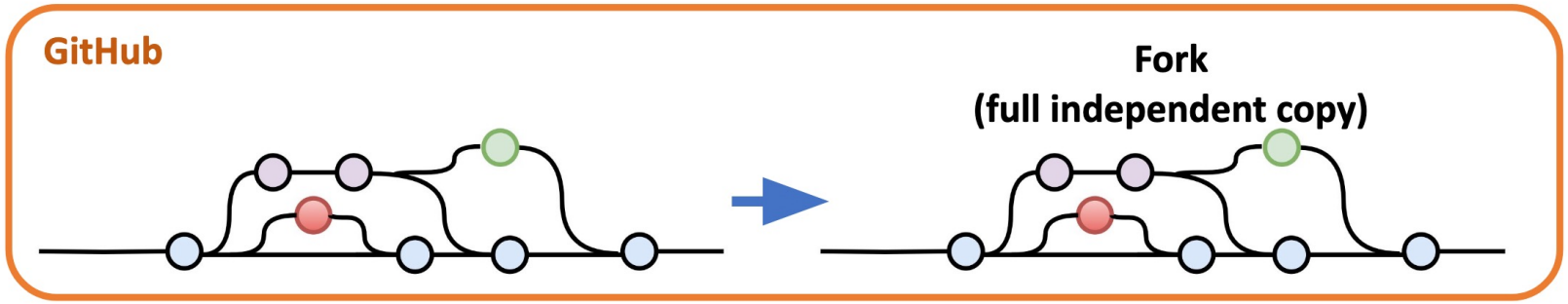
- **git clone** creates a local copy of the repo and a working copy of the files
- Ideal for contributing to a repo alongside other devs
- **git push** sends local changes to remote repo



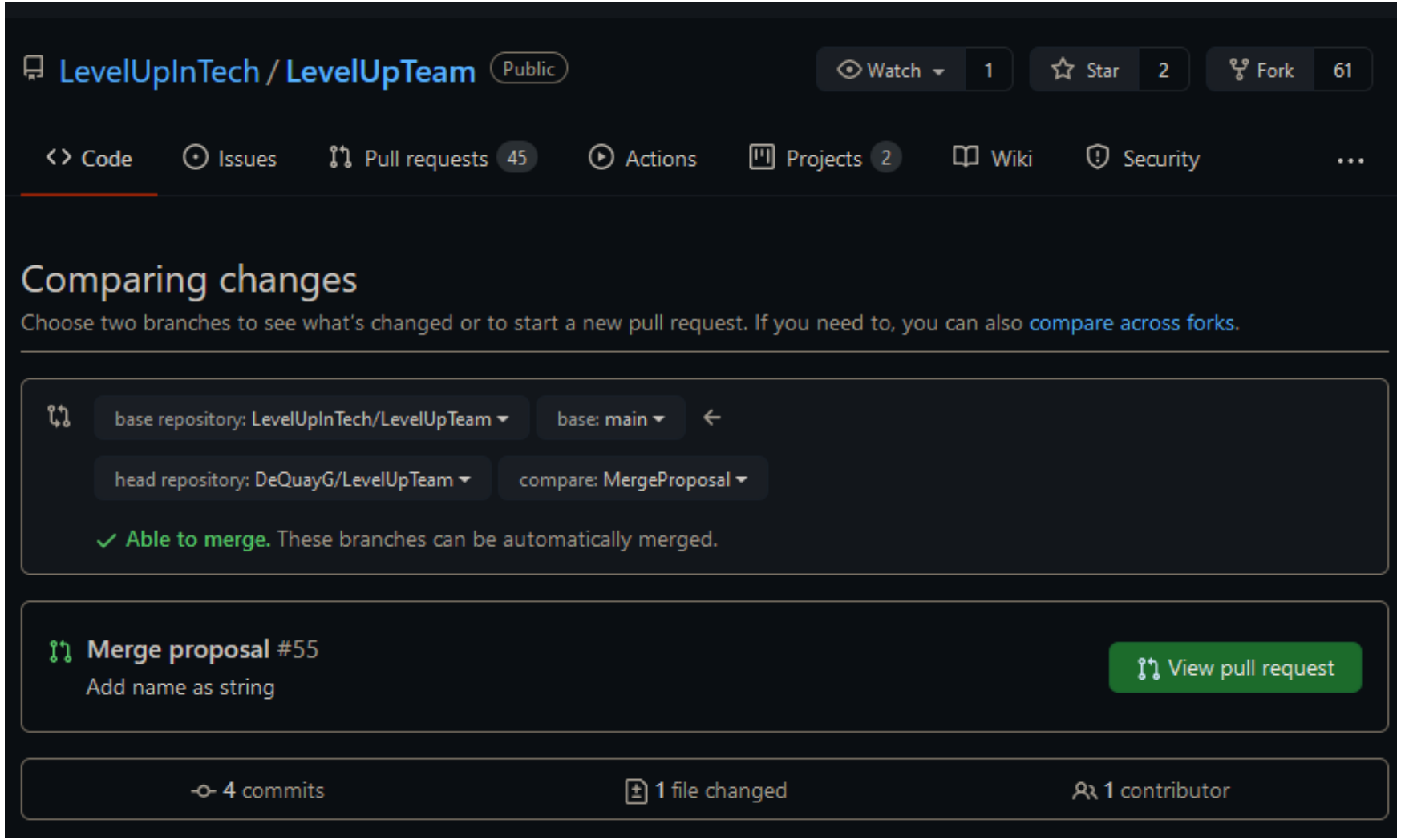


Forking (Github concept, not git)

- Creates a new, unrelated repository (GitHub project) that is initially an exact copy (including SHAs)
- Changes to either repository do not affect the other
- You can evolve the fork without impacting the upstream
- If the original repo is deleted, forked repo will exist
- Is it possible to update original?



Forking (Pushing changes to original repo)



Choose between branch, clone, or fork

1. Fix bugs in your 5150 project
2. Fix bugs in an open-source project not owned by you
3. Work on the **scrambler** project in the assignment locally
4. Start a new **XScrambler** project based on **scrambler** for 5150 SP 27
5. Edit an old existing branch on a repository you own

Git/VCS Terms You Should Know

- Blame
- Bisect
- Remote
- Branch
- Submodule
- Restore
- Checkout
- Tag
- Rebase
- Diff
- Cherry Pick
- Fetch
- Staging
- Revert/Reset
- Merge
- Log
- Amend
- HEAD
- SHA

Activity: Answer the following questions

1. What is the state of my changes when I do “git add” but don’t commit the changes?
2. How do I check who the last person was who edited a line of code in a file in my repo?
3. How do I change my working branch?
4. I want to merge my branch with the main branch. However, the main branch has newer changes. How do I ensure my changes are added as the latest commit(s)?
5. How do I undo the last commit without deleting any code?

Learn more!

- Other resources: explanations, tips, best practices
  - GitHub git cheat sheet: https://training.github.com/downloads/github-git-cheat-sheet/
  - Atlassian merge vs rebase (but don't rebase)
  - Git branching and merging
  - Video tutorial "Git, GitHub, & GitHub Desktop"
  - Learn Git Branching

Programming Styles/Conventions

Quiz: Is this code good or bad?

Quiz setup

- Project groups or small teams of neighboring students
- 6 code snippets (same for both rounds)

- Round 1:
  - For each code snippet, decide if it represents good or bad practice
  - Goal: discuss and reach consensus on good or bad practice
- Round 2:
  - For each code snippet, try to understand why it is good or bad practice
  - Goal: come up with an explanation or a counterargument

Snippet 1: good or bad?

```
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = … // determine number of log files
            File[] allLogs = new File[numLogs];
        for (int i = 0; i < numLogs; ++i) {
            allLogs[i] = dir.getLogFile(i); // populate the array
        }
        return allLogs;
    }
}
```

Snippet 2: good or bad?

```
public void addStudent(Student student, String course) {
    if (course.equals("CS5150")) {
        cs5150Students.add(student);
    }
    allStudents.add(student)
}
```

Snippet 3: Good or Bad?

```
public enum PaymentType { DEBIT, CREDIT }
public void doTransaction(double amount, PaymentType payType) {
    switch (payType) {
        case DEBIT:
            … // process debit card
            break;
        case CREDIT:
            … // process credit card
            break;
        default:
            throw new IllegalArgumentException("Unexpected payment type");
    }
}
```

Snippet 4: good or bad?

```
public int getAbsMax(int x, int y) {
    if (x < 0) {
        x = -x;
    }
    if (y < 0) {
        y = -y;
    }
    return Math.max(x, y);
}
```

Snippet 5: good or bad?

```
public class ArrayList < E > {
    public E remove(int index) {…}
    public boolean remove(Object o) {…}…
}
```

Snippet 6: good or bad?

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
}
```

Spoiler alert: Staff opinions

Snippet 1: bad

Snippet 2: bad

Snippet 3: good

Snippet 4: bad

Snippet 5: bad

Snippet 6: good

Snippet 1: good or bad?

```
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = … // determine number of log files
            File[] allLogs = new File[numLogs];
        for (int i = 0; i < numLogs; ++i) {
            allLogs[i] = dir.getLogFile(i); // populate the array
        }
        return allLogs;
    }
}
```

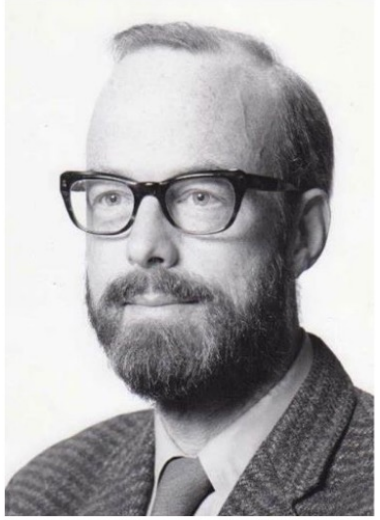
Snippet 1: Bad!

```
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = … // determine number of log files
            File[] allLogs = new File[numLogs];
        for (int i = 0; i < numLogs; ++i) {
            allLogs[i] = dir.getLogFile(i); // populate the array
        }
        return allLogs;
    }
}
```

Null references: The billion dollar mistake!

Apologies and retractions

Speaking at a software conference named QCon London[24] in 2009, he apologised for inventing the null reference:[25]



Tony Hoare

- Programming languages
- Concurrent programming
- Creator of **quicksort**

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Snippet 1: Bad!

```
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = … // determine number of log files
            File[] allLogs = new File[numLogs];
        for (int i = 0; i < numLogs; ++i) {
            allLogs[i] = dir.getLogFile(i); // populate the array
        }
        return allLogs;
    }
}
```

Don’t return null; return empty array instead!

Snippet 1: Bad!

```
public File[] getAllLogs(Directory dir) {
    if (dir == null || !dir.exists() || dir.isEmpty()) {
        return null;
    } else {
        int numLogs = … // determine number of log files
            File[] allLogs = new File[numLogs];
        for (int i = 0; i < numLogs; ++i) {
            allLogs[i] = dir.getLogFile(i); // populate the array
        }
        return allLogs;
    }
}
```

Don’t return null; return empty array instead!

No diagnostic info!

Snippet 2: good or bad?

```
public void addStudent(Student student, String course) {
    if (course.equals("CS5150")) {
        cs5150Students.add(student);
    }
    allStudents.add(student)
}
```

Snippet 2: short but bad!

```
public void addStudent(Student student, String course) {
  if (course.equals("CS5150")) {
      cs5150Students.add(student);
   }
   allStudents.add(student)
}
```

Defensive programming: add an assertion (or write the literal first).
Use constants and enums to avoid literal duplication.
Return success or failure!

Snippet 3: Good or Bad?

```
public enum PaymentType { DEBIT, CREDIT }
public void doTransaction(double amount, PaymentType payType) {
    switch (payType) {
        case DEBIT:
            … // process debit card
            break;
        case CREDIT:
            … // process credit card
            break;
        default:
            throw new IllegalArgumentException("Unexpected payment type");
    }
}
```

Snippet 3: This is good but why?

```
public enum PaymentType { DEBIT, CREDIT }
public void doTransaction(double amount, PaymentType payType) {
    switch (payType) {
        case DEBIT:
            … // process debit card
            break;
        case CREDIT:
            … // process credit card
            break;
        default:
            throw new IllegalArgumentException("Unexpected payment type");
    }
}
```

Type safety using enums; throws an exception for unexpected cases (e.g., future extensions of PaymentType)

Snippet 4: good or bad?

```
public int getAbsMax(int x, int y) {
    if (x < 0) {
        x = -x;
    }
    if (y < 0) {
        y = -y;
    }
    return Math.max(x, y);
}
```

Snippet 4: also bad?!

```
public int getAbsMax(int x, int y) {
    if (x < 0) {
        x = -x;
    }
    if (y < 0) {
        y = -y;
    }
    return Math.max(x, y);
}
```

Avoid reassigning method parameters;
use local variables to sanitize inputs.
(Making parameters final somewhat achieves this.)

Snippet 5: good or bad?

```
public class ArrayList <E> {
    public E remove(int index) {…}
    public boolean remove(Object o) {…}…
}
```

Snippet 5: Java API but still bad, why?

```
public class ArrayList <E> {
    public E remove(int index) {…}
    public boolean remove(Object o) {…}…
}
```

```
ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add(“Hello”);
l.add(“World”);
l.remove(index);
```

What does the last call return?
(l.remove(index))?

Snippet 5: Java API but still bad, why?

```
public class ArrayList <E> {
    public E remove(int index) {…}
    public boolean remove(Object o) {…}…
}
```

```
ArrayList<String> l = new ArrayList<>();
Integer index = Integer.valueOf(1);
l.add(“Hello”);
l.add(“World”);
l.remove(index);
```

What does the last call return? (l.remove(index))?

Be careful with method overloading, which is statically resolved.

Hesitate to use overloading and different return values

Snippet 6: good or bad?

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
}
```

Snippet 6: this is good but why?

```
public class Point {
    private final int x;
    private final int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return this.x;
    }
    public int getY() {
        return this.y;
    }
}
```

Good encapsulation; immutable object