CS61B: 2021

Lecture 12: Command Line Programming and Data Structures Preview

- public static void main(String[] args)
- Command line compilation and execution
- Git Case Study: Maps, Hashing, Serializable

Random 61C Preview

Decimal, Binary, Hexadecimal

Before we start today, I'd like to cover a concept you'll go over in 61C.

- Will be useful later today.

In the decimal number system, we have digits 0123456789.

- Numbers larger than 1 are represented by a sequence, e.g. 932.

In the binary number system, we have digits 01.

- Numbers larger than 1 are represented by a sequence, e.g. 1110100100.

In the hexadecimal number system, we have digits 0123456789abcdef.

- Numbers larger than f are represented by a sequence, e.g. 3a4.

Decimal, Binary, Hexadecimal

The numbers 932, 1110100100, and 3a4 are all the same exact number.

Hexadecimal is often used in computer science in lieu of decimal.

- The exact reasons I'll leave for 61C.
- Key property of hexadecimal: Number of digits is a power of 2.

Today, we'll use hexadecimal to represent large numbers.

- Binary:
  1100110110011001101110001100100010111001001110100010101011011010101110001111001011011011011111001101110110101110110100001000001100001100000101010110001010100010
- Hexadecimal: 66ccdc645c9d156d5c796dbe6ed768430c1562a2

Command Line Compilation

Compilation

The standard tools for executing Java programs use a two step process:

- This is not the only way to run Java code.

In our course so far we've been using Intellij, which uses javac and java.

- IntelliJ hides this two step process from us.

However, it is also possible to manually invoke javac and java ourselves.

public static void main(String[] args)

One Special Role for Strings: Command Line Arguments

```
public class ArgsDemo {
    /** Prints out the 0th command line argument. */
    public static void main(String[] args) {
        System.out.println(args[0]);
    }
}
```

```
jug ~/Dropbox/61b/lec/gitletIntro
$ java ArgsDemo hello some args
hello
```

ArgsSum Exercise

Goal: Create a program ArgsSum that prints out the sum of the command line arguments, assuming they are numbers.

- Search engines are our friend!

One Special Role for Strings: Command Line Arguments

```
public class ArgsSum {
    /** Prints out the sum of arguments, assuming they are
    *  integers.
    */
    public static void main(String[] args) {
        int index = 0;
        int sum = 0;
        while (index < args.length) {
            sum = sum + Integer.parseInt(args[index]);
            index = index + 1;
        }
        System.out.println(sum);
    }
}
```

How'd we know to do this? We Googled "convert string integer java".

```
$ java ArgsSum 1 2 3 4
10
```

Git: A Command Line Program

Git: A Command Line Tool

The git tool we've been using is a command line program.

- Written in C.
- Unlike Java, C code is typically compiled into a binary which doesn't require an interpreter. See CS61C for more.
  - Thus, instead of saying "java git status", we just type "git status".

```
jug ~/Dropbox/61b/lec/git
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
        modified:   HelloWorld.java
```

Git Source Code

Git is just a program.

- Source code for Git's main method is to the right or at this link.
- In the C programming language, `const char **argv` is the equivalent of Java's `String[] args`.

```
26  int main(int argc, const char **argv)
27  {
28          int result;
29
30          trace2_initialize_clock();
31
32          /*
33           * Always open file descriptors 0/1/2 to avoid clobbering files
34           * in die().  It also avoids messing up when the pipes are dup'ed
35           * onto stdin/stdout/stderr in the child processes we spawn.
36           */
37          sanitize_stdfds();
38          restore_sigpipe_to_default();
39
40          git_resolve_executable_dir(argv[0]);
41
42          git_setup_gettext();
43
44          initialize_the_repository();
45
46          attr_start();
47
48          trace2_initialize();
49          trace2_cmd_start(argv);
50          trace2_collect_process_info(TRACE2_PROCESS_INFO_STARTUP);
51
52          result = cmd_main(argc, argv);
53
54          trace2_cmd_exit(result);
55
56          return result;
57  }
```

Git Source Code

Git is just a program.

- argv gets passed to another function called cmd_main. First few lines shown to the right or at this link.
- Even though we don't know C, we can infer from lines 850 and 851 that if the user doesn't enter any command line arguments, git will assume the user wants help.

```
844  int cmd_main(int argc, const char **argv)
845  {
846          const char *cmd;
847          int done_help = 0;
848
849          cmd = argv[0];
850          if (!cmd)
851                  cmd = "git-help";
852          else {
853                  const char *slash = find_last_dir_sep(cmd);
854                  if (slash)
855                          cmd = slash + 1;
856          }
```

Git Source Code

Git is just a program.

- argv gets passed to another function called cmd_main. First few lines shown to the right or at this link.
- Even though we don't know C, we can infer from lines 850 and 851 that if the user doesn't enter any command line arguments, git will assume the user wants help.

```
844  int cmd_main(int argc, const char **argv)
845  {
846          const char *cmd;
847          int done_help = 0;
848
849          cmd = argv[0];
850          if (!cmd)
851                  cmd = "git-help";
852          else {
853                  const char *slash = find_last_dir_sep(cmd);
854                  if (slash)
855                          cmd = slash + 1;
856          }
```

Why are we talking about this?

- To show that we've already been using command line arguments in class.
- In project 2, you are going to build your own implementation of git.

Git

Git is a sophisticated piece of software. Relies on many ideas we have not yet covered:

- Maps.
- Hashing.
- File I/O.
- Graphs.

Today, we'll get a preview of the first three of these things, along with some insight into how git works.

Basic Git Functionality

Why Version Control?

Software development is an iterative process.

Maintaining multiple copies is useful:

- When working on projects with others, want to ensure we don't damage other's work.
- When working alone, we sometimes want to make some (possibly complex) tentative set of changes.

Naive approach: Store a bunch of old versions in multiple directories, e.g. 2048UpFinallyWorks, 2048MergeBugFixed, etc.

The Naive Approach

The good:

- Very easy! Just have to know how to copy files.

Issues:

- Wasteful with storage (many near identical programs).
- Requires caution to avoid erroneous restoration / storage.
- Requires self-discipline to get good backup coverage.
  - Going 3 days without making a copy might mean losing 3 days of work.
- Merging two divergent copies requires careful manual work.
  - A bit beyond the scope of what we've done in 61B.

Version Control Software

There are many software packages out there that handle version control. Some popular systems:

- Git (2005, open source)
- Perforce (1995)
- SVN (2000, open source)
- Mercurial (2005, open source)

These days, git is the most popular overall.

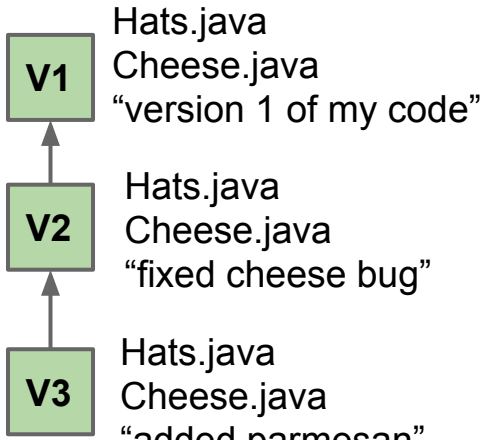
Today we'll talk a little bit about how git works.

- Utilizes lots of ideas from later in the course.
- You'll be implementing git in project 2.

Git: How it Works

Every time you commit changes to a file, it stores a copy of the **entire project** in a secret folder on your computer called .git.

```
$ subl Hats.java
$ subl Cheese.java
$ git add .; git commit -m “version 1 of my code”
$ subl Cheese.java
$ git add .; git commit -m “fixed cheese bug”
$ git add .; git commit -m “added parmesan”
```



Let’s try this out.

Git: How it Works

Every time you commit changes to a file, it stores a copy of the **entire repository** in a secret folder on your computer called .git.

```
$ subl Hats.java
$ subl Cheese.java
$ git add .; git commit -m “version 1 of my code”
$ subl Cheese.java
$ git add .; git commit -m “fixed cheese bug”
$ subl Cheese.java
$ git add .; git commit -m “added parmesan”
```

V1
Hats.java
Cheese.java
“version 1 of my code”

V2
Hats.java
Cheese.java
“fixed cheese bug”

V3
Hats.java
Cheese.java
“added parmesan”

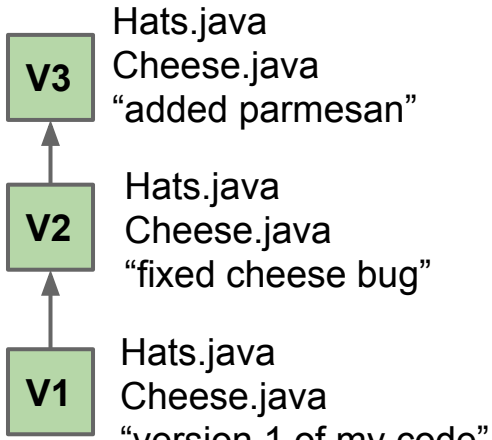
Various tricks are employed to avoid redundancy.

- Example: Don’t need three different copies of Hats.java

Git log

Can view history using the `git log` command:

```
$ git log --graph
* commit 7e41ce1a924ca616bded92bf5a4d0d899bdd6ca0
| Author: JoshHug <hug@cs.berkeley.edu>
| Date:   Tue Dec 2 23:00:53 2014 -0800
|
|     added parmesan
|
* commit aa45fbd68235e21393f27af44098c9b487345cdb
| Author: JoshHug <hug@cs.berkeley.edu>
| Date:   Tue Dec 2 23:00:35 2014 -0800
|
|     fixed cheese bug
|
* commit d1bde19ffd43a14ea959585df3fd0722c8aa0c61
  Author: JoshHug <hug@cs.berkeley.edu>
  Date:   Tue Dec 2 22:59:56 2014 -0800

      version 1 of my code
```

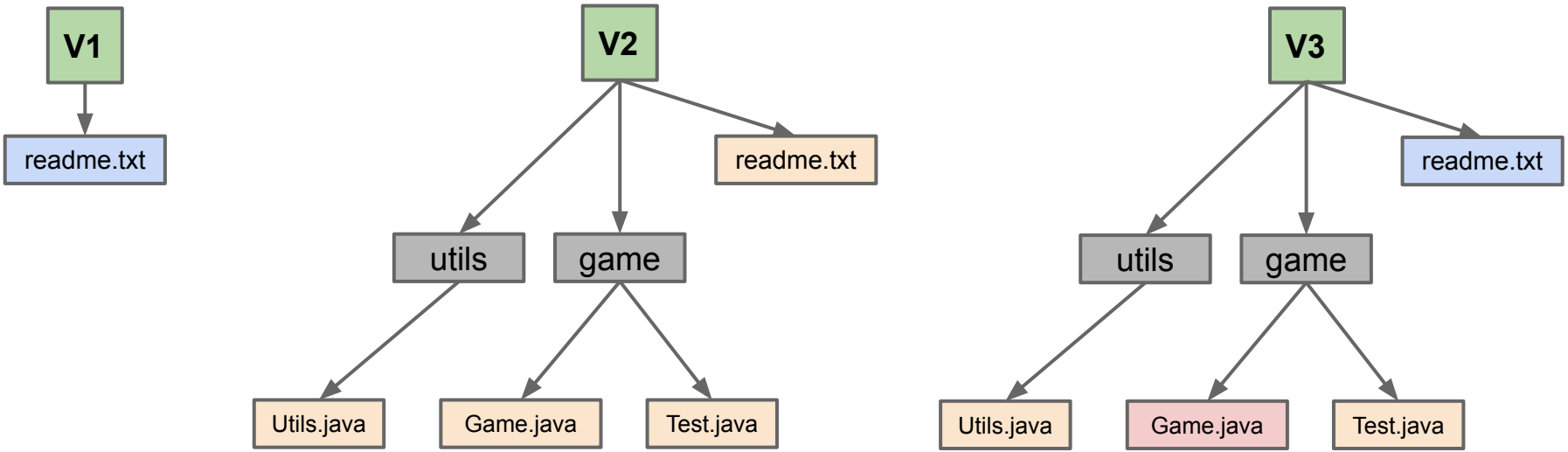


Avoiding Redundancy

Consider the Following Scenario

Suppose a programmer makes 3 commits while working on a Java project.

- In V1: Create readme.txt
- In V2: Create utils/Utils.java, game/Game.java, game/Test.java. Modify readme.
- In V3: Modify game/Game.java, change readme.txt back to V1 version.



Approach 1: Store Multiple Copies of Everything

As noted before, every time you commit changes to a file, it stores a copy of the **entire project** in the .git folder as a new commit.

Naive approach: Each commit is stored in a subdirectory with copies of every file.

```
.git/v1/readme.txt

.git/v2/readme.txt
.git/v2/utils/Utils.java
.git/v2/game/Game.java
.git/v2/game/Test.java

.git/v3/readme.txt
.git/v3/utils/Utils.java
.git/v3/game/Game.java
.git/v3/game/Test.java
```

Easy to implement!

- Commit simply creates a new subdirectory then copies all added files to the subdirectory.
- Checkout simply deletes everything in the current folder and copies all files from the requested subdirectory in their place.

Approach 1: Store Multiple Copies of Everything

As noted before, every time you commit changes to a file, it stores a copy of the **entire project** in the .git folder as a new commit.

Naive approach: Each commit is stored in a subdirectory with copies of every file.

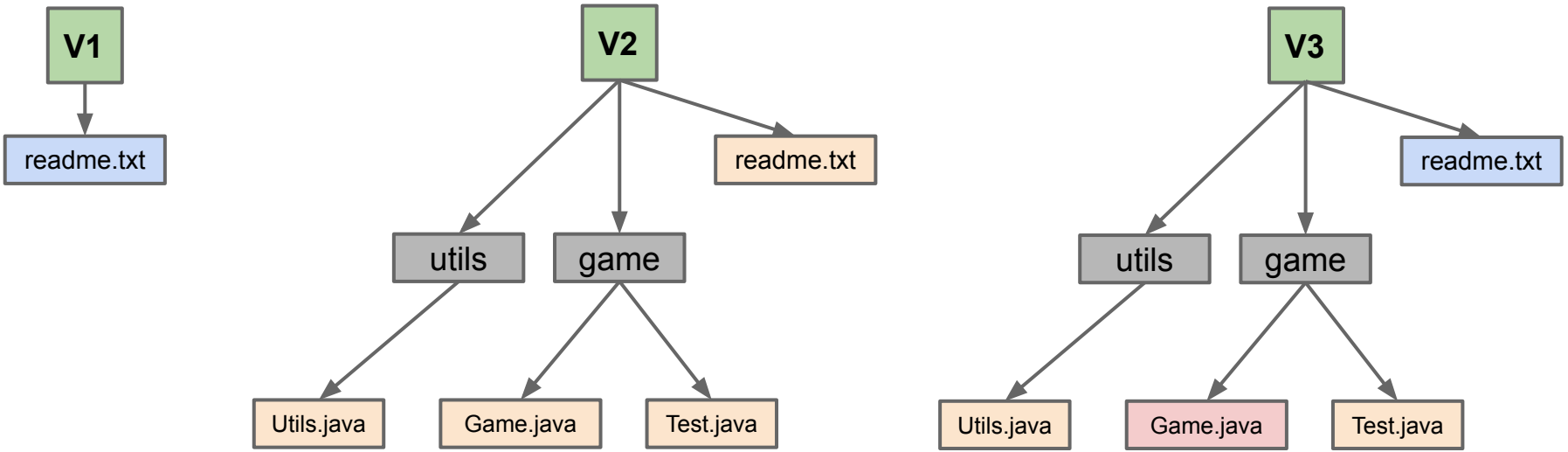
```
.git/v1/readme.txt

.git/v2/readme.txt
.git/v2/utils/Utils.java
.git/v2/game/Game.java
.git/v2/game/Test.java

.git/v3/readme.txt
.git/v3/utils/Utils.java
.git/v3/game/Game.java
.git/v3/game/Test.java
```

Naive approach is very inefficient. Here:

- 2 identical Utils.java files.
- 2 identical Test.java files.
- 2 identical readme.txt files.

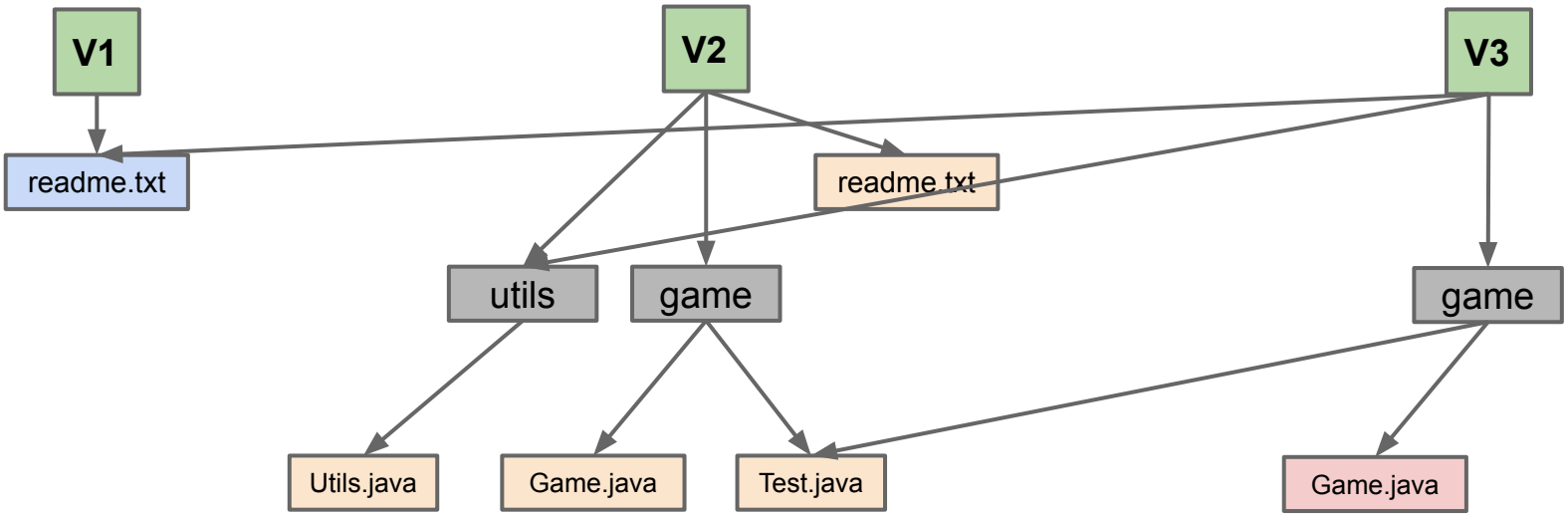
Eliminating Inefficiency

One obvious improvement: Don’t store multiple copies of the same file.



Approach 2: Store Only Files That Change

One obvious improvement: Don’t store multiple copies of the same file.



Approach 2: Store Only Files That Change

In revised approach 2, we only store files that change.

- Much more efficient. Avoids storing redundant files.
- Checkout is now more complicated. If we checkout a commit, we have to copy files from a variety of different folders.

```
.git/v1/readme.txt

.git/v2/readme.txt
.git/v2/utils/Utils.java
.git/v2/game/Game.java
.git/v2/game/Test.java

.git/v3/game/Game.java
```

Test Your Understanding

Suppose we have the commits for versions 1 through 5 stored in the folder below. If we check out commit version 4, which files will we use?

```
.git/v1/Hello.java

.git/v2/Hello.java
.git/v2/Friend.java

.git/v3/Friend.java
.git/v3/Egg.java

.git/v4/Friend.java

.git/v5/Hello.java
```

Test Your Understanding

Suppose we have the commits for versions 1 through 5 stored in the folder below. If we check out commit version 4, which files will we use?

```
.git/v1/Hello.java

.git/v2/Hello.java
.git/v2/Friend.java

.git/v3/Friend.java
.git/v3/Egg.java

.git/v4/Friend.java

.git/v5/Hello.java
```

To figure out which files to copy, we had to walk through the entire commit history starting from commit 1.

- Inefficient.

V4: Hello.java → v2, Friend.java → v4, Egg.java → v3

Approach 3: Approach 2 but with Version Data Structure

Better approach: Rather than walking through commits from the beginning, explicitly store a list of “commits”, where each commit tells us the filename and version number for the files in that commit.

```
.git/v1/Hello.java

.git/v2/Hello.java
.git/v2/Friend.java

.git/v3/Friend.java
.git/v3/Egg.java

.git/v4/Friend.java

.git/v5/Hello.java
```

V1: Hello.java → v1
V2: Hello.java → v2, Friend.java → v2
V3: Hello.java → v2, Friend.java → v2, Egg.java → v3
V4: Hello.java → v2, Friend.java → v4, Egg.java → v3
V5: Hello.java → v5, Friend.java → v4, Egg.java → v3

Note: Each commit is a “map” or “dictionary”. For each filename, it maps that filename to a version number.

Example: V4 in Python might be represented by {“Hello.java”: 2, “Friend.java”: 4, “Egg.java”: 3}

Test Your Understanding

Suppose we have the committed files for versions 1 through 5 stored in the folders on the left, and also have the list of commits on the right. Which files do we copy if we check out version 4?

```
.git/v1/X.java
.git/v1/Y.java

.git/v2/Y.java

.git/v3/Z.java

.git/v4/X.java
.git/v4/A.java

.git/v5/X.java
.git/v5/Y.java
```

V1: X.java → v1, Y.java → v1
V2: X.java → v1, Y.java → v2
V3: X.java → v1, Y.java → v2, Z.java → v3
V4: X.java → v4, Y.java → v2, Z.java → v3, A.java → v4
V5: X.java → v5, Y.java → v5, Z.java → v3, A.java → v4

Test Your Understanding

Suppose we have the committed files for versions 1 through 5 stored in the folders on the left, and also have the list of commits on the right. Which files do we copy if we check out version 4?

```
.git/v1/X.java
.git/v1/Y.java

.git/v2/Y.java

.git/v3/Z.java

.git/v4/X.java
.git/v4/A.java

.git/v5/X.java
.git/v5/Y.java
```

V1: X.java → v1, Y.java → v1
V2: X.java → v1, Y.java → v2
V3: X.java → v1, Y.java → v2, Z.java → v3
V4: X.java → v4, Y.java → v2, Z.java → v3, A.java → v4
V5: X.java → v5, Y.java → v5, Z.java → v3, A.java → v4

Another Advantage of Approach 3

Approach 3 also allows us to avoid even more redundancy.

- Example, suppose v5's X.java is the same as v1.

```
.git/v1/X.java
.git/v1/Y.java

.git/v2/Y.java

.git/v3/Z.java

.git/v4/X.java
.git/v4/A.java

.git/v5/X.java
.git/v5/Y.java
```

V1: X.java → v1, Y.java → v1
V2: X.java → v1, Y.java → v2
V3: X.java → v1, Y.java → v2, Z.java → v3
V4: X.java → v4, Y.java → v2, Z.java → v3, A.java → v4
V5: X.java → v5, Y.java → v5, Z.java → v3, A.java → v4

Another Advantage of Approach 3

Approach 3 also allows us to avoid even more redundancy.

- Example, suppose v5's X.java is the same as v1. How would we change the file structure on the left and the list of commits on the right?

```
.git/v1/X.java
.git/v1/Y.java

.git/v2/Y.java

.git/v3/Z.java

.git/v4/X.java
.git/v4/A.java

.git/v5/X.java
.git/v5/Y.java
```

V1: X.java → v1, Y.java → v1
V2: X.java → v1, Y.java → v2
V3: X.java → v1, Y.java → v2, Z.java → v3
V4: X.java → v4, Y.java → v2, Z.java → v3, A.java → v4
V5: X.java → v5, Y.java → v5, Z.java → v3, A.java → v4

Another Advantage of Approach 3

Approach 3 also allows us to avoid even more redundancy.

- Example, suppose v5's X.java is the same as v1. How would we change the file structure on the left and the list of commits on the right?

```
.git/v1/X.java
.git/v1/Y.java

.git/v2/Y.java

.git/v3/Z.java

.git/v4/X.java
.git/v4/A.java

.git/v5/Y.java
```

V1: X.java → v1, Y.java → v1
V2: X.java → v1, Y.java → v2
V3: X.java → v1, Y.java → v2, Z.java → v3
V4: X.java → v4, Y.java → v2, Z.java → v3, A.java → v4
V5: X.java → v1, Y.java → v5, Z.java → v3, A.java → v4

Rather than store v5/X.java, our commit data structure specifics that v5's X.java is the same as v1's.

Avoiding Redundancy with “Hashing”

Thought Experiment

Suppose we have two different programmers working on the same project.

- They both start at V3.
- Programmer A changes Horse.java and commits.
- Programmer B changes Fish.java and commits.
- Who gets to be V4?

Thought Experiment

Suppose we have two different programmers working on the same project.

- They both start at V3.
- Programmer A changes Horse.java and commits.
- Programmer B changes Fish.java and commits.
- Who gets to be V4?
  - Why can’t it just be whoever committed first?

Thought Experiment

Suppose we have two different programmers working on the same project.

- They both start at V3.
- Programmer A changes Horse.java and commits.
- Programmer B changes Fish.java and commits.
- Who gets to be V4?
  - Why can't it just be whoever committed first?

Git is a distributed version control system. Everything is done locally, and there is no central server that stores everything.

- Suppose B were first. Programmer A might be on a ship in the middle of the Pacific when Programmer B commits Fish.java.
- Programmer A's computer will have no idea this commit has been made.

Approach 4: Use Time and Date as the Version Number

Rather than using an escalating integer version number, we could use the current time and date.

```
.git/02_16_2021_03_29_45/X.java
.git/02_16_2021_03_29_45/Y.java

.git/02_16_2021_11_29_45/Y.java
.git/02_16_2021_11_29_45/Z.java

.git/02_16_2021_13_29_45/X.java
```

V02_16_2021_03_29_45:

- X.java → 02_16_2021_03_29_45
- Y.java → 02_16_2021_03_29_45

V02_16_2021_11_29_45:

- X.java → 02_16_2021_03_29_45
- Y.java → 02_16_2021_11_29_45
- Z.java → 02_16_2021_11_29_45

V02_16_2021_13_29_45:

- X.java → 02_16_2021_13_29_45
- Y.java → 02_16_2021_11_29_45
- Z.java → 02_16_2021_11_29_45

Approach 4: Use Time and Date as the Version Number

Rather than using an escalating integer version number, we could use the current time and date. What could go wrong in this approach?

```
.git/02_16_2021_03_29_45/X.java
.git/02_16_2021_03_29_45/Y.java

.git/02_16_2021_11_29_45/Y.java
.git/02_16_2021_11_29_45/Z.java

.git/02_16_2021_13_29_45/X.java
```

V02_16_2021_03_29_45:

- X.java → 02_16_2021_03_29_45
- Y.java → 02_16_2021_03_29_45

V02_16_2021_11_29_45:

- X.java → 02_16_2021_03_29_45
- Y.java → 02_16_2021_11_29_45
- Z.java → 02_16_2021_11_29_45

V02_16_2021_13_29_45:

- X.java → 02_16_2021_13_29_45
- Y.java → 02_16_2021_11_29_45
- Z.java → 02_16_2021_11_29_45

Approach 4: Use Time and Date as the Version Number

Rather than using an escalating integer version number, we could use the current time and date.

- Possible concern: Two programmers make commits (or files) at the same time.

```
.git/02_16_2021_03_29_45/X.java
.git/02_16_2021_03_29_45/Y.java

.git/02_16_2021_11_29_45/Y.java
.git/02_16_2021_11_29_45/Z.java

.git/02_16_2021_13_29_45/X.java
```

V02_16_2021_03_29_45:
- X.java → 02_16_2021_03_29_45
- Y.java → 02_16_2021_03_29_45

V02_16_2021_11_29_45:
- X.java → 02_16_2021_03_29_45
- Y.java → 02_16_2021_11_29_45
- Z.java → 02_16_2021_11_29_45

V02_16_2021_13_29_45:
- X.java → 02_16_2021_13_29_45
- Y.java → 02_16_2021_11_29_45
- Z.java → 02_16_2021_11_29_45

Approach 5: Use a “Hash” as the Version Number

The actual approach employed by Git is to use the “git-SHA1 hash” of a file as its version number.

- The git-SHA1 hash is a deterministic function of the file’s contents.
  - Two identical files will always have the same git-SHA1 hash.
  - git-SHA1 hash is 160 bits long.
- Example: The git-SHA1 hash of the code to the left is as shown on the right (given in both binary and hexadecimal).

```
public class HelloWorld {
   public static void main(String[] args) {
     System.out.println("Hello World!");
   }
}
```

```
1100110110011001101110001100100010
0111001001110100010101011011010101
1110001111001011011011011111001101
1110110101110110100001000011000010
1000001010101100010101001000
```

66ccdc645c9d156d5c79
6dbe6ed768430c1562a2

Note: The git-SHA1 hash is the SHA1 hash of (file size + a zero + the file contents).

Using the git-SHA1 Hash

Example of how git uses the git-SHA1 hash to store HelloWorld.java

- First, git computes the git-SHA1 hash:
  - HelloWorld.java → 66ccdc645c9d156d5c796dbe6ed768430c1562a2
- Git creates a folder called .git/objects/66
  - The 66 is the first two characters of the git-SHA1 hash.
- Git stores the contents in a file called ccdc645c9d156d5c796dbe6ed768430c1562a2.
  - File is stored in a compressed format (zlib) to save space.

Let's try it out!

Approach Comparison

Approach Number	Information to use as file version number	Downside
1, 2, and 3	Commit ID (that goes up by 1) that includes the file.	No central server to decide which commit is “next” if people are working offline.
4	Date and time of file.	Awkward to deal with simultaneous file changes. Not as elegant as SHA1-hash.
5	git-SHA1 hash of file.	???

Approach Comparison

Approach Number	Information to use as file version number	Downside
1, 2, and 3	Commit ID (that goes up by 1) that includes the file.	No central server to decide which commit is “next” if people are working offline.
4	Date and time of file.	Awkward to deal with simultaneous file changes. Not as elegant as SHA1-hash.
5	git-SHA1 hash of file.	???

Can you think of something that could go wrong in approach 5?

Approach Comparison

Approach Number	Information to use as file version number	Downside
1, 2, and 3	Commit ID (that goes up by 1) that includes the file.	No central server to decide which commit is “next” if people are working offline.
4	Date and time of file.	Awkward to deal with simultaneous file changes. Not as elegant as SHA1-hash.
5	git-SHA1 hash of file.	???

Can you think of something that could go wrong in approach 5?

- git-SHA1 hash is only 160 bits. Infinite number of possible files, but only finite number of git-SHA1 hashes. Some files must have same git-SHA1 hash.

SHA1-Hash

Good news: The chance that two files have the same SHA hash is $1 / 2^{160}$ or roughly $1 / 10^{37}$.

- Would need roughly 2^80 files before we expect to see two files with the same SHA hash (see CS70).
- 2^80 is 1,208,925,819,614,629,174,706,176 files.

In other words, git has a “bug”, but it is unlikely to ever occur in the history of the universe.

Added Benefit of SHA1-Hashing: Security

Git uses the git-SHA1 hash to verify file integrity.

- Hard to sneak in a security vulnerability into a git repository.
- Suppose we have ImportantCode.java whose git-SHA1 hash is ee380e192bb631b585652001d909de8905df0080.
- If an attacker went into my .git/objects/ee folder and tried to replace 380e192bb631b585652001d909de8905df0080 so that it contained malicious code, then Git would detect the issue.
    - Detection occurs because the change would result in a change in the git-SHA1 hash.

Serializable and Storing Data Structures

Git Commits

Every commit in git stores (at least):

- An author.
- A date.
- A commit message.
- A list of all files and their versions.
  - Versions are git-SHA1 hashes.
- The parent's commit ID.
  - Example: aa45f...db's parent ID is d1bd...61.

```
$ git log --graph
* commit 7e41ce1a924ca616bded92bf5a4d0d899bdd6ca0
| Author: JoshHug <hug@cs.berkeley.edu>
| Date:   Tue Dec 2 23:00:53 2014 -0800
|
|     added parmesan
|
* commit aa45fbd68235e21393f27af44098c9b487345cdb
| Author: JoshHug <hug@cs.berkeley.edu>
| Date:   Tue Dec 2 23:00:35 2014 -0800
|
|     fixed cheese bug
|
* commit d1bde19ffd43a14ea959585df3fd0722c8aa0c61
  Author: JoshHug <hug@cs.berkeley.edu>
  Date:   Tue Dec 2 22:59:56 2014 -0800

      version 1 of my code
```

Git Commit IDs

The commit ID is the git-SHA1 hash of the commit.

- You might object: “A commit is an object, not a file”.
- Imagine a file containing the author, date, commit message, list of files and their versions, and parent ID, then git-SHA1 hash that.

```
$ git log --graph
* commit 7e41ce1a924ca616bded92bf5a4d0d899bdd6ca0
| Author: JoshHug <hug@cs.berkeley.edu>
| Date:   Tue Dec 2 23:00:53 2014 -0800
|
|     added parmesan
|
* commit aa45fbd68235e21393f27af44098c9b487345cdb
| Author: JoshHug <hug@cs.berkeley.edu>
| Date:   Tue Dec 2 23:00:35 2014 -0800
|
|     fixed cheese bug
|
* commit d1bde19ffd43a14ea959585df3fd0722c8aa0c61
  Author: JoshHug <hug@cs.berkeley.edu>
  Date:   Tue Dec 2 22:59:56 2014 -0800

      version 1 of my code
```

Representing a Commit in Java

Suppose we have the Commit class below.

- For simplicity, I've only included the four fields shown.

```
public class Commit {
    public String author;
    public String date;
    public String commitMessage;
    public String parentID;
    ...
}
```

Storing Commits

When a user of your project 2 creates a commit, you'll need to somehow store the object below so that it can be read later.

```
public class Commit {
    public String author;
    public String date;
    public String commitMessage;
    public String parentID;
    ...
}
```

Storing Commits using Serializable

Java has a built-in feature called Serializable that lets you store arbitrary objects.

- Easy to use: Just make your class implement `Serializable`.
  - There are no methods to implement (weird).
- Then use our `Utils` class to write/read objects to/from files.

```
public class Commit implements Serializable {
    public String author;
    public String date;
    public String commitMessage;
    public String parentID;
    ...
}
```

Let's see a quick demo.

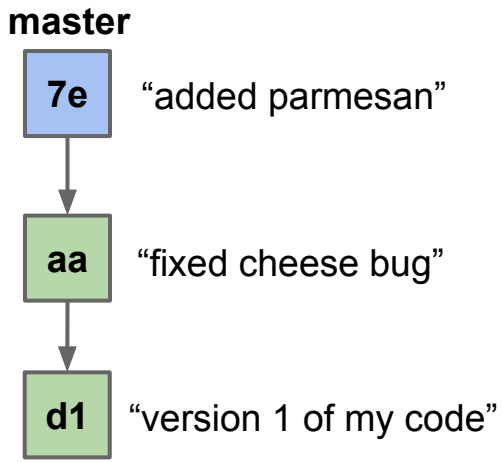
Branching

Merging

A common feature in version control systems is the ability to create branches.

- e.g. might “branch” from aa if I don’t trust the code in 7e and want to try something else.

```
$ git log --graph --oneline --all --decorate


* 7e41ce1 (HEAD, master) added parmesan
* aa45fbd fixed cheese bug
* d1bde19 version 1 of my code
```



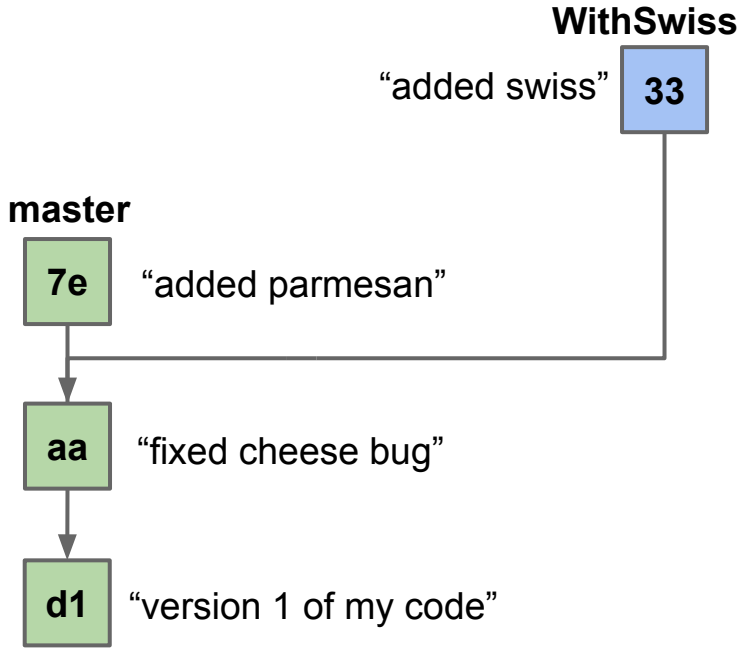
Note: An earlier version of the slide had some diagram errors.

Merging

A common feature in version control systems is the ability to create branches.

- e.g. might “branch” from aa if I don’t trust the code in 7e and want to try something else.

```
$ git checkout -b WithSwiss aa45fbd
$ subl Cheese.java
$ git add .; git commit -m “added swiss”
$ git log --graph --oneline --all --decorate

  * 33c7a92 (HEAD, WithSwiss) added swiss
* | 7e41ce1 (master) added parmesan
|/
* aa45fbd fixed cheese bug
* d1bde19 version 1 of my code
```

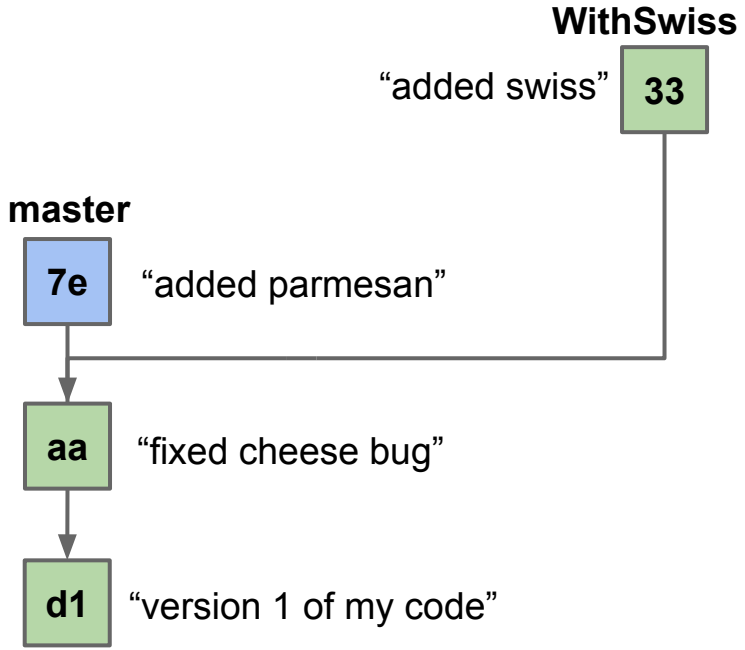


Note: An earlier version of the slide had some diagram errors.

Merging

Can switch back to the master branch with checkout.

```
$ git checkout master
$ git log --graph --oneline --all --decorate

  * 33c7a92 (WithSwiss) added swiss
* | 7e41ce1 (HEAD, master) added parmesan
|/
* aa45fbd fixed cheese bug
* d1bde19 version 1 of my code
```

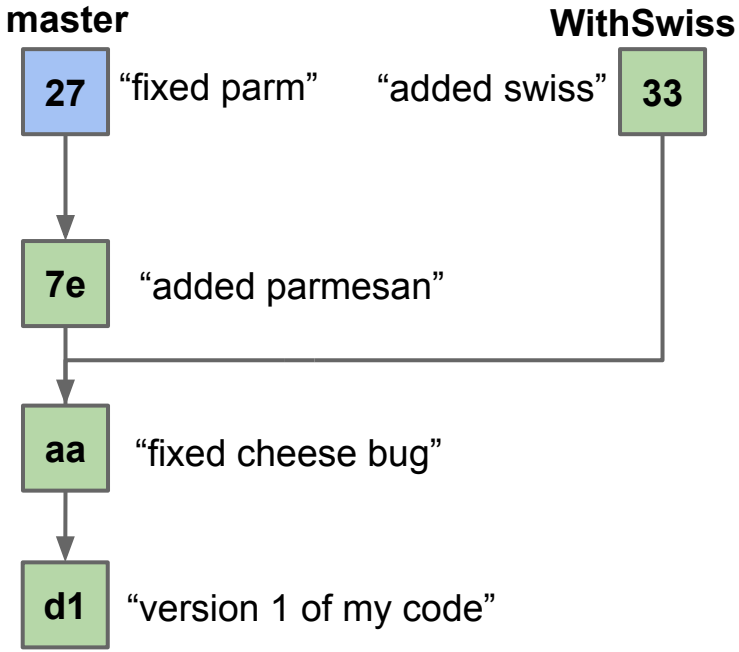


Note: An earlier version of the slide had some diagram errors.

Merging

After switching back to master branch, can continue to make changes.

```
$ git checkout master
$ subl Cheese.java
$ git add .; git commit -m “fixed parm”
$ git log --graph --oneline --all --decorate

  * 33c7a92 (WithSwiss) added swiss
* | 2720092 (HEAD, master) fixed parm
* | 7e41ce1 added parmesan
|/
* aa45fbd fixed cheese bug
* d1bde19 version 1 of my code
```



Note: An earlier version of the slide had some diagram errors.

Merging

Can (attempt to) merge branches.

```
$ git merge WithSwiss
$ Auto-merging Cheese.java
CONFLICT (content): Merge conflict in Cheese.java
Automatic merge failed; fix conflicts and commit the result.
```

Stuff that was in Cheese.java in the master branch, but not in the WithSwiss branch

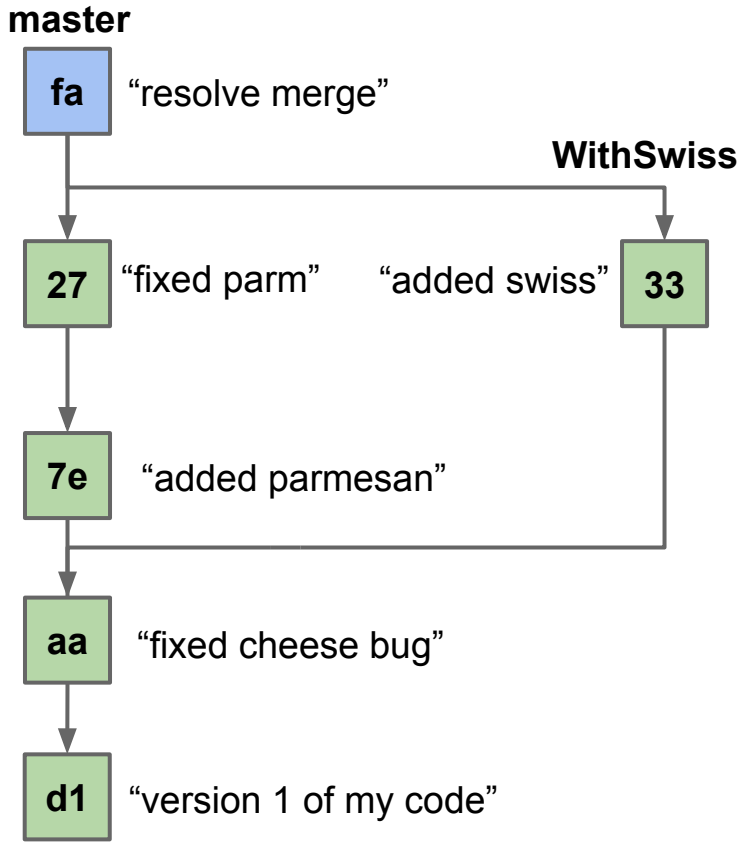
Stuff that was in Cheese.java in the WithSwiss branch, but not in the master branch

```
public class Cheese {
    blahblahblah
}
<<<<<<< HEAD
aefawefawef

public Cheese Parmesan {
    blahblahblah
}
=======

public Cheese Swiss {
    blahblahblah
}
>>>>>>> Swiss
```

Merging

After resolving conflict and making a new commit:

```
$ git add .; git commit -m “resolve merge”
$ git log --graph --oneline --all --decorate

*   faff9d1 (HEAD, master) resolve merge
|\
| * 33c7a92 (WithSwiss) added swiss
* | 2720092 fixed parm
* | 7e41ce1 added parmesan
|/
* aa45fbd fixed cheese bug
* d1bde19 version 1 of my code
```



Note: An earlier version of the slide had some diagram errors.

Merging

After resolving the conflict.

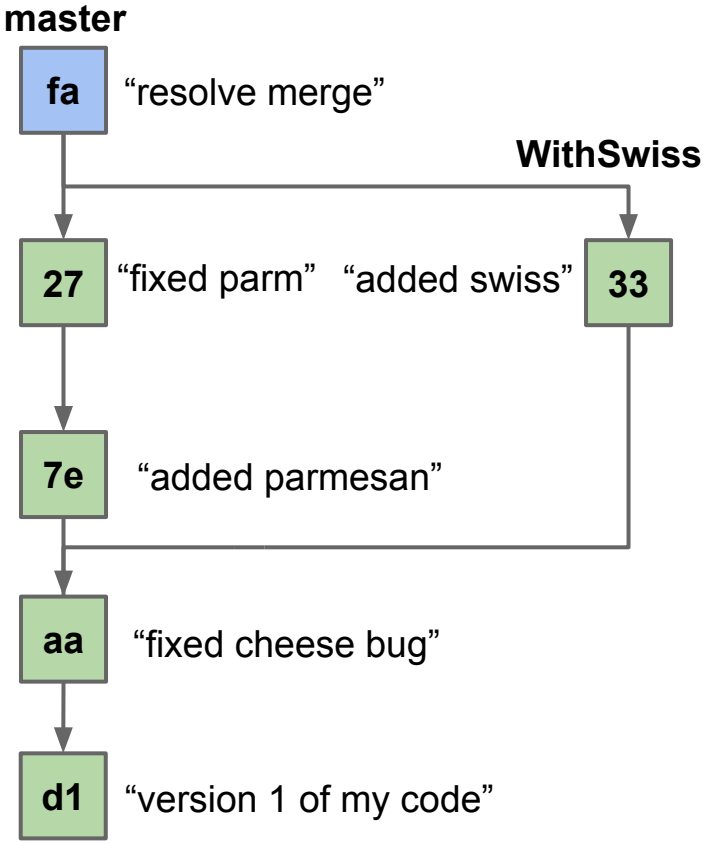
- The new commit has two parents!

Writing merge will be very tough.

- Today was just a brief look.
- Will have more supplementary videos later.

Note: Commits are no longer a linked list.

- This is a more general structure called a “graph”.
- More on graphs later in our class.



Conclusion

Today we got a sneak peek into how git works under the hood.

- Along the way we got a brief look at 4 different unrelated topics:
  - Maps: Same as a Python dictionary.
  - Hashing: Representing an object by a sequence of (160) bits.
  - Serialization: Saving and loading Java objects from a file.
  - Graphs: Generalization of a Linked List.
- Will talk about Maps, Hashing, and Graphs in much more detail in later lectures.
- Project 2 is very big!
  - But you'll have lots of time.
  - Not due until April.