CS 5004:
Object Oriented Design and Analysis

Chris Geeng, Tamara Bonaci

c.geeng@northeastern.edu

Announcement

- Assignments now due Tuesday 12:00PM
- Xinyi's OH: Tuesday 9-10AM

Agenda

- Today's lab – setting up the working environment

- Steps:
  1. Create your first Gradle project
  2. Create your first Java class
  3. Create your first Java test class
  4. Push your work to your remote GitHub repo – submitted!

INTRODUCTION TO UNIT TESTING AND JUNIT 5

Unit Testing

- Unit testing - search for errors in a subsystem in isolation
  - A "subsystem" typically means a particular class or object
  - The Java library JUnit helps us to easily perform unit testing
- Basic idea:
  - For a given class `Foo`, create another class `FooTest` to test it, containing various "test case" methods to run
  - Each method looks for particular results and either passes or fails
- JUnit provides "assert" commands to help us write tests
  - Idea - put assertion calls in your test methods to check things you expect to be true
  - If they are not, the test will fail

JUnit Assertion Methods

`assertTrue(`**test**`)`	fails if the boolean test is `false`
`assertFalse(`**test**`)`	fails if the boolean test is `true`
`assertEquals(`**expected**`, `**actual**`)`	fails if the values are not equal
`assertSame(`**expected**`, `**actual**`)`	fails if the values are not the same (by ==)
`assertNotSame(`**expected**`, `**actual**`)`	fails if the values *are* the same (by ==)
`assertNull(`**value**`)`	fails if the given value is *not* `null`
`assertNotNull(`**value**`)`	fails if the given value is `null`
`fail()`	causes current test to immediately fail

Import statements for *assert…()*

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
class ZooTest {
    private Zoo zoo1;  5 usages

    @BeforeEach
    void setUp() {
        zoo1 = new Zoo( name: "Cincinnatti Zoo",  city: "Cincinnatti",  state: "Ohio",  animalCount: 50);
    }

    @Test
    void getName() {
        assertEquals( expected: "Cincinnatti Zoo", zoo1.getName());
    }

    @Test
    void getCity() {
        assertEquals( expected: "Cincinnatti", zoo1.getCity());
    }

    @Test
    void getState() {
        assertEquals( expected: "Ohio", zoo1.getState());
    }

    @Test
    void getAnimalCount() {
        assertEquals( expected: 50, zoo1.getAnimalCount());
    }
}
```

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
class ZooTest {
    private Zoo zoo1;  5 usages

    @BeforeEach
    void setUp() {
        zoo1 = new Zoo( name: "C                                    state: "Ohio",  animalCount: 50);
    }

    @Test
    void getName() {
        assertEquals( expected: "Cincinnatti Zoo", zoo1.getName());
    }

    @Test
    void getCity() {
        assertEquals( expected: "Cincinnatti", zoo1.getCity());
    }

    @Test
    void getState() {
        assertEquals( expected: "Ohio", zoo1.getState());
    }

    @Test
    void getAnimalCount() {
        assertEquals( expected: 50, zoo1.getAnimalCount());
    }
}
```

@BeforeEach method runs before tests

```java
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;
class ZooTest {
    private Zoo zoo1;  5 usages

    @BeforeEach
    void setUp() {
        zoo1 = new Zoo( name: "Cincinnatti Zoo",  city: "Cincinnatti",  state: "Ohio",  animalCount: 50);
    }

    @Test
    void getName() {
        assertEquals
    }

    @Test
    void getCity() {
        assertEquals( expected: "Cincinnatti", zoo1.getCity());
    }

    @Test
    void getState() {
        assertEquals( expected: "Ohio", zoo1.getState());
    }

    @Test
    void getAnimalCount() {
        assertEquals( expected: 50, zoo1.getAnimalCount());
    }
}
```

Each @Test tells JUnit this is a test method, not a helper method

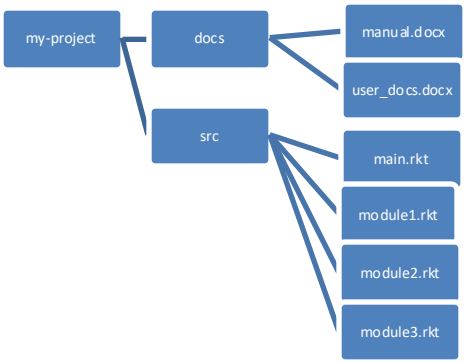
INTRODUCTION TO GIT

Introduction to Git

- Git is a distributed version-control system
  - Allows project collaboration
  - Enables creation of backup version of the code

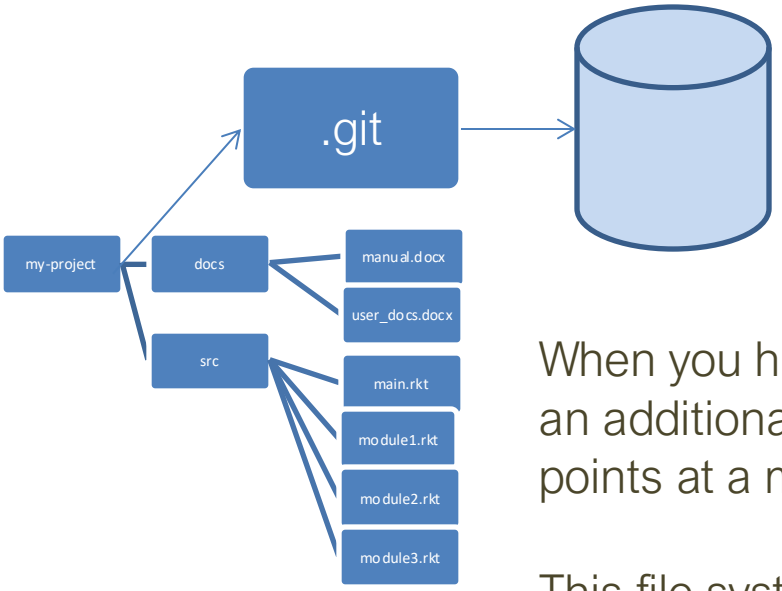
- How git works:
  1. You keep your files in a repository on your local machine.
  2. You synchronize your repository with a repository on a server.
  3. If you move from one machine to another, you can pick up the changes by synchronizing with the server.
  4. If your partner uploads some changes to your files, you can pick those up by synchronizing with the server.

Simple Model of Git: Your Files



Here are your files, sitting in a directory called my-project

Your Files in Your git Repository

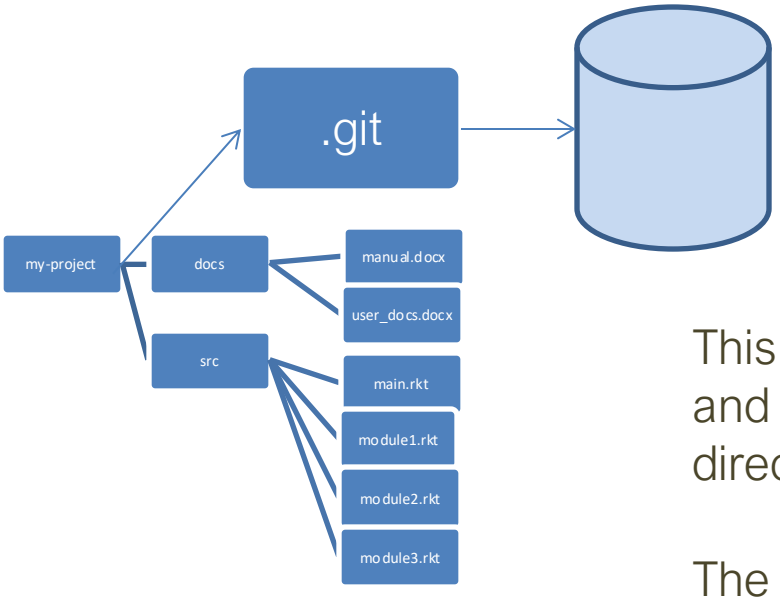


When you have a git repository, you have an additional directory called .git, which points at a mini-filesystem.

This file system keeps all your data, plus the bells and whistles that git needs to do its job.

All this sits on your local machine.

The git Client



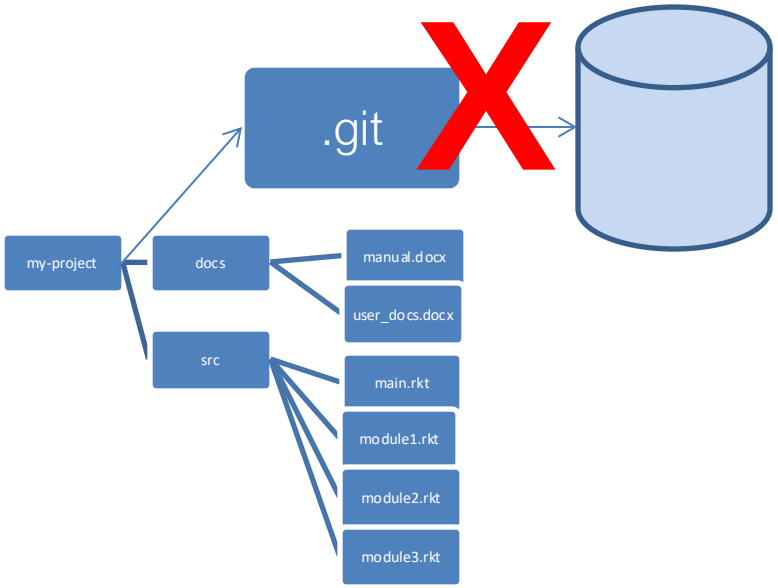
This mini-filesystem is highly optimized and very complicated. Don't try to read it directly.

The job of the git client (either Github for Windows, Github for Mac, or a suite of command-line utilities) is to manage this for you.

To remove repo and leave working directory:

- rm –rf .git

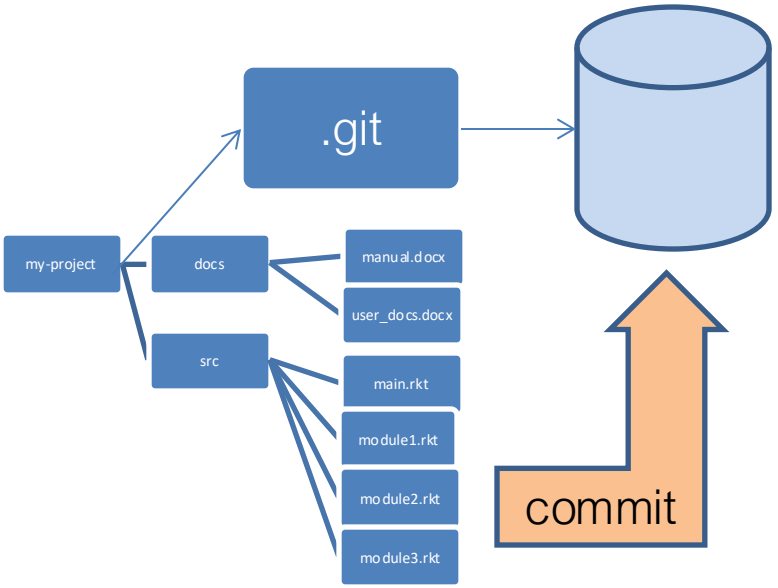
Be very careful with this command!!



Your Workflow

- You edit your local files directly
  - You can edit, add files, delete files, etc., using whatever tools you like
  - This does not change the mini-filesystem, so now your mini-fs is behind

A Commit



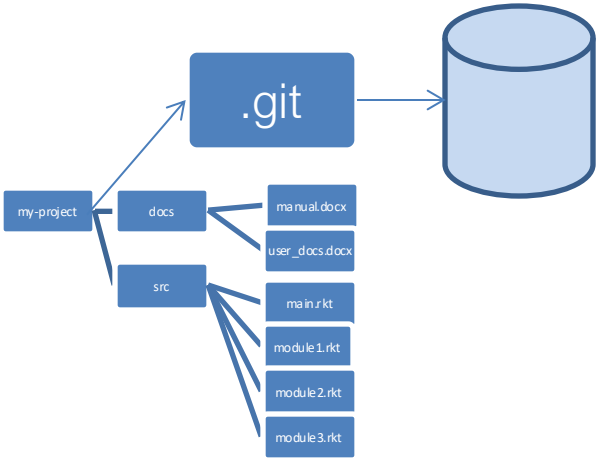
When you do a “commit”, you record all your local changes into the mini-fs

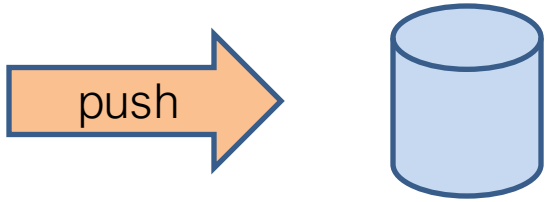
The mini-fs is “append-only”. Nothing is ever over-written there, so everything you ever commit can be recovered

Synchronizing with the Server -1

your local machine

a server, somewhere on the internet, eg. github.com



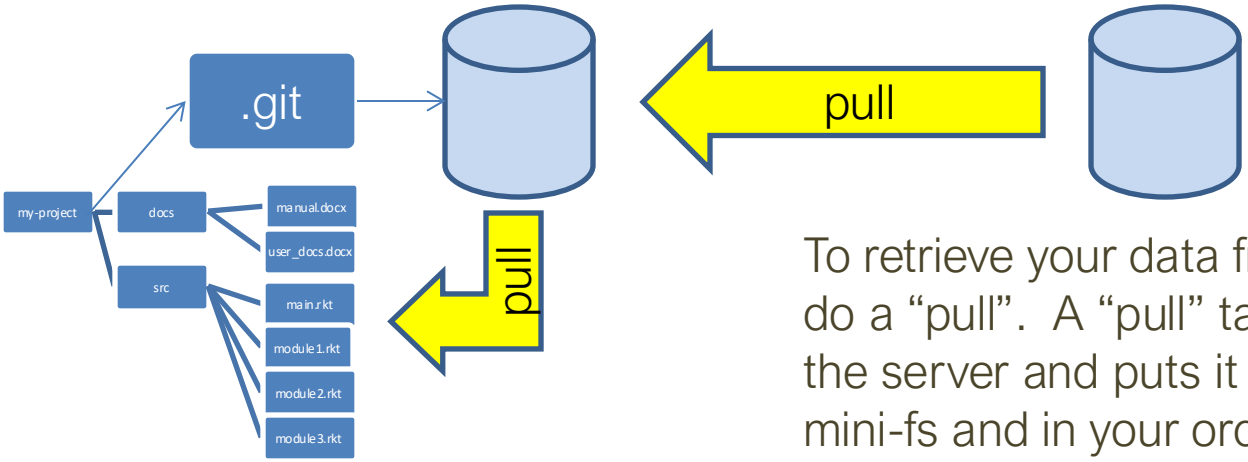


At the end of each work session, you need to save your changes on the server This is called a “push”.

Now all your data is backed up

- You can retrieve it, on your machine or some other machine.
- We can retrieve it (that’s how we collect homework)

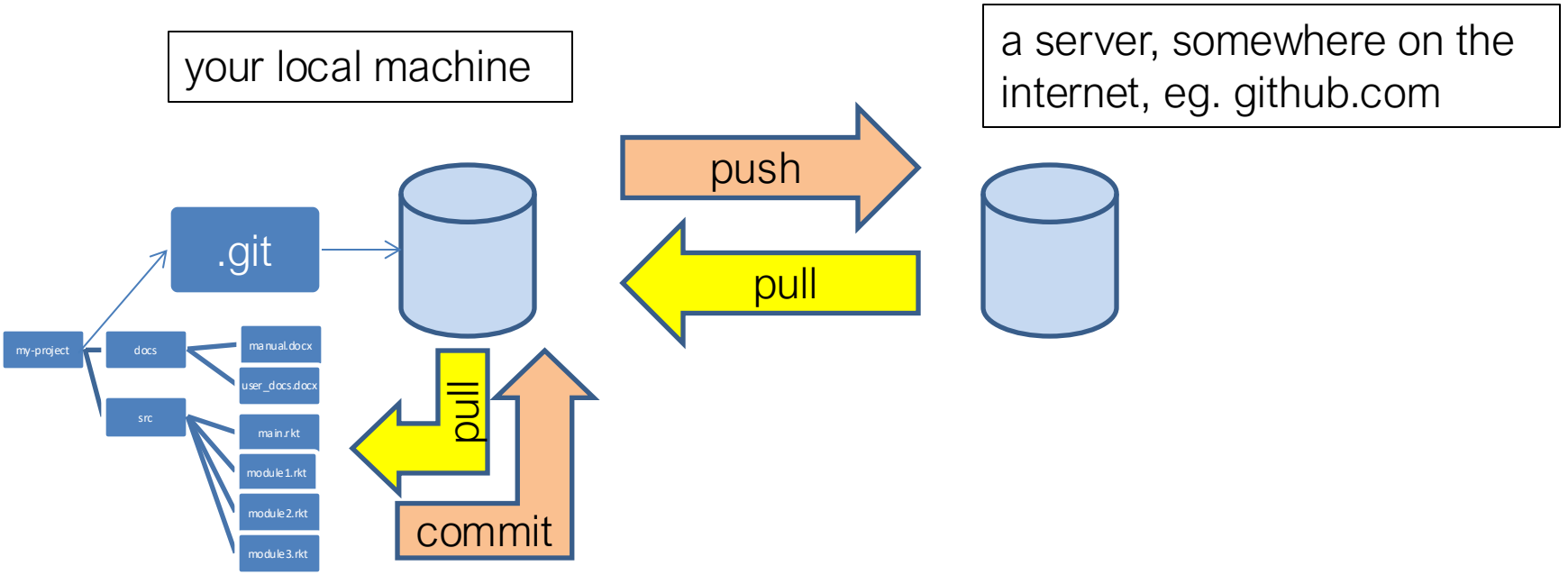
Synchronizing with the Server - 2



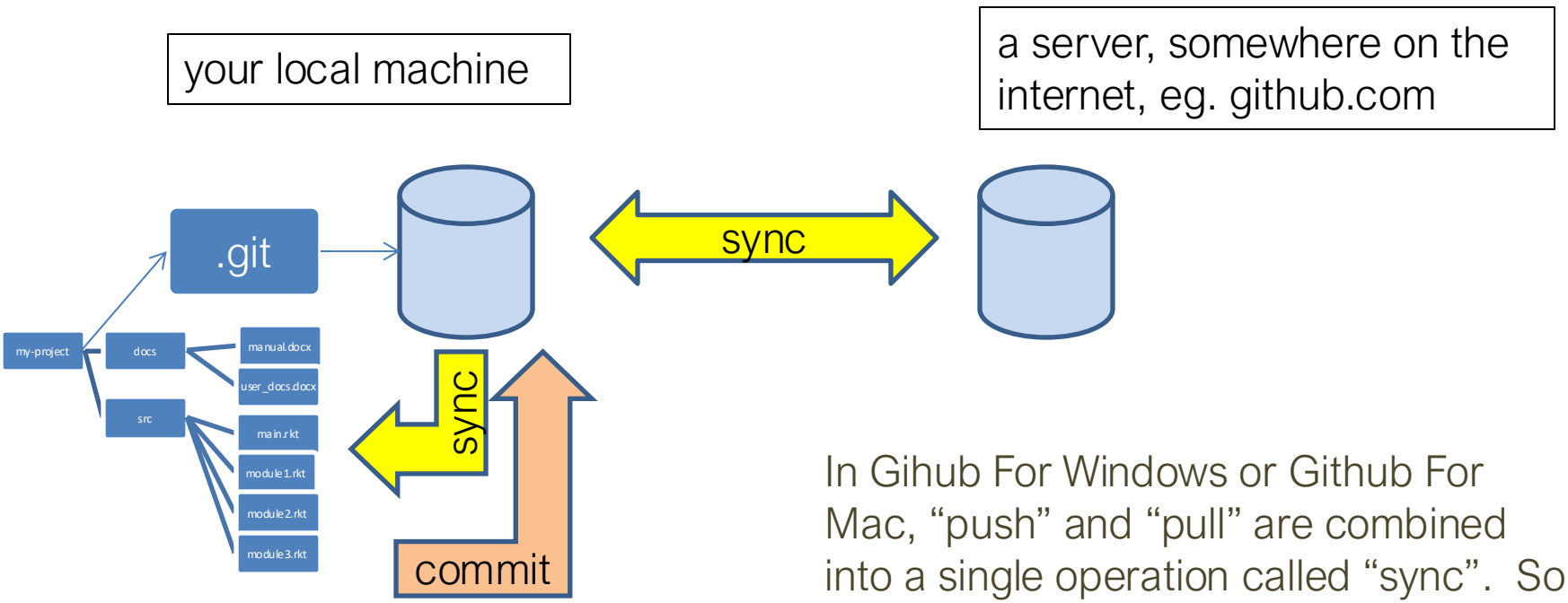
To retrieve your data from the server, you do a “pull”. A “pull” takes the data from the server and puts it both in your local mini-fs and in your ordinary files.

If your local file has changed, git will merge the changes if possible. If it can’t figure out how to the merge, you will get an error message.

The Whole Picture

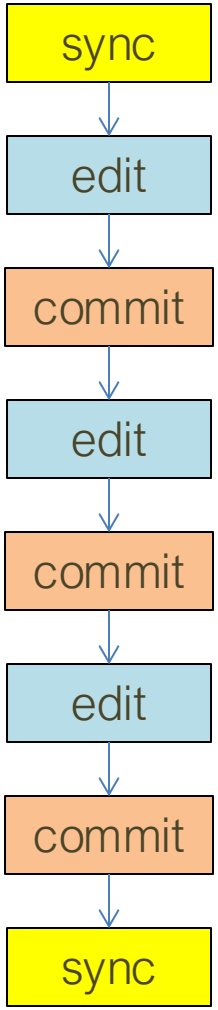


The Whole Picture using Desktop App



In Gihub For Windows or Github For Mac, “push” and “pull” are combined into a single operation called “sync”. So in these clients, there are only two steps (“commit” and “sync”) to worry about, not three.

Your Workflow - 2



Best practice: commit your work whenever you've gotten one part of your problem working, or before trying something that might fail.

If your new stuff is screwed up, you can always "revert" to your last good commit.
(Remember: always "revert", never "roll back")

Questions about Lab 1?

Documentation

Javadoc style comments

Constructor comments with @param for parameter

```
/**
 * Constructs a Person object and initializes it
 * to the given first name, last name and year of birth
 * @param firstName the first name of this person
 * @param lastName the last name of this person
 * @param yearOfBirth the year of birth of this person
 */
```

Method comment with @return for return value

```
/**
 * Get the first name of this person
 * @return the first name of this person
 */
```

Assignment 1

- Making a new package and a new class
- General Design Rubric
  - Code documentation
  - Gradle-built
  - Testing: Jacoco code coverage
  - Constants
    - private final static type CONSTANTNAME = …;

Equals and Comparisons

- Problem 2 in Assignment
- AssertEquals(object1, object1) will return False even if values are the same
  - B/c pointing to two different locations

- Compare primitive data types