JUnit
Test suites

• Obviously you have to test your code to get it working in the first place
  – You can do *ad hoc* testing (testing whatever occurs to you at the moment), or
  – You can build a **test suite** (a thorough set of tests that can be run at any time)

• Disadvantages of writing a test suite
  – It’s a lot of extra programming
    • *True*—but use of a good **test framework** can help quite a bit
  – You don’t have time to do all that extra work
    • *False*—Experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite

• Advantages of having a test suite
  – Your program will have many fewer bugs
  – It will be a **lot** easier to maintain and modify your program
    • This is a **huge** win for programs that, unlike class assignments, get actual use!
XP approach to testing

• In the Extreme Programming approach,
  – Tests are written before the code itself
  – If code has no automated test case, it is *assumed not to work*
  – A test framework is used so that automated testing can be done after every small change to the code
    • This may be as often as every 5 or 10 minutes
  – If a bug is found after development, a test is created to keep the bug from coming back

• Consequences
  – Fewer bugs
  – More maintainable code
  – *Continuous integration*—During development, the program *always works*—it may not do everything required, but what it does, it does right
JUnit

- **JUnit** is a framework for writing tests
  - JUnit was written by Erich Gamma (of *Design Patterns* fame) and Kent Beck (creator of XP methodology)
  - JUnit uses Java’s **reflection** capabilities (Java programs can examine their own code)
  - JUnit helps the programmer:
    - define and execute tests and test suites
    - formalize requirements and clarify architecture
    - write and debug code
    - integrate code and always be ready to release a working version
- Almost all IDEs include JUnit
Terminology

• A **test fixture** sets up the data (both objects and primitives) that are needed to run tests
  – Example: If you are testing code that updates an employee record, you need an employee record to test it on

• A **unit test** is a test of a *single* class

• A **test case** tests the response of a single method to a particular set of inputs

• A **test suite** is a collection of test cases

• A **test runner** is software that runs tests and reports results

• An **integration test** is a test of how well classes work together
  – JUnit provides some limited support for integration tests
Writing a JUnit test class in Eclipse

• Eclipse makes writing JUnit tests very easy
  – Right-click on the class you want to test -> New -> JUnit Test Case
  – Click Next, select all methods/stubs, click Finish and your test class is created
  – Eclipse automatically adds the import:
    ```java
    import static org.junit.jupiter.api.Assertions.*;
    ```
  – You also need to add:
    ```java
    import org.junit.jupiter.api.*;
    ```
  – A test class is just a standard Java class
  – Usually you should declare a instance of the class you want to test
  – Other variables can be declared as well but don’t initialize them yet

• Eclipse example: Counter.java
Adding methods to your test class

• First, define a method (or several methods) to be executed before each test
  – Initialize your variables in this method, so that each test starts with a fresh set of values
  – You can call this setUp, for consistency with Python
  – Add the @BeforeEach statement before the method definition to indicate that it will be executed before each test

• Write your tests
  – Each test is a single void method
  – Add the @Test statement before each test to indicate that it is a test

• You can also write a method to be executed after each test
  – Usually, to close resources such as files
  – Use the @AfterEach statement
  – It is rare to use this

• Eclipse examples: Counter.java, CounterTest.java
Assert methods I

• Within a test,
  – Call the method being tested and get the actual result
  – **Assert** what the correct result should be with one of the **assert methods**
  – These methods are very similar to the ones in Python

• An assert method is a JUnit method that performs a test, and throws an **AssertionError** if the test fails
  – JUnit catches these Errors and shows you the result

• **static void assertTrue(boolean test)**
  **static void assertTrue(String message, boolean test)**
  – Throws an **AssertionError** if the test fails
  – The optional **message** is included in the Error

• **static void assertFalse(boolean test)**
  **static void assertFalse(String message, boolean test)**
  – Throws an **AssertionError** if the test fails
Assert methods II

- `assertEquals(expected, actual)`
  - `assertEquals(String message, expected, actual)`
    - `expected` and `actual` must be both objects or the same primitive type
    - For objects, uses your equals method, *if* you have defined it properly, as described on the previous slide

- `assertSame(Object expected, Object actual)`
  - `assertSame(String message, Object expected, Object actual)`
    - Asserts that two arguments refer to the same object

- `assertNotSame(Object expected, Object actual)`
  - `assertNotSame(String message, Object expected, Object actual)`
    - Asserts that two objects do not refer to the same object
Warning: equals

- As we saw before, you can compare *primitives* with ==
- For comparing objects, you need to define *public boolean equals(Object obj) { ... }*
  - The argument must be of type *Object*, which isn’t what you want, so you must *cast* it to the correct type
Assert methods III

• `assertNull(Object object)`
  `assertNull(String message, Object object)`
  – Asserts that the object is null (undefined)

• `assertNotNull(Object object)`
  `assertNotNull(String message, Object object)`
  – Asserts that the object is not null

• `fail()`
  `fail(String message)`
  – Causes the test to fail and throw an `AssertionFailedError`
  – Useful as a result of a complex test, when the other assert methods aren’t quite what you want
Unit testing overkill for Counter class?

• The Extreme Programming view is: *If it isn’t tested, it doesn’t work*
• You are not likely to have many classes this trivial in a real program, so writing JUnit tests for those few trivial classes is no big deal
• Often even XP programmers don’t bother writing tests for *simple* getter methods such as `getCount()`
  – It is also autogenerated so not likely to have bugs
• In general, test everything that has even the slightest chance of having a bug
  – I.e., all code that you wrote!
Special features of @Test

• You can limit how long a method is allowed to take
  – This is good protection against infinite loops
  – The test fails if the method takes too long
  – Need to import java.time.Duration.*

```java
@Test
public void testTimeout() {
    assertTimeout(ofSeconds(2), () -> someStatement);
}
```

• Some method calls should throw an exception
  – You can specify that a particular exception is expected
  – The test will pass if the expected exception is thrown, and fail otherwise

```java
@Test
public void testException() {
    assertThrows(SomeException.class, () -> someStatement);
}
```

• Eclipse example: CounterTest.java
Test-Driven Development (TDD)

• It is difficult to add JUnit tests to an existing program
  – The program probably wasn’t written with testing in mind
• It’s actually better to write the tests *before* writing the code you want to test
• This seems backward, but it really does work better:
  – When tests are written first, you have a clearer idea what to do when you write the methods
  – Because the tests are written first, the methods are necessarily written to be testable
  – Writing tests first encourages you to write simpler, single-purpose methods
  – Because the methods will be called from more than one environment (the “real” one, plus your test class), they tend to be more independent of the environment
Stubs

• In order to run our tests, the methods we are testing have to exist, but they don’t have to be right

• Instead of starting with “real” code, we start with stubs—minimal methods that always return the same values
  – A stub that returns **void** can be written with an empty body
  – A stub that returns a number can return **0** or **-1** or **666**, or whatever number is most likely to be **wrong**
  – A stub that returns a **boolean** value should usually return **false**
  – A stub that returns an object of any kind (including a **String** or an array) should return **null**

• When we run our test methods with these stubs, we want the test methods to **fail**!
  – This helps “test the tests” – to help make sure that an incorrect method doesn’t pass the tests
Disabling a test

• The `@Disable` annotation says to not run a test
  ```java
  @Disable("I don’t want Rado to know this doesn’t work")
  @Test
  public void add() {
    assertEquals(4, program.sum(2, 2));
  }
  ```

• You shouldn't use `@Disable` without a very good reason!
  – Code you submit should probably not have any disabled tests
Recommended approach

1. Write a test for some method you intend to write
   – If the method is fairly complex, test only the simplest case
2. Write a stub for the method
3. Run the test and make sure it fails
4. Replace the stub with code
   – Write just enough code to pass the tests
5. Run the test
   – If it fails, debug the method (or maybe debug the test); repeat until the test passes
6. If the method needs to do more, or handle more complex situations, add the tests for these first, and go back to step 3