# What is Unit Testing?

- Unit testing focuses on verifying the functionality of individual pieces (units) of code in isolation.

# Why Unit Testing is Important

- Ensures that code behaves as expected, prevents bugs early in development, and facilitates refactoring.

# Best Practices for Unit Testing

- Keep tests small and focused, name tests clearly, and isolate tests from external dependencies.

# Test-Driven Development (TDD)

- Write tests first, then write code to pass the tests. Red-Green-Refactor cycle is often followed.

# Anatomy of a Unit Test

- A typical unit test includes setup, execution, assertion, and teardown.

# Example: Simple Unit Test in JavaScript

```
function add(a, b) {
  return a + b;
}

function test(message, assertion) {
  if (assertion()) {
    console.log("Pass: " + message);
  } else {
    console.error("Fail: " + message);
  }
}

test('adds 1 + 2 to equal 3', () => {
  return add(1, 2) === 3;
});
```

# Edge Cases in Unit Testing

- Testing edge cases (e.g., null values, empty inputs) ensures robustness.

```
// Edge case tests
test('handles null values', () => {
    return add(1, null) === null && add(null, 2) === null && add(null, null) === null;
});

test('handles undefined values', () => {
    return add(1, undefined) === undefined && add(undefined, 2) === undefined && add(undefined, undefined) === undefined;
});

test('handles empty string', () => {
    return add(1, "") === "1" && add("", 2) === "2" && add("", "") === "";
});
```

# Mocks and Stubs in Unit Testing

- Use **mocks** and **stubs** to simulate dependencies and control the behavior of external systems.
- **Stubs**: Fake versions of things that give pre-determined answers. They help isolate the code you're testing.
- **Mocks**: Fake versions of things that record how they were used. They help verify that your code interacts with other parts of the system correctly.

# Example: Mocking

```
// Function to test
function isAdult(age) {
    return age >= 18;
}

// Mock
const mockAge = 20;

// Test with mock
test('mock age is adult', () => {
  return isAdult(mockAge);
});

function test(message, assertion) {
  if (assertion()) {
    console.log("Pass: " + message);
  } else {
    console.error("Fail: " + message);
  }
}
```

# Example: Stubs

```
// Function to test
function canDrive(age, licenseService) {
    return licenseService.hasLicense(age) && age >= 16;
}

// Stub (Corrected)
const stubLicenseService = {
    hasLicense: function(age) {
        return age >= 18; // Returns true if age is 18 or older
    }
};

// Test with stub
test('stub license service allows driving at 18', () => {
    const age = 18;
    return canDrive(age, stubLicenseService);
});

function test(message, assertion) {
    if (assertion()) {
        console.log("Pass: " + message);
    } else {
        console.error("Fail: " + message);
    }
}
```

# How to Write Good Assertions

- Ensure that your tests cover a range of inputs and make clear, focused assertions.

# Common Unit Testing Libraries (JavaScript)

- Popular tools include Jest, Mocha, Chai, and Jasmine for writing and running unit tests in JavaScript.

# Example: Scenario

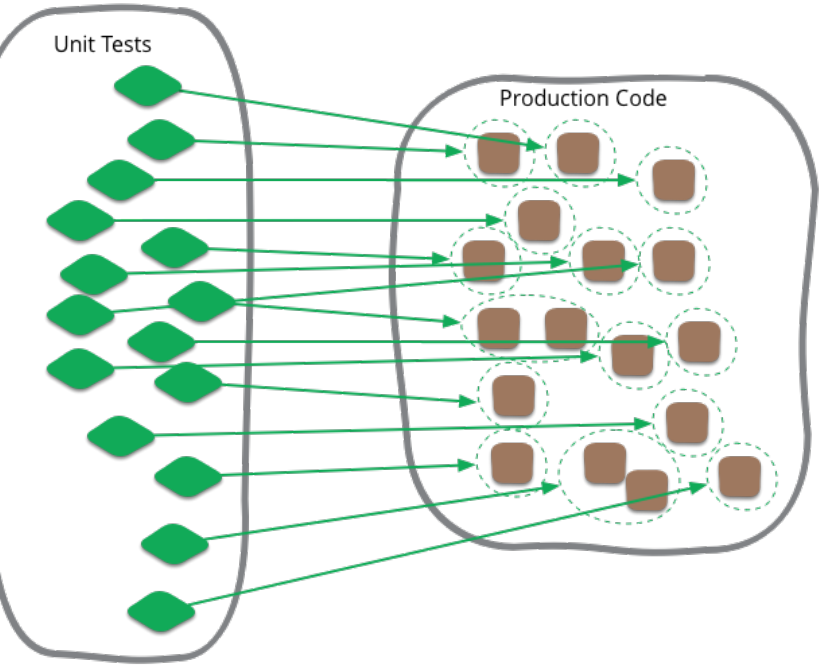
A teacher needs to update the grade of a student for a particular assignment. They would log into the system, search for the student by name or ID, navigate to the relevant course and assignment, and then enter the new grade. The system would then update the student's record with the new grade.

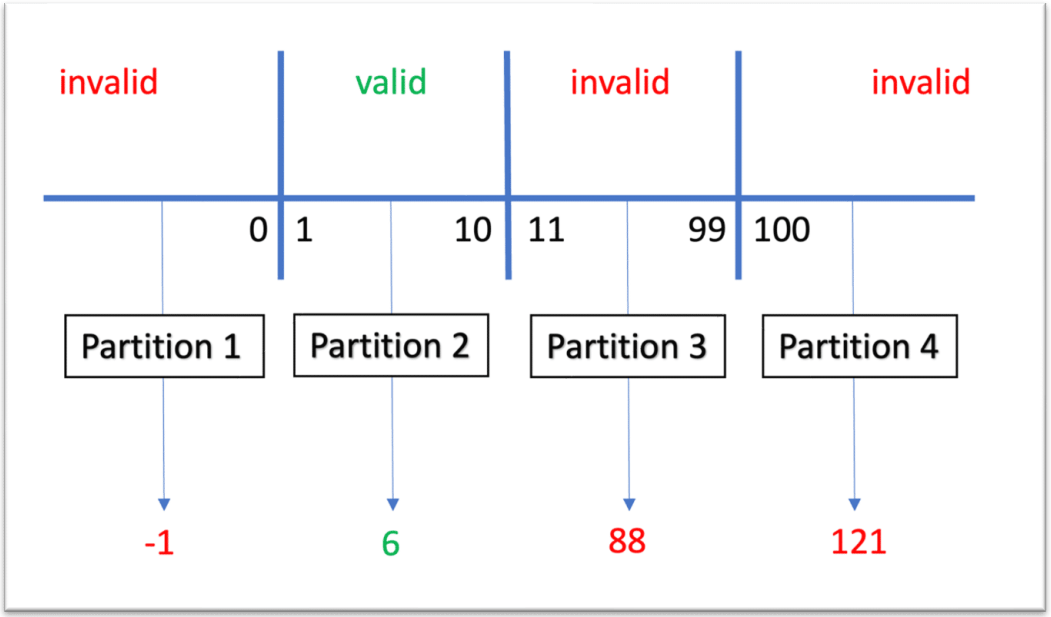
```
function test(message, assertion) {
    if (assertion()) {
    | console.log("Pass: " + message);
    } else {
    | console.error("Fail: " + message);
    }
}

test('calculates final grade for all marks above 70', () => {
    const marks = [75, 80, 72];
    const grade = calculateFinalGrade(marks); // calculateFinalGrade function doesn't exist yet
    return grade === 'First Class';
});
```

# Testing with Different Inputs

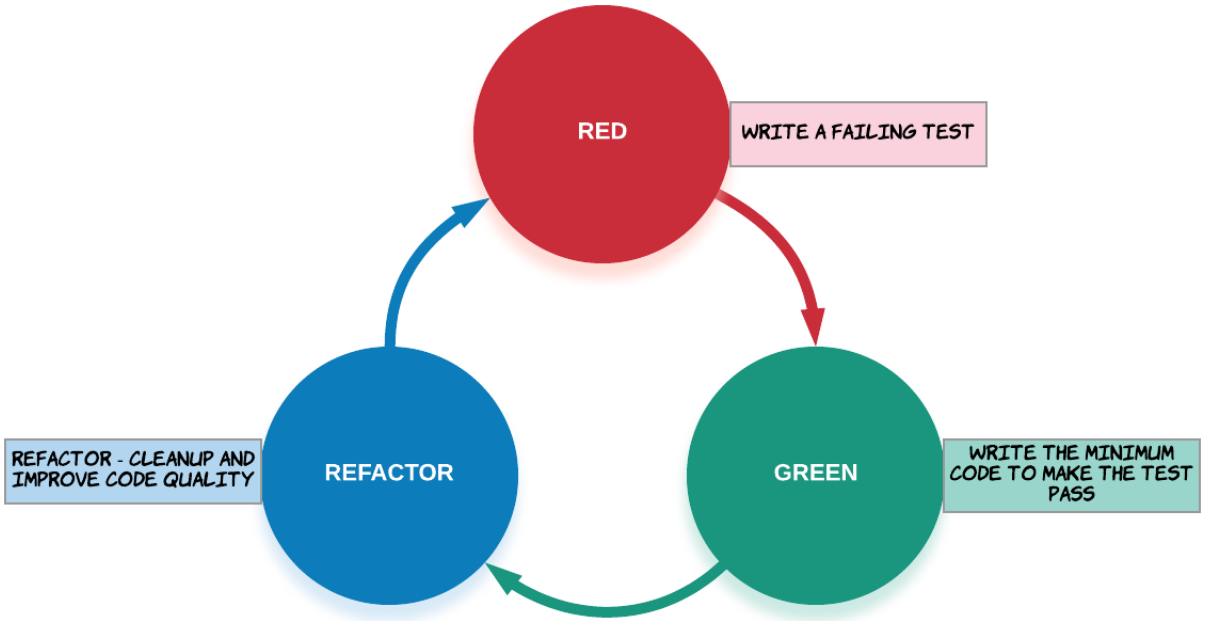
- Test cases should cover normal, boundary, and exceptional inputs to ensure comprehensive coverage.



# Refactoring and Unit Testing

- Unit tests enable safe refactoring. Code can be improved without breaking existing functionality.



# Unit Testing in Continuous Integration (CI)

- Integrating unit tests in CI/CD pipelines ensures that code is tested every time changes are made.