2023-03-24

10.Bonus: Testing

Remember this slide...

Memo-ization

```
private static long fibonacciCache[] = new long[1000];

// Precondition: n >= 0 && n < 1000

public long fibonacci(long n) {

  if (n == 0 || n == 1) {

    // Base cases: fibo(0) = 0, fibo(1) = 1

    return n;

  } else if (fibonacciCache[n] != 0) {

    return fibonacciCache[n];

  } else {

    // General case: fibo(n) = fibo(n-1) + fibo(n-2)

    return fibonacciCache[n] = fibonacci(n-1) + fibonacci(n-2);

  }

}
```

Memo-ization

```
private static long fibonacciCache[] = new long[1000];

// Precondition: n >= 0 && n < 1000

public long fibonacci(long n) {

  if (n == 0 || n == 1) {

    // Base cases: fibo(0) = 0, fibo(1) = 1

    return n;

  } else if (fibonacciCache[n] != 0) {

    return fibonacciCache[n];

  } else {

    // General case: fibo(n) = fibo(n-1) + fibo(n-2)

    return fibonacciCache[n] = fibonacci(n-1) + fibonacci(n-2);

  }

}
```

Memo-ization

```
private static long fibonacciCache[] = new long[1000];
// Precondition: n >= 0 && n < 1000
public long fibonacci( long n) {
  if (n == 0 || n == 1) {
    // Base cases: fibo(0) = 0, fibo(1) = 1
    return n;
  } else if (fibonacciCache[n] != 0) {
    return fibonacciCache[n];
  } else {
    // General case: fibo(n) = fibo(n-1) + fibo(n-2)
    return fibonacciCache[n] = fibonacci(n-1) + fibonacci(n-2);
  }
}
```

Memo-ization

```
private static long fibonacciCache[] = new long[1000];
// Precondition: n >= 0 && n < 1000
public long fibonacci( long n) {
  if (n == 0 || n == 1) {
    // Base cases: fibo(0) = 0, fibo(1) = 1
    return n;
  } else if (fibonacciCache[n] != 0) {
    return fibonacciCache[n];
  } else {
    // General case: fibo(n) = fibo(n-1) + fibo(n-2)
    return fibonacciCache[n] = fibonacci(n-1) + fibonacci(n-2);
  }
}
```

Testing

- During the development of a solution - either professionally, for a class project, or on the AP Exam - how do you verify your solution is correct?
- Today we are going to take a look at the kinds of data sets you should test your solutions against
- Then spend some time talking about how you can write code to test your code

printStatus

```
void printStatus(boolean isDone) {
  if (isDone) {
    System.out.println("Done!");
  } else {
    System.out.println("Not Done!");
  }
}
```

Analysis

Test Sets

printStatus

```
void printStatus(boolean isDone) {
  if (isDone) {
    System.out.println("Done!");
  } else {
    System.out.println("Not Done!");
  }
}
```

Analysis

- `isDone can be true or false`

Test Sets

printStatus

```
void printStatus(boolean isDone) {
  if (isDone) {
    System.out.println("Done!");
  } else {
    System.out.println("Not Done!");
  }
}
```

Analysis

- `isDone can be true or false`

Test Sets

- `true`
- `false`

calcFoodAmount

```
int calcFoodAmount(int age) {
  if (age < 3) {
    return 1;
  }
  if (age < 6) {
    return 3;
  }
  if (age < 10) {
    return 5;
  }
  return 10;
}
```

Analysis

Test Sets

calcFoodAmount

```
int calcFoodAmount(int age) {
  if (age < 3) {
    return 1;
  }
  if (age < 6) {
    return 3;
  }
  if (age < 10) {
    return 5;
  }
  return 10;
}
```

Analysis

- age could be negative, zero, or positive
- < is used for comparison evaluation: Are there any conditions where <= need special-casing

Test Sets

calcFoodAmount

```
int calcFoodAmount(int age) {
  if (age < 3) {
    return 1;
  }
  if (age < 6) {
    return 3;
  }
  if (age < 10) {
    return 5;
  }
  return 10;
}
```

Analysis

- age could be negative, zero, or positive
- < is used for comparison evaluation: Are there any conditions where <= need special-casing

Test Sets

- -1, 0, 1
- 3
- 6
- 10

reverseString

```
String reverseString(String s) {
  String result = "";
  for (int i = s.length() - 1; i >= 0; i--) {
    result += s.charAt(i);
  }
  return result;
}
```

Analysis

Test Sets

reverseString

```
String reverseString(String s) {
  String result = "";
  for (int i = s.length() - 1; i >= 0; i--) {
    result += s.charAt(i);
  }
  return result;
}
```

Analysis

- s could be null (probably okay to skip)
- s could be empty
- i is used as an index into s: Are there any conditions by which (i < 0) or (i > s.length-1)

Test Sets

reverseString

```
String reverseString(String s) {
  String result = "";
  for (int i = s.length() - 1; i >= 0; i--) {
    result += s.charAt(i);
  }
  return result;
}
```

Analysis

- s could be null (probably okay to skip)
- s could be empty
- i is used as an index into s: Are there any conditions by which (i < 0) or (i > s.length-1)

Test Sets

- ""
- "test"

findMinValue

```
int findMinValue(int[] array) {
  int minValue = array[0];
  for (int i = 1, n = array.length; i < n; i++) {
    if (array[i] < minValue) {
      minValue = array[i];
    }
  }
  return minValue;
}
```

Analysis

Test Sets

findMinValue

```
int findMinValue(int[] array) {
  int minValue = array[0];
  for (int i = 1, n = array.length; i < n; i++) {
    if (array[i] < minValue) {
      minValue = array[i];
    }
  }
  return minValue;
}
```

Analysis

- array could be null (probably okay to skip)
- array cannot be empty
- i is used as an index into array: Are there any conditions by which (i < 0) or (i > array.length-1)
- < is used for comparison evaluation: Are there any conditions where <= needs special-casing

Test Sets

findMinValue

```
int findMinValue(int[] array) {
  int minValue = array[0];
  for (int i = 1, n = array.length; i < n; i++) {
    if (array[i] < minValue) {
      minValue = array[i];
    }
  }
  return minValue;
}
```

Analysis

- array could be null (probably okay to skip)
- array cannot be empty
- i is used as an index into array: Are there any conditions by which (i < 0) or (i > array.length-1)
- < is used for comparison evaluation: Are there any conditions where <= needs special-casing

Test Sets

- [1]
- [1,2]
- [2,1]
- [1,1]

hasDuplicates

```
boolean hasDuplicates(int[] values) {
  int n = values.length;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      if (values[i] == values[j]) {
        return true;
      }
    }
  }
  return false;
}
```

Analysis

Test Sets

hasDuplicates

```
boolean hasDuplicates(int[] values) {
  int n = values.length;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      if (values[i] == values[j]) {
        return true;
      }
    }
  }
  return false;
}
```

Analysis

- values could be null (probably okay to skip)
- values can be empty
- i and j are both used as indexes into values: Are there any conditions by which (i|j < 0) or (i|j > values.length-1)

Test Sets

hasDuplicates

```
boolean hasDuplicates(int[] values) {
  int n = values.length;
  for (int i = 0; i < n; i++) {
    for (int j = i + 1; j < n; j++) {
      if (values[i] == values[j]) {
        return true;
      }
    }
  }
  return false;
}
```

Analysis

- values could be null (probably okay to skip)
- values can be empty
- i and j are both used as indexes into values: Are there any conditions by which (i|j < 0) or (i|j > values.length-1)

Test Sets

- null, [], [1]
- [1,2], [1,1]
- [1,2,1]

binarySearch

```
private int binarySearch(
  ArrayList<String> words, String targetWord) {
  int lowIdx = 0;
  int highIdx = words.size() - 1;
  while (lowIdx <= highIdx) {
    int midIdx = ((highIdx - lowIdx) / 2) + lowIdx;
    int middleCompareResult =
      words.get(midIdx).compareTo(targetWord);
    if (middleCompareResult == 0) {
      return midIdx;
    } else if (middleCompareResult > 0) {
      highIdx = midIdx - 1;
    } else {
      lowIdx = midIdx + 1;
    }
  }
  return -1;
}
```

Analysis

binarySearch

```
private int binarySearch(
  ArrayList<String> words, String targetWord) {
  int lowIdx = 0;
  int highIdx = words.size() - 1;
  while (lowIdx <= highIdx) {
    int midIdx = ((highIdx - lowIdx) / 2) + lowIdx;
    int middleCompareResult =
      words.get(midIdx).compareTo(targetWord);
    if (middleCompareResult == 0) {
      return midIdx;
    } else if (middleCompareResult > 0) {
      highIdx = midIdx - 1;
    } else {
      lowIdx = midIdx + 1;
    }
  }
  return -1;
}
```

Analysis

- words/targetWord could be null (probably okay to skip)
- words can be empty
- midIdx is used as an index into words: Are there any conditions by which (midIdx < 0) or (midIdx > words.length-1)
- int division is used to calculate midIdx: Are there any conditions where rounding results in unexpected results

binarySearch

```
private int binarySearch(
  ArrayList<String> words, String targetWord) {
  int lowIdx = 0;
  int highIdx = words.size() - 1;
  while (lowIdx <= highIdx) {
    int midIdx = ((highIdx - lowIdx) / 2) + lowIdx;
    int middleCompareResult =
      words.get(midIdx).compareTo(targetWord);
    if (middleCompareResult == 0) {
      return midIdx;
    } else if (middleCompareResult > 0) {
      highIdx = midIdx - 1;
    } else {
      lowIdx = midIdx + 1;
    }
  }
  return -1;
}
```

Analysis

- words/targetWord could be null (probably okay to skip)
- words can be empty
- midIdx is used as an index into words: Are there any conditions by which (midIdx < 0) or (midIdx > words.length-1)
- int division is used to

Test Sets

- ([],""), ([],"fig"),
- (["fig"],"fig"), (["date"],"fig"),
- (["fig","pear"],"fig"),
- (["pear","fig"],"fig"),
- (["pear","kiwi", "fig"],"fig"),
- (["pear","kiwi", "date"],"fig")

Automated Testing

- **Automated Testing** is a fancy way of saying that you have code that tests your code
  - The de-facto standard for how most software testing is currently performed
  - Reduced costs, increased scale, and continuous delivery
  - **"We only want to fix something one time"**
- This differs from **Manual Testing** where a human is manually interacting with your system (e.g. using the mouse or keyboard)
  - Typically done in conjunction with a **Test Plan** (document explaining the manual actions to take and the expected results)
  - Error-prone, expensive, and difficult to scale as your code grows
  - Still required for some platforms or solutions

Types of Automated Testing

- **Test Driven Development (TDD)**
  - As we discussed in Unit 3.2 - TDD is a popular development technique where you write the tests before your code - when the tests pass you're done!
  - A good hedge for groups where there is resistance to test writing
  - Can also help with the documentation effort
- **Unit Testing**
  - Tests are written alongside the code and verify at the class or method level
  - These typically test the public surface of classes and objects; To enforce their "contract" with the other components of the system
  - We have been using unit tests in a number of our exercises
  - A popular Java Unit Test framework is JUnit (more on that in a couple of slides)
- **UI or Browser Testing**
  - Test code mimics user interactions by sending mouse and keyboard events
  - Can be done "headless" - without the UI appearing on screen
  - Often makes use of APIs built to support assistive technologies

Types of Automated Testing

- **API Testing**
  - If your product offers a public API this testing makes sure the contracts don't ever break in an unexpected way.
  - Often simulate how API payloads are constructed and how APIs are versionsed
  - Often also have built-in capabiilties to easily support things like slow or intermittent connections and authentication and security
- **Code Coverage**
  - Systems that "instrument" code and test code to tell you how effective your tests are - e.g. which lines are exercised and which code branches are taken
  - If code coverage indicates that an object has no test coverage - then your confidence in making changes to the system can be reduced
  - **Tests the Tests**

Unit Testing with JUnit

```
class MyTestArea {
  @Test
  void firstTest() {
    assertEquals(expected, got)
  }
  @Test
  void secondTest() {
    assertTrue(got)
  }
}
```

Unit Testing with JUnit

```
class MyTestArea {
  @Test
  void firstTest() {
    assertEquals(expected, got)
  }
  @Test
  void secondTest() {
    assertTrue(got)
  }
}
```

- **A JUnit Test Class contains one or more related Tests - A JUnit Test Suite consists of many JUnit Test Classes**

Unit Testing with JUnit

```
class MyTestArea {
  @Test
  void firstTest() {
    assertEquals(expected, got)
  }
  @Test
  void secondTest() {
    assertTrue(got)
  }
}
```

- A JUnit Test Class contains one or more related Tests - A JUnit Test Suite consists of many JUnit Test Classes
- **Each Test is an instance method of the JUnit Test Class and has the @Test annotation**

Unit Testing with JUnit

```
class MyTestArea {
  @Test
  void firstTest() {
    assertEquals(expected, got)
  }
  @Test
  void secondTest() {
    assertTrue(got)
  }
}
```

- A JUnit Test Class contains one or more related Tests - A JUnit Test Suite consists of many JUnit Test Classes
- Each Test is a public instance method of the JUnit Test Class and has the @Test annotation
- **Each Test uses assertion statements to test for expected outcomes**

Unit Testing with JUnit

```
class MyTestArea {
  @Test
  void firstTest() {
    assertEquals(expected, got)
  }
  @Test
  void secon
    assertTru
  }
}
```

- A JUnit Test Class contains one or more related Tests - A JUnit Test Suite consists of many JUnit Test Classes
- Each Test is a public

the JUnit

the @Test

ertion

for

- JUnit Test Classes are regular Java classes and adhere to all the normal rules about what they can and cannot access.
- This means that generally JUnit Test Classes test the public surface - "the contract" with the rest of the system - and does not have access to the private internals.

Unit Testing with JUnit

```
class Student {
  private String name;

  public Student() {
  }

  public Student(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

Unit Testing with JUnit

```
class StudentTest {
  @Test
  void emptyConstruction() {
    Student s = new Student();
    assertEquals(null, s.getName())
  }

  @Test
  void goodConstruction() {
    Student s = new Student("Chris");
    assertEquals("Chris", s.getName())
  }
}
```

```
class Student {
  private String name;

  public Student() {
  }

  public Student(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

Unit Testing with JUnit

```
class StudentTest {
  @Test
  void emptyConstruction() {
    Student s = new Student();
    assertEquals(null, s.getName())
  }

  @Test
  void goodConstruction() {
    Student s = new Student("Chris");
    assertEquals("Chris", s.getName())
  }
}
```

```
class Student {
  private String name;

  public Student() {
  }

  public void Student(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

Unit Testing with JUnit

```
class StudentTest {
  @Test
  void emptyConstruction() {
    Student s = new Student();
    assertEquals(null, s.getName())
  }

  @Test
  void goodConstruction() {
    Student s = new Student("Chris");
    assertEquals("Chris", s.getName())
  }
}
```

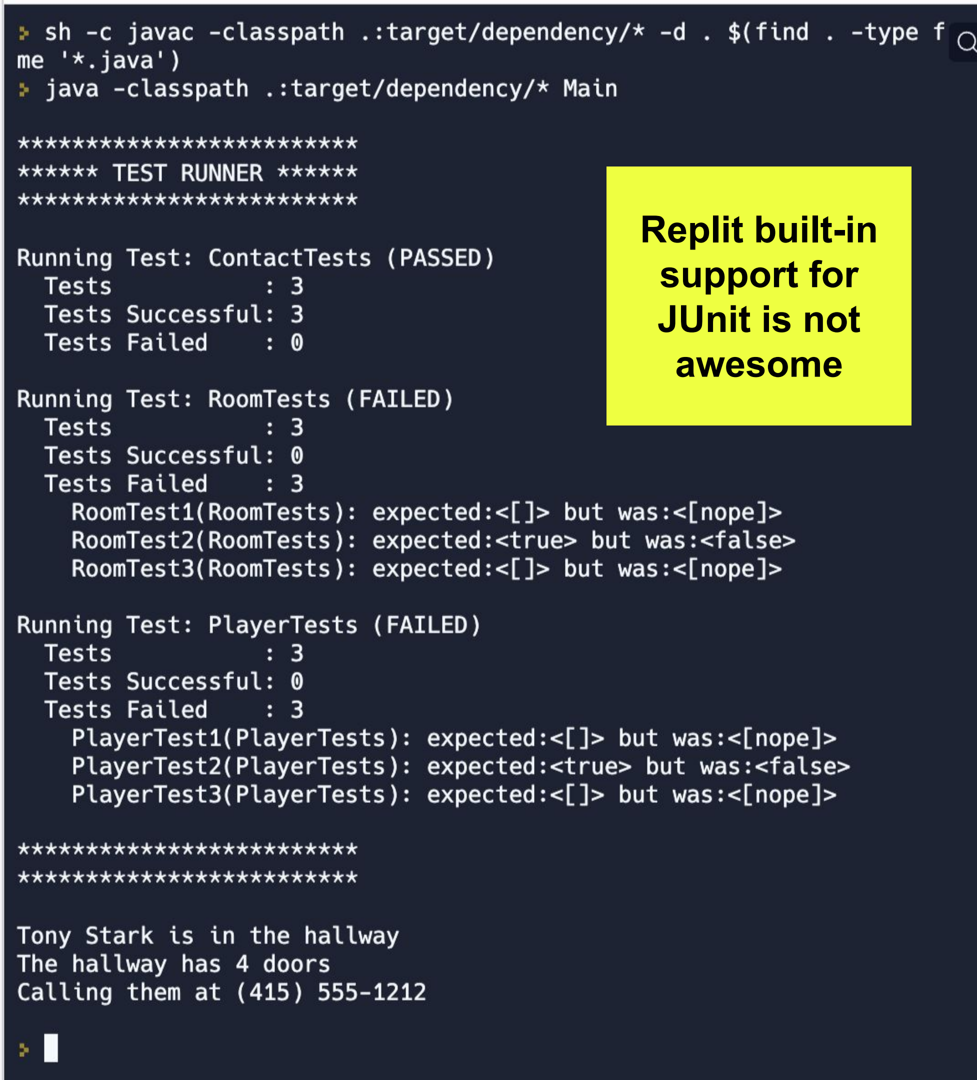
```
class Student {
  private String name;

  public Student() {
  }

  public Student(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
}
```

Unit Testing with JUnit

- JUnit has LOTS more functionality to explore
- Setup and Teardown
  - `@BeforeAll` - A static method run before any Test in a Test Class
  - `@BeforeEach` - An instance method run every Test in a Test Class
  - `@AfterEach` - An instance method run after every Test in a Test Class
  - `@AfterAll` - A static method run after the last Test in a Test Class
- Assertions
  - `assertEquals / assertNotEquals`
  - `assertTrue / assertFalse / assertNull`
  - `assertArrayEquals`
  - `assertInstanceOf / assertThrows`

Practice on your own

- Replit - Unit Testing
  - Basic program with three classes we have seen before: `Room`, `Contact`, and `Player`
  - Tests have been provided for Contact - **Add tests for the other two objects**
- References
  - Support for Unit Testing in Replit
  - JUnit References
    - User Guide
    - Writing Tests



```
› sh -c javac -classpath .:target/dependency/* -d . $(find . -type f
me '*.java')
› java -classpath .:target/dependency/* Main

************************
****** TEST RUNNER ******
************************

Running Test: ContactTests (PASSED)
  Tests            : 3
  Tests Successful: 3
  Tests Failed     : 0

Running Test: RoomTests (FAILED)
  Tests            : 3
  Tests Successful: 0
  Tests Failed     : 3
    RoomTest1(RoomTests): expected:<[]> but was:<[nope]>
    RoomTest2(RoomTests): expected:<true> but was:<false>
    RoomTest3(RoomTests): expected:<[]> but was:<[nope]>

Running Test: PlayerTests (FAILED)
  Tests            : 3
  Tests Successful: 0
  Tests Failed     : 3
    PlayerTest1(PlayerTests): expected:<[]> but was:<[nope]>
    PlayerTest2(PlayerTests): expected:<true> but was:<false>
    PlayerTest3(PlayerTests): expected:<[]> but was:<[nope]>

************************
************************

Tony Stark is in the hallway
The hallway has 4 doors
Calling them at (415) 555-1212

› █
```

Replit built-in support for JUnit is not awesome