Lecture 9 - Oct. 3

TDD with JUnit

Parsing Integers
Error Handling: Console vs. Exceptions
Deriving Test Cases
JUnit Test Method vs. Method Under Test
Regression Testing
JUnit Test: An Exception Not Expected

Announcements/Reminders

- **Written Test 1** result to be released Fri or Mon
- **Lab1** due tomorrow (Friday) at **noon**
- **Lab2** to be released tomorrow

More Example: Parsing Strings as Integers

```
Scanner input = new Scanner(System.in);
boolean validInteger = false;
while (!validInteger) {
  System.out.println("Enter an integer:");
  String userInput = input.nextLine();
  try {
    int userInteger = Integer.parseInt(userInput);
    validInteger = true;
  }
  catch(NumberFormatException e) {
    System.out.println(userInput + " is not a valid integer.");
    /* validInteger remains false */
  }
}
```

Test Case: ✓

User Enters: twenty-three

User Then Enters: 23 ✓

"twenty-three"

23

23

"twenty-three" → NFE

23

Error Handling via Console Messages: Circles

```
class Circle {
  double radius;
  Circle() { /* radius defaults to 0 */ }
  void setRadius(double r) {
    if (r < 0) { System.out.println("Invalid radius."); }
    else { radius = r; }
  }
  double getArea() { return radius * radius * 3.14; }
}
```

-10

① No more CoS Req. enforced on the callers

② caller of this method (error originated) is no longer obliged to handle the error.

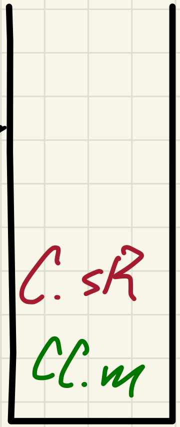
Caller?

Callee?

```
class CircleCalculator {
  public static void main(String[] args) {
    Circle c = new Circle();
    c.setRadius(-10);
    double area = c.getArea();
    System.out.println("Area: " + area);
  }
}
```

given that the callee already signaled an error, caller still proceeds to calculate

call stack



Error Handling via Console Messages: Banks

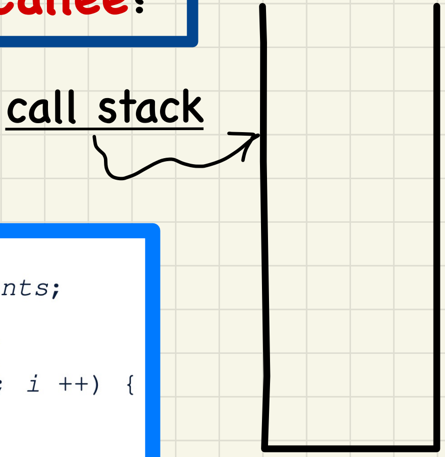
```
class Account {
  int id; double balance;
  Account(int id) { this.id = id; /* balance defaults to 0 */ }
  void deposit(double a) {
    if ( a < 0 ) { System.out.println("Invalid deposit."); }
    else { balance += a; }
  }
  void withdraw(double a) {
    if ( a < 0 || balance - a < 0 ) {
      System.out.println("Invalid withdraw."); }
    else { balance -= a; }
  }
}
```

Caller?

Callee?

```
class Bank {
  Account[] accounts; int numberOfAccounts;
  Bank(int id) { ... }
  void withdrawFrom(int id, double a) {
    for(int i = 0; i < numberOfAccounts; i ++) {
      if(accounts[i].id == id) {
        accounts[i].withdraw( a );
      }
    } /
  } /*
}
```

```
class BankApplication {
  pubic static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    Bank b = new Bank(); Account acc1 = new Account(23);
    b.addAccount(acc1);
    double a = input.nextDouble();
    b.withdrawFrom(23, a);
    System.out.println("Transaction Completed.");
}
```



context	caller	callee

Review: Specify-or-Catch Principle

Approach 1 – Specify: Indicate in the method signature that a specific exception might be thrown.

Example 1: Method that throws the exception

```
class C1 {
  void m1(int x) throws ValueTooSmallException {
    if(x < 0) {
      throw new ValueTooSmallException("val " + x);
    }
  }
}
```

origin of exception

Example 2: Method that calls another which throws the exception

```
class C2 {
  C1 c1;
  void m2(int x) throws ValueTooSmallException {
    c1.m1(x);
  }
}
```

Review: Specify-or-Catch Principle

Approach 2 – Catch: Handle the thrown exception(s) in a try-catch block.

```
class C3 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int x = input.nextInt();
    C2 c2 = new c2();
    try {
      c2.m2(x);
    }
    catch(ValueTooSmallException e) { ... }
  }
}
```

** run a test, and a VTLE/VTSE happened → pass
no VTLE/VTSE " → fail

Coming Up with Test Cases: A Single, Bounded Variable

* run a test, and an exception happened → fail
no exception " → pass

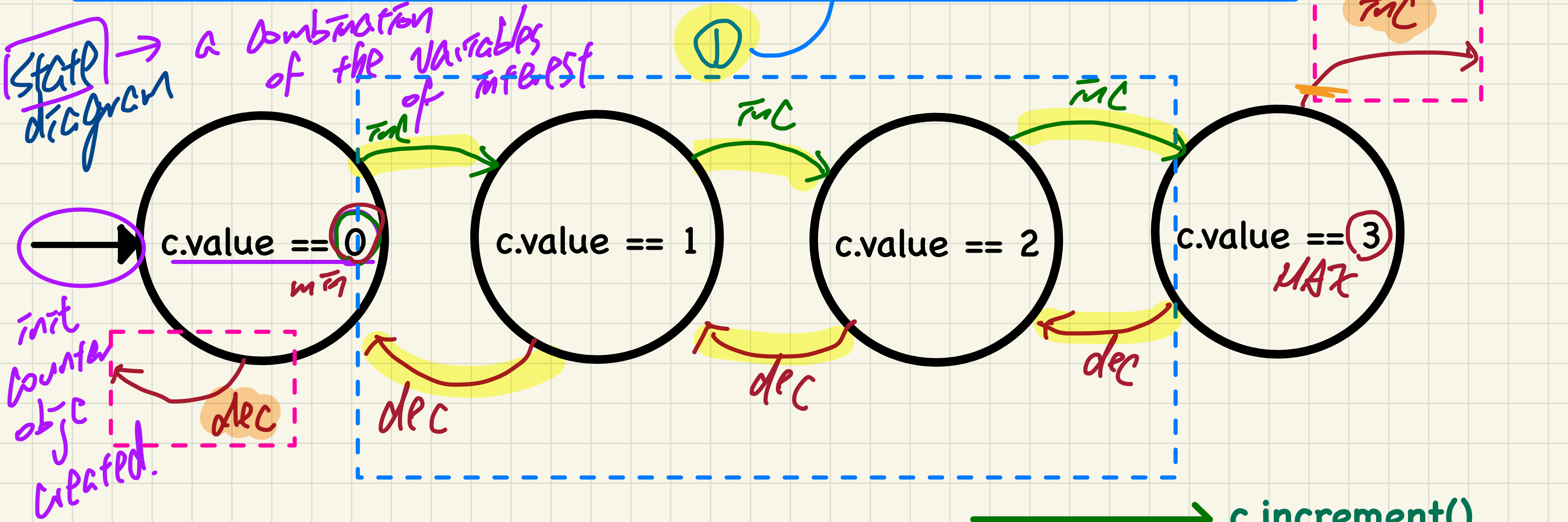
* no exception expected

** exception expected

Boundries:

Counter.MIN_VALUE <= c.value <= Counter.MAX_VALUE

state diagram → a combination of the variables of interest



c.increment()

c.decrement()

A Class for Bounded Counters

```java
public class Counter {
  public final static int MAX_VALUE = 3;
  public final static int MIN_VALUE = 0;
  private int value;
  public Counter() {
    this.value = Counter.MIN_VALUE;
  }
  public int getValue() {
    return value;
  }
  ... /* more later!
```

```java
/* class Counter */
  public void increment() throws ValueTooLargeException {
    if(value == Counter.MAX_VALUE) {
      throw new ValueTooLargeException("counter value is " + value);
    }
    else { value ++; }
  }

  public void decrement() throws ValueTooSmallException {
    if(value == Counter.MIN_VALUE) {
      throw new ValueTooSmallException("counter value is " + value);
    }
    else { value --; }
  }
}
```

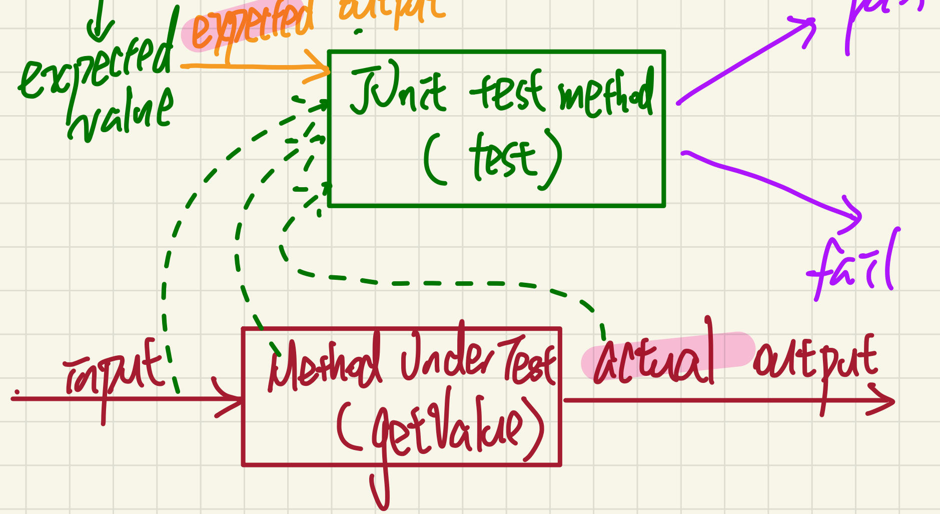
JUnit Test Method vs. Method Under Test

```
@Test
public void test() {
  MyClsss o = new MyClass();
  assertEquals(23, o.getValue());
}
```

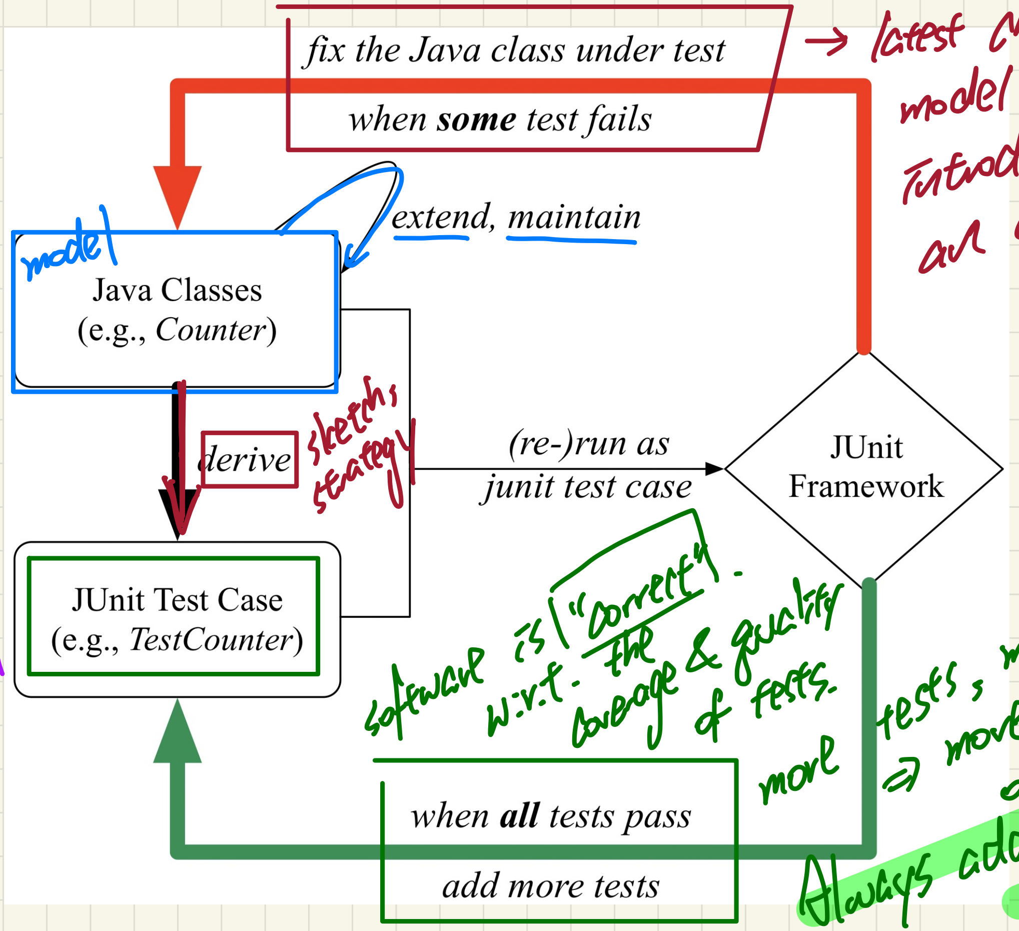
✓ test method

return value: actual value

method under test



Test-Driven Development (TDD): Regression Testing



→ latest change to model classes introduced an error

Incremental development ⇒ re-run all tests after each small, logical unit is imp.

software is "correct" w.r.t. the coverage & quality of tests.

more tests, more coverage ⇒ more confidence of correctness.

Always add more tests

JUnit: An Exception Not Expected ✓

```
@Test
public void testIncAfterCreation() {
  Counter c = new Counter();
  assertEquals(Counter.MIN_VALUE, c.getValue());
  try {
    c.increment(); → may throw VTLE
    assertEquals(1, c.getValue());
  }
  catch(ValueTooLargeException e) {
    /* Exception is not expected to be thrown. */
  → fail ("ValueTooLargeException is not expected.");
  }
}
```

① VTLE ② VTLE ✗ ✓

What if increment is implemented correctly?

① not throw VTLE
② increment

Expected Behaviour:

Calling c.increment()
when c.value is 0 should **not**
trigger a ValueTooLargeException

```
@Test
public void testIncAfterCreation() {
  Counter c = new Counter();
  assertEquals(Counter.MIN_VALUE, c.getValue());
  try {
    c.increment();
    assertEquals(1, c.getValue());
  }
  catch(ValueTooLargeException e) {
    /* Exception is not expected to be thrown. */
    fail ("ValueTooLargeException is not expected.");
  }
}
```

✓

① did not incre.
② throw VTLE unexpectedly.

What if increment is implemented incorrectly?

e.g., It throws VTLE when

c.value < Counter.MAX_VALUE

Running JUnit Test 1 on Correct Implementation

```java
public void increment() throws ValueTooLargeException {
⑤ if(value == Counter.MAX_VALUE) {
X  throw new ValueTooLargeException("counter value is " + value);
  }
⑥ else { ⑦ value ++; }
}
```

0 ③ 0 → ①

```java
@Test
public void testIncAfterCreation() {
① Counter c = new Counter();
② assertEquals(Counter.MIN_VALUE, c.getValue());
③ try { c.v==0 ✓
④   c.increment();
⑦   assertEquals(1, c.getValue());  1
⑧ }
X catch(ValueTooLargeException e) {
    /* Exception is not expected to be thrown. */
X   fail ("ValueTooLargeException is not expected.");
X }
}⑨
```

Running JUnit Test 1 on Incorrect Implementation

```java
public void increment() throws ValueTooLargeException {
  if(value == Counter.MAX_VALUE) {
    throw new ValueTooLargeException("counter value is " + value);
  }
  else { value ++; }
}
```

0 <

```java
@Test
public void testIncAfterCreation() {
  Counter c = new Counter();
  assertEquals(Counter.MIN_VALUE, c.getValue());
  try {
    c.increment();
    assertEquals(1, c.getValue());
  }
  catch(ValueTooLargeException e) {
    /* Exception is not expected to be thrown. */
    fail ("ValueTooLargeException is not expected.");
  }
}
```

0 0

c.v == 0

↳ reject the unexpected VTLE by failing the test