

CSE201: Monsoon 2024  
Advanced Programming

# **Lecture 15:**

## **Unit Testing and Inner Classes**


**Dr. Arun Balaji Buduru**

Founding Head, Usable Security Group (USG)

Associate Professor, Dept. of CSE | HCD

IIIT-Delhi, India

# Bugs and Testing

- **Software reliability:** Probability that a software system will not cause failure under specified conditions.
  - Measured by uptime, MTTF (mean time till failure), crash data.
- **Bugs** are inevitable in any complex software system. 
  - Industry estimates: 10-50 bugs per 1000 lines of code.
  - A bug can be visible or can hide in your code until much later.
- **Testing:** A systematic attempt to reveal errors.
  - Failed test: an error was demonstrated.
  - Passed test: no error was found (for this particular situation)

# Manual Testing v/s Automated Testing

Manual Testing	Automated Testing
Executing a test cases manually without any tool support is known as manual testing	Taking tool support and executing the test cases by using an automation tool is known as automation testing
<b>Time-consuming and tedious</b> – Since test cases are executed by human resources, it is very slow and tedious	<b>Fast</b> – Automation runs test cases significantly faster than human resources
<b>Huge investment in human resources</b> – As test cases need to be executed manually, more testers are required in manual testing	<b>Less investment in human resources</b> – Test cases are executed using automation tools, so less number of testers are required in automation testing
<b>Less reliable</b> – Manual testing is less reliable, as it has to account for human errors	<b>More reliable</b> – Automation tests are precise and reliable.
<b>Non-programmable</b> – No programming can be done to write sophisticated tests to fetch hidden information	<b>Programmable</b> – Testers can program sophisticated tests to bring out hidden information

# JUnit: Java Unit Testing Framework



- The Java library **JUnit** helps us to easily perform automated unit testing
- The 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 passes / fails
- JUnit provides "**assert**" commands to help us write tests.
  - The idea: Put assertion calls in your test methods to check things you expect to be true. If they aren't, the test will fail

# Sample JUnit Test

```
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public int sum () {
        return var1 + var2;
    }
}
```

```
/* Junit test class */

import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testSum() {
        Sum mySum = new Sum(1, 1);
        int sum = mySum.sum();
        assertEquals(2, sum);
    }
}
```

```
/* Junit test runner class */

import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
    public static void main(String[] args) {
        Result result= JUnitCore.runClasses(MyTest.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        System.out.println(result.wasSuccessful());
    }
}
```

*static import allows us to access the static members of a class directly without specifying the class name*

```
$ javac -cp ../path_to/junit-4.10.jar Sum.java MyTest.java TestRunner.java
```

```
$ java -cp ../path_to/junit-4.10.jar TestRunner
```

# JUnit Assertion Methods

<code>assertTrue(<b>test</b>)</code>	fails if the boolean test is <code>false</code>
<code>assertFalse(<b>test</b>)</code>	fails if the boolean test is <code>true</code>
<code>assertEquals(<b>expected</b>, <b>actual</b>)</code>	fails if the values are not equal
<code>assertSame(<b>expected</b>, <b>actual</b>)</code>	fails if the values are not the same (by <code>==</code> )
<code>assertNotSame(<b>expected</b>, <b>actual</b>)</code>	fails if the values <i>are</i> the same (by <code>==</code> )
<code>assertNotNull(<b>value</b>)</code>	fails if the given value is <i>not</i> <code>null</code>
<code>assertNotNull(<b>value</b>)</code>	fails if the given value is <code>null</code>
<code>fail()</code>	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. `assertEquals("message", expected, actual)`
  - Why is there no pass method?
- Detailed description: <https://junit.org/junit4/javadoc/4.8/org/junit/Assert.html>

# What is Wrong?

```
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }
}
```

```
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(2, 2);
        assertEquals(expected, mySum);
    }
}
```

- We are passing two objects of Sum type into the testIncr() method where the assertEquals checks for equality
  - Missing equals() method in Sum !
  - No compilation/runtime error but test will fail

# What's Still Wrong?

```
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }

    @Override
    public boolean equals(Object o) {
        if(o!=null && getClass()==o.getClass()) {
            Sum s = (Sum) o;
            return ((var1==s.var1)&&(var2==s.var2));
        }
        return false;
    }
}
```

```
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(3, 3);
        assertEquals(expected, mySum); //should fail
    }
}
```

*testIncr(MyTest):  
expected:<Sum @2e817b38> but  
was:<Sum @c4437c4>*

- Missing toString() method!!



# The Correct Version

```
/* The class method to be tested */
public class Sum {
    private int var1, var2;
    public Sum(int v1, int v2) {var1=v1; var2=v2;}
    public void incr () {
        var1++; var2++;
    }

    @Override
    public boolean equals(Object o) {
        if(o!=null && getClass()==o.getClass()) {
            Sum s = (Sum) o;
            return ((var1==s.var1)&&(var2==s.var2));
        }
        return false;
    }

    @Override
    public String toString() {
        return "("+Integer.toString(var1)+","
            +Integer.toString(var2)+")";
    }
}
```

```
/* Junit test class */
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class MyTest {

    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(3, 3);
        assertEquals(expected, mySum); //should fail
    }
}
```

*testIncr(MyTest):  
expected:<(3,3)> but was:<(2,2)>*

**Note: JUnit tests should be independent to each other as JUnit can run them in any order by using multithreading**

# Tests With a Timeout

```
@Test(timeout = 5000)
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
```

```
...
```

```
@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

# Testing for Exceptions

```
@Test(expected = ExceptionType.class)
public void name() {
    ...
}
```

- Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails
  - Use this to test for expected errors

```
@Test(expected = ArrayIndexOutOfBoundsException.class)
public void testBadIndex() {
    ArrayIntList list = new ArrayIntList();
    list.get(4);    // should fail
}
```

# Setup and Teardown

**@Before**

```
public void name() { ... }
```

**@After**

```
public void name() { ... }
```

- Methods to run before/after **each test case** method is called

**@BeforeClass**

```
public static void name() { ... }
```

**@AfterClass**

```
public static void name() { ... }
```

- Methods to **run once** before/after the entire test class runs

# JUnit Test Suites

```
/* JUnit testcase class-1 */
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class MyTest1 {
    @Test
    public void testSum() {
        Sum mySum = new Sum(1, 1);
        int sum = mySum.sum();
        assertEquals(2, sum);
    }
}
```

```
/* JUnit testcase class-2 */
import org.junit.Test;
import static org.junit.Assert.assertEquals;
public class MyTest2 {
    @Test
    public void testIncr() {
        Sum mySum = new Sum(1, 1);
        mySum.incr();
        Sum expected = new Sum(2, 2);
        assertEquals(expected, mySum);
    }
}
```

```
/* JUnit test suite class */

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({
    MyTest1.class,
    MyTest2.class
})

public class TestSuite { }
```

- **Test suite:** One class that runs many JUnit tests
  - An easy way to run all of your app's tests at once
- For this example, the classes Sum and TestRunner are still the same (Slide no. 7). Simply replace “MyTest” in TestRunner with “TestSuite”

# Tips for Testing

- You cannot test every possible input, parameter value, etc.
  - So you must think of a limited set of tests likely to expose bugs.
- Think about boundary cases
  - positive; zero; negative numbers
  - right at the edge of an array or collection's size
- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
- test behavior in combination
  - maybe add usually works, but fails after you call remove
  - make multiple calls; maybe size fails the second time only

# Trustworthy Tests

- Test one thing at a time per test method
  - 10 small tests are much better than 1 test 10x as large
- Each test method should have few (likely 1) assert statements
  - If you assert many things, the first that fails stops the test
  - You won't know whether a later assertion would have failed
- Tests should avoid logic.
  - minimize if/else, loops, switch, etc
  - avoid try/catch
    - If it's supposed to throw, use `expected=` ... if not, let JUnit catch it
- Torture tests are okay, but only *in addition to* simple tests



Let's change gears...



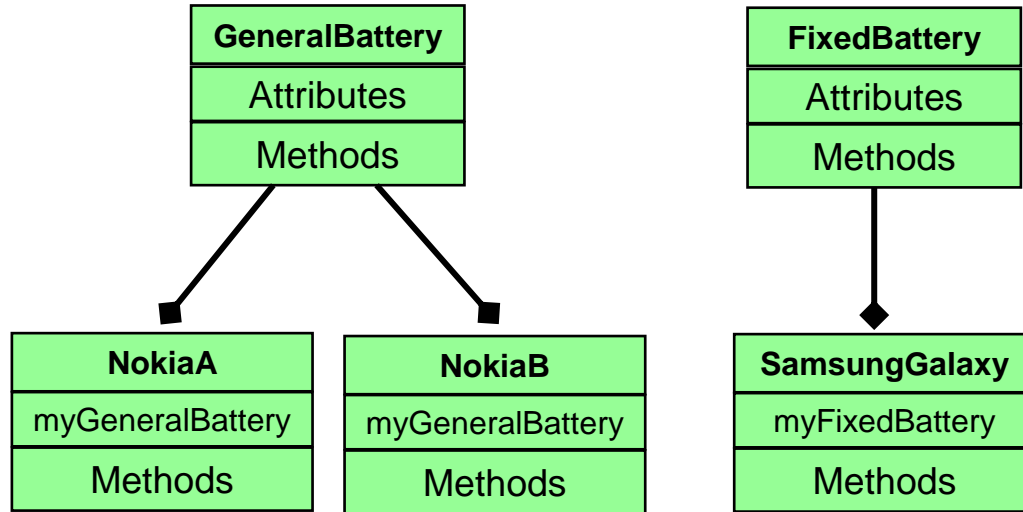
# Phone Batteries are Becoming Non-Removable



Non Removable Battery    Vs.    Removable Battery

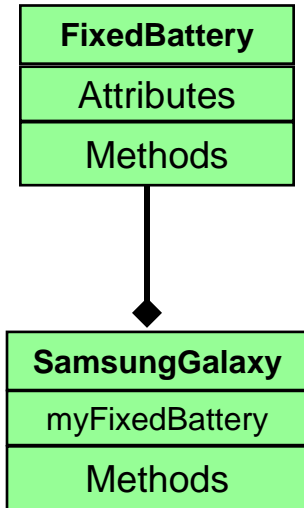
- Nowadays most of the phones are coming with non removable batteries
- Earlier, when there were removable batteries, we could easily keep a spare battery and replace when the primary one drained!
  - *Yes, I know we can use a power bank today!*

# Observations (1/2)



- In this example, several Nokia phones could use an instance of the same **GeneralBattery** class
- However, our class **SamsungGalaxy** uses an instance of **FixedBattery**
  - **FixedBattery** will never be instantiated outside **SamsungGalaxy** class

# Observations (2/2)



- These two classes are very much related
- Having two separate classes might mean two different files
  - Slightly less readability
- When FixedBattery type instance variable is only going to be used inside SamsungGalaxy, then why not include the contents of FixedBattery class into SamsungGalaxy class?
  - Object oriented programming ?

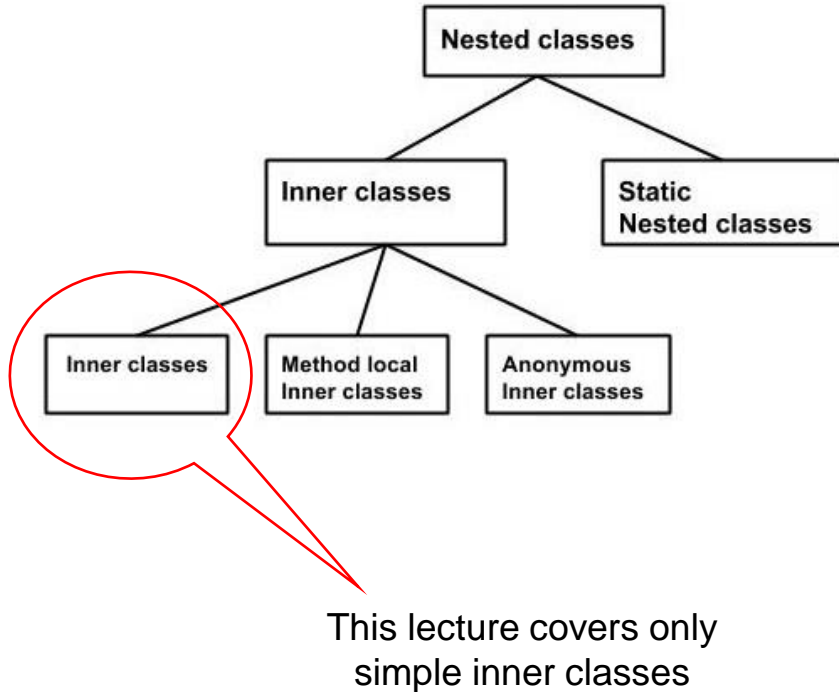
# Solution

- How about writing FixedBattery class inside SamsungGalaxy class? After all only SamsungGalaxy is going to use FixedBattery
  - How to nest and use the classes?

# Solution: Nested Class

```
public class SamsungGalaxy {  
    private FixedBattery myBattery;  
  
    public SamsungGalaxy() {  
        myBattery = new FixedBattery();  
    }  
  
    private class FixedBattery {  
        ....  
    }  
}
```

# Nested Classes in Java



- Non-static classes
  - Contains non-static members **only**
- Static nested classes
  - These contain static members of a class
- Method local inner class contains classes inside method body
- Anonymous inner classes are nameless class declared and instantiated at same time

# Inner Class

- Description
  - Class defined in scope of another class
- Property
  - Can directly access **all** variables & methods of enclosing class (including private fields & methods)
- Why inner class?
  - Logical grouping of functionality
  - Increases encapsulation
    - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world
  - More readability and maintainable code

# Question

```
public class SamsungGalaxy {
    private int version;           // =2
    private FixedBattery myBattery;

    public SamsungGalaxy() {
        myBattery = new FixedBattery();
    }

    private class FixedBattery {
        private int version;       // =3
        ....
        private void print() {
            int version = 4;
            System.out.println(version);
            System.out.println(this.version);
        }
    }

    System.out.println(SamsungGalaxy.this.version);
}
```

- Find the output in this program

Output = 4, 3, 2



# Inner Class Instance Inside a Method of Outer

```
public class SamsungGalaxy {  
  
    private FixedBattery myBattery;  
  
    public SamsungGalaxy() {  
        myBattery = new FixedBattery();  
    }  
  
    private class FixedBattery {  
        private boolean runDiagnosis() { ..... }  
        ....  
    }  
  
    public static void main(String[] args) {  
        SamsungGalaxy sg = new SamsungGalaxy();  
        SamsungGalaxy.FixedBattery sgb  
            = sg.new FixedBattery();  
        boolean test = sgb.runDiagnosis();  
    }  
}
```

- To instantiate the inner class in some other class, first we have to instantiate the outer class
- Thereafter, using the object of the outer class, we can instantiate the inner class
  - Note the usage of “new” keyword

# Inner Class Instance Outside Outer Class

```
public class SamsungGalaxy {  
  
    private FixedBattery myBattery;  
  
    public SamsungGalaxy() {  
        myBattery = new FixedBattery();  
    }  
  
    private class FixedBattery {  
        private boolean runDiagnosis() { ..... }  
        ....  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        SamsungGalaxy sg = new SamsungGalaxy();  
        SamsungGalaxy.FixedBattery sgb  
            = sg.new FixedBattery();  
        boolean test = sgb.runDiagnosis();  
    }  
}
```

- Is this code correct?
  - NO, compilation error!
- Inner class FixedBattery is private and hence cannot be accessed in another class
  - Rules of “private” modifier
  - Making the inner class and its method as public will correct this code