

Practical aspects

Write code that is easy to unit test

- We should write code such that we can easily provide test values
- We should write code such that we can easily check the result

```
class BadClass {  
    int method(int v1) {  
        something using v1;  
        int v2=...  
        something using v2;  
        return result;  
    }  
}
```

Difficult to test whether v2 is correct.
Add a print(v2) statement?

Difficult to test this part for different v2 values

You can easily check the
results for v2



You can easily provide your
own test values



```
class GoodClass {  
    int method1(int v1) {  
        something using v1;  
        int v2=...  
        return v2;  
    }  
    int method2(int v2) {  
        something using v2;  
        return result;  
    }  
}
```

Testing methods that need an object

- If we test non-static methods, we have to work with objects

```
class Element {
    Element next;
    int value;

    Element(int value) {
        this.value=value;
    }
}
```

```
class LinkedList {
    Element head=null;

    void add(Element e) {
        if(head!=null) {
            e.next=head;
        }
        head=e;
    }
}
```

- We want to test the add method for
 - Test case 1: empty lists
 - Test case 2: non-empty lists

Test case 1: Add element to empty list

```
class Element {
    Element next;
    int value;

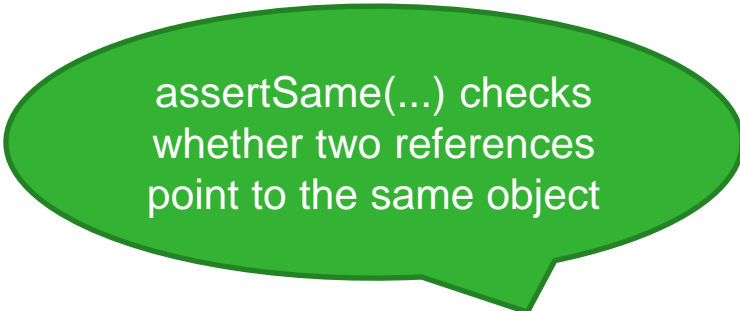
    Element(int value) {
        this.value=value;
    }
}
```

```
class LinkedList {
    Element head=null;

    void add(Element e) {
        if(head!=null) {
            e.next=head;
        }
        head=e;
    }
}
```

@Test

```
public void addToEmptyList() {
    LinkedList list=new LinkedList();
    Element e=new Element(2);
    list.add(e);
    assertSame("Empty list must contain element after add",list.head, e);
}
```



assertSame(...) checks whether two references point to the same object

Test case 2: Add element to non-empty list

```
class Element {
    Element next;
    int value;

    Element(int value) {
        this.value=value;
    }
}
```

```
class LinkedList {
    Element head=null;

    void add(Element e) {
        if(head!=null) {
            e.next=head;
        }
        head=e;
    }
}
```

@Test

```
public void addToNonEmptyList() {
    LinkedList list=new LinkedList();
    Element e1=new Element(2);
    list.add(e1);
    Element e2=new Element(3);
    list.add(e2);

    assertSame("Must contain new element as head after add", list.head, e2);
    assertSame("Old element must be second element after add",list.head.next,e1);
}
```

More JUnit features

- Sometimes, you want to test whether a method throws an exception:

```
void myMethod(int i) {  
    if(i<0)  
        throw new NumberFormatException();  
}
```

- In JUnit:

```
@Test(expected = NumberFormatException.class)  
public void myTest() {  
    MyClass.myMethod(-1);  
}
```

- You can also give a time limit :

```
@Test(timeout = 1000)           // <- 1000 milliseconds  
public void myTest() {  
    ...  
}
```

More JUnit features (2)

- Sometimes there are things that you want to do before or after each test
- Example:
 - You have written a program that writes a .png file
 - You want that the png file is deleted after a test
 - In JUnit:

```
@After
public void after() {
    // delete the file here
    ...
}
```

- See

https://www.tutorialspoint.com/junit/junit_execution_procedure.htm

Be careful with "Hidden" if

- Be careful with statements that look like basic statements...

- Example: `r = a/b;`

If b can be zero, this is what is actually happening in the program:

```
if (b==0)
    throw new DivisionByZeroException()
else
    r=a/b
```

- Example: `name = animal.getName();`

If animal can be null, this is what is actually happening in the program:

```
if (animal==null)
    throw new NullPointerException()
else
    name = animal.getName()
```