Exercise 1

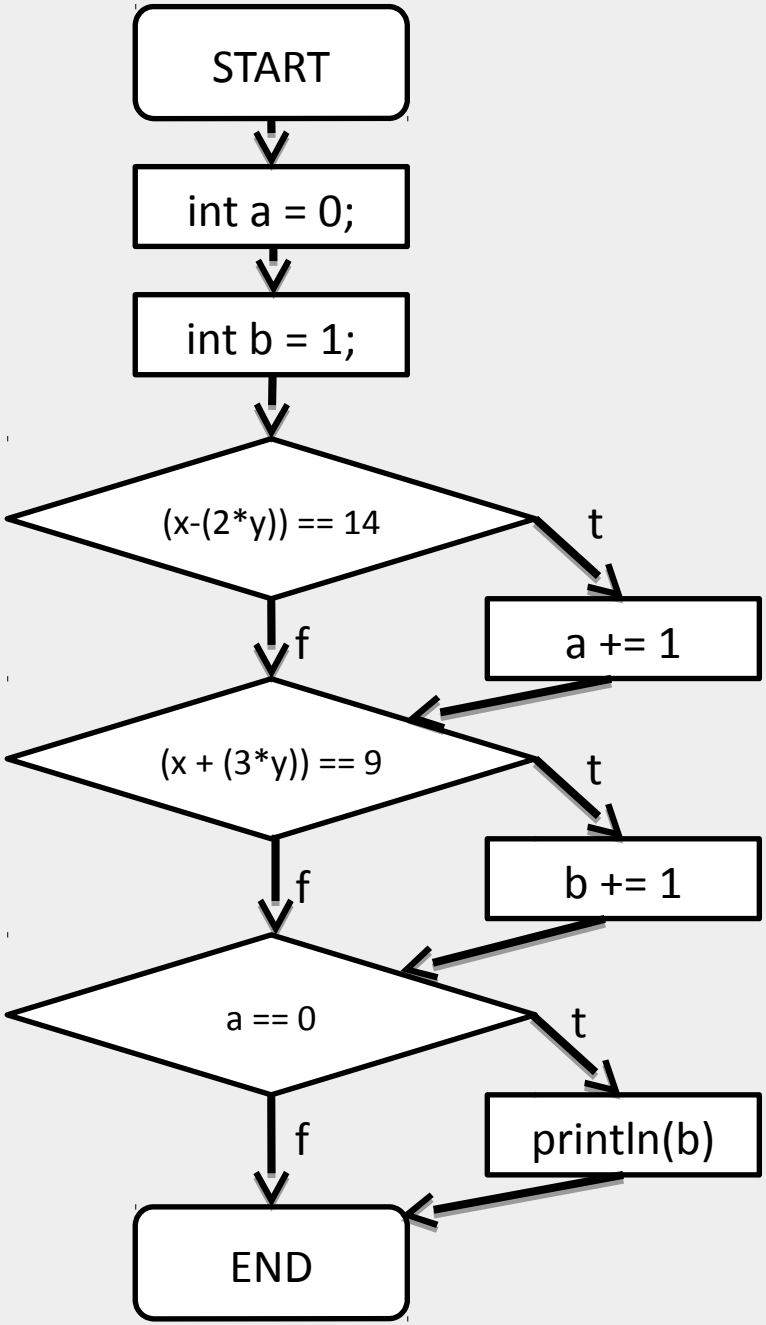
```
static void testMethod(int x, int y) {
   int a = 0;
   int b = 1;

   if((x-(2*y)) == 14)
    a = a + 1;

   if((x + (3*y)) == 9)
    b = b + 1;

   if(a == 0)
    System.out.println(b);
}
```

1. Draw a CFG for this method
2. Give a minimal size test set for:
   a) Statement coverage
   b) Branch coverage
   c) Path coverage (for feasible paths only of course)



Statement coverage

{ { x = 12, y = -1},
{ x = 0, y = 0} }

Branch coverage

Same as statement coverage
(at least in this exercise)

Path coverage (feasible paths)

{
{ x = 12, y = -1},
{ x = 0, y = 0},
{ x = 6, y = 1},
{ x = 16, y = 1}
}

Exercise 2

Write a black-box test set for the ArrayList<E>.get(int index) method. Write your test as a JUnit test.

Specification:

public E get(int index) Returns the element at the specified position in this list.

Parameters: index - index of the element to return

Returns: the element at the specified position in this list

Throws: IndexOutOfBoundsException - if the index is out of range (index < 0 || index >= size())

```
@Before
List jimmy = new ArrayList<int>();

@Test
public void TestValidIndex(){
    int n = 0;
    jimmy.add(n);
    Assert.equals(n, jimmy.get(0));
}

@Test (expected=IndexOutOfBoundsException.class)
public void TestInvalidIndex(){
    jimmy.get(1);
}

@Test (expected=IndexOutOfBoundsException.class)
public void TestIndexLessThanZero(){
    jimmy.get(-1);
}

@Test (expected=IndexOutOfBoundsException.class)
public void TestIndexGreaterThanSize(){
    jimmy.get(jimmy.size()+1);
}
```

Not so bad but most of the tests are performed on an empty list so only a special case is covered

```java
@Test(expected = IndexOutOfBoundsException.class)
public void testUpperIndexOutOfBoundException() throws IndexOutOfBoundsException{
     ArrayList<String> testList = new ArrayList<String>();
     testList.add("a");
     testList.get(-1);
}
@Test(expected = IndexOutOfBoundsException.class)
public void testLowerIndexOutOfBoundException() throws IndexOutOfBoundsException{
     ArrayList<String> testList = new ArrayList<String>();
     testList.add("a");
     testList.get(1);
}
public void testUpperBoundary() throws IndexOutOfBoundsException{
     ArrayList<String> testList = new ArrayList<String>();
     testList.add("a");
     testList.add("b");
     testList.add("c");
     assertEquals("c", testList.get(2));
}
public void testLowerBoundary() throws IndexOutOfBoundsException{
     ArrayList<String> testList = new ArrayList<String>();
     testList.add("a");
     testList.add("b");
     testList.add("c");
     assertEquals("a", testList.get(0));
}
```

Not so bad

Exercise 3

Write a black-box test set for the Comparable<Integer>.compareTo (Integer o) method. Write your test as a JUnit test.

Specification: Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object

Parameters: o - the object to be compared.

Returns :a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws: NullPointerException - if the specified object is null
ClassCastException - if the specified object's type prevents it from being compared to this object.

```
public static Integer WOOD_DAVERS = 14;

//Test
@Test (expected = NullPointerException.class)
public void TestNullComparisonObject() {
     Integer cats = null;
     WOOD_DAVERS.compare(cats);
}

@Test (expected = ClassCastException.class)
public void TestInvalidCastComparisonObject() {
     String cats = "CATS";
     WOOD_DAVERS.compareTo((Integer) cats);
}

//Test Less Than
@Test
public void TestLessThanInteger(){
Integer lessThan = 12;
Assert.equals(-1, lessThan.compareTo(WOOD_DAVERS));
}


//Test Greater Than
@Test
public void TestGreaterThanInteger(){
Integer greaterThan = 17;
Assert.equals(+1, greaterThan.compareTo(WOOD_DAVERS));
}
//Test Equality
@Test
public void TestEqualToInteger(){
Integer EQUAL = 14;
Assert.equals(0, lessThan.compareTo(WOOD_DAVERS));
```

Not bad regarding equivalence partition, could be better for boundary testing