# Topics for this Lecture

- Structural Testing (Test Coverage)
- Control Flow Graph (CFG)

# Testing Techniques

- **How to test software.**
  - How do we design tests?

| Functional Testing | Structural Testing |
|---|---|
| Called “black-box” or “specification-based” testing | Called “white-box or glass-box” testing |
| We ignore how the program is being written. | The program is the base. |
| Test based on the specification. | Test based on code. |
| Test covers as much *specified* behavior as possible. | Test covers as much *implemented* behavior as possible. |

# Why Structural (code-based) testing?

- Structural tests can be automated
- Use source code (or other structure beyond the input/output spec.) to design test cases.
- Use source code to select test cases and determine whether a set of test cases has been sufficiently thorough.
- If part of the program under test is never executed by any test case in the suite, a fault in that part cannot be exposed
    - A “part” can be a statement, function, branch…
- Structural Testing complements functional testing by including cases that may not be identified from specifications alone. Run functional tests first, then measure what is missing.

# Why Structural (code-based) testing?

```
public class Root {
 double rootOne, rootTwo;
 int numRoots;
   public Root(double a, double b, double c) {
      double q;
      double r;
      q = b * b - 4 * a * c;
      if (q > 0 && a != 0) {
      // if b^2 > 4ac there are two dinstict roots
          numRoots = 2;
          r = (double) Math.sqrt(q);
          rootOne = ((0 - b) + r) / (2 * a);
          rootTwo = ((0 - b) - r) / (2 * a);
      } else if (q == 0) { // DEFECT HERE
                numRoots = 1;
                rootOne = (0 - b) / (2 * a);
                rootTwo = rootOne;
      } else {
             // equation had no roots if b^2<4ac
             numRoots = 0;
             rootOne = -1;
             rootTwo = -1;
      }
   }
 }
```

# Why Structural (code-based) testing?

```
public class Root {
 double rootOne, rootTwo;
 int numRoots;
   public Root(double a, double b, double c) {
      double q;
      double r;
      q = b * b - 4 * a * c;
      if (q > 0 && a != 0) {
      // if b^2 > 4ac there are two dinstict roots
         numRoots = 2;
         r = (double) Math.sqrt(q);
         rootOne = ((0 - b) + r) / (2 * a);
         rootTwo = ((0 - b) - r) / (2 * a);
      } else if (q == 0) { // DEFECT HERE
                numRoots = 1;
                rootOne = (0 - b) / (2 * a);
                rootTwo = rootOne;
      } else {
             // equation had no roots if b^2<4ac
             numRoots = 0;
             rootOne = -1;
             rootTwo = -1;
      }
   }
}
```

We can test this case

and test this case

and test this case

# Why Structural (code-based) testing?

```
public class Root {
 double rootOne, rootTwo;
 int numRoots;
   public Root(double a, double b, double c) {
      double q;
      double r;
      q = b * b - 4 * a * c;
      if (q > 0 && a != 0) {
      // if b^2 > 4ac there are two dinstict roots
          numRoots = 2;
          r = (double) Math.sqrt(q);
          rootOne = ((0 - b) + r) / (2 * a)
          rootTwo = ((0 - b) - r) / (2 * a);
      } else if (q == 0) { // DEFECT HERE
                numRoots = 1;
                rootOne = (0 - b) / (2 * a)
                rootTwo = rootOne;
      } else {
             // equation had no roots if b^2<4ac
             numRoots = 0;
             rootOne = -1;
             rootTwo = -1;
      }
   }
 }
```

**(a, b, c)= (3, 4, 1)**

**(a, b, c)= (0, 0, 1)**

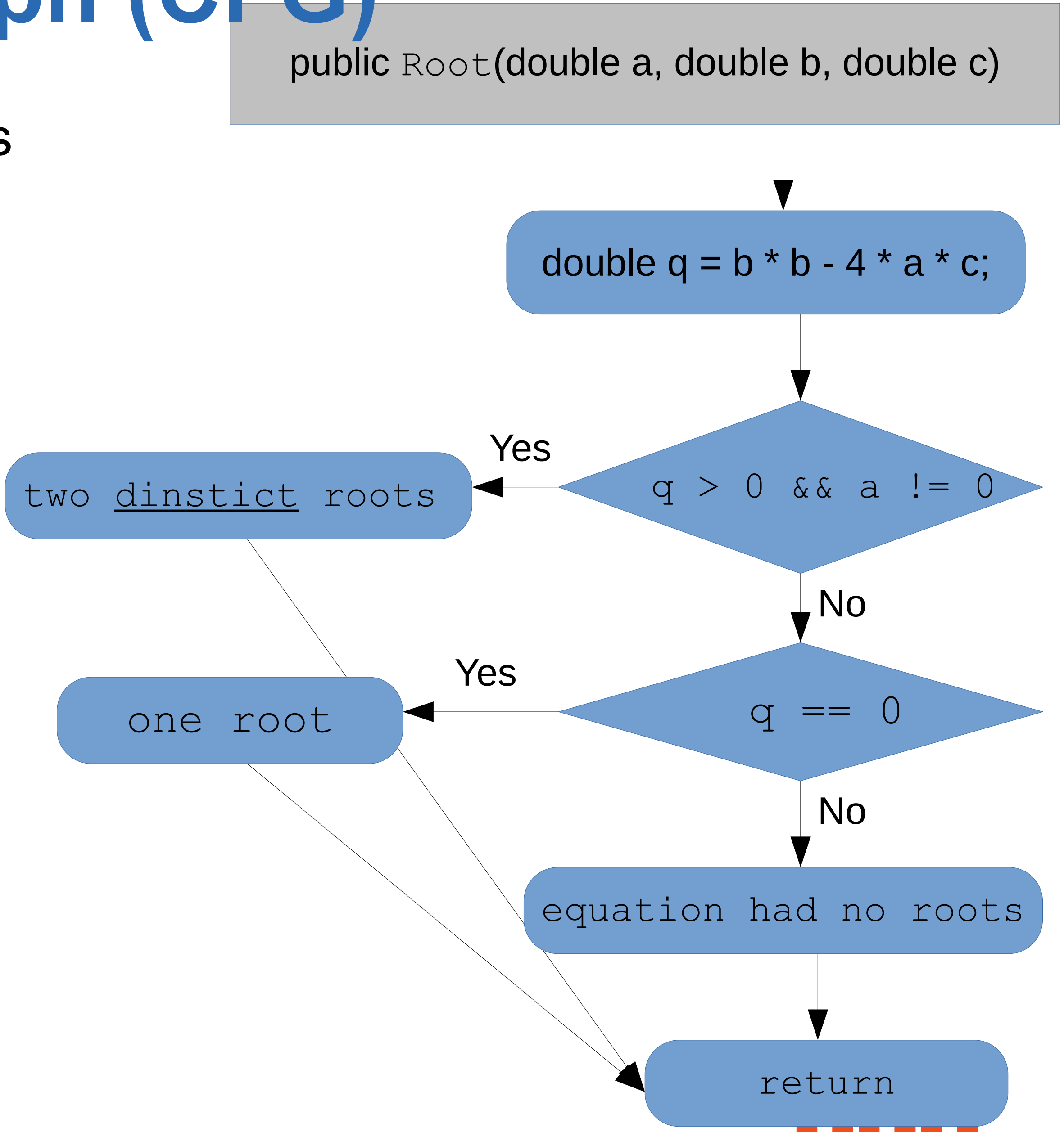
**(a, b, c)= (3, 2, 1)**

# Some thoughts and observations!

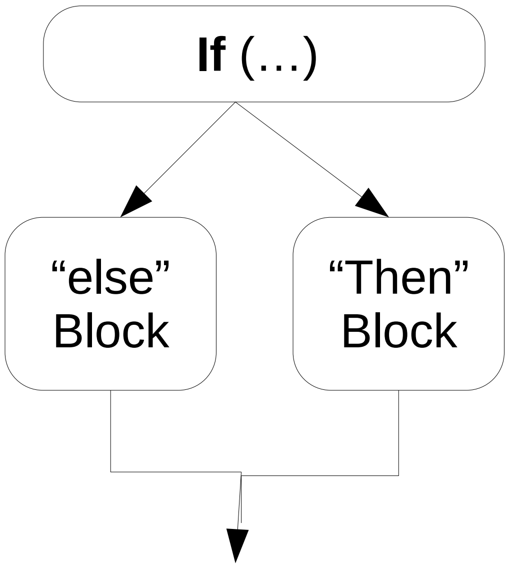
- Testing can reveal a fault only when execution of the faulty statement causes a failure.
- For example, the fault in the statement at line 16 in the Root class, could be revealed only with test cases in the which the input contains a=0, b=0, c=1
- A program has not been adequately tested if some of its parts have not been executed.
- Execution of a faulty statement may not always result a failure.
  - For example, a test case a=1, b= 2 and c=1
- Finding appropriate input values is a challenge.

# Control Flow Graph (CFG)

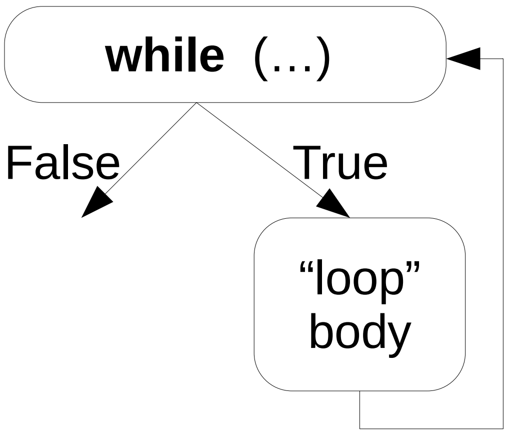
- **A control flow graph (CFG)** is a directed graph and represents paths of program execution
- **Nodes** are basic blocks and sequences of statements with a single entry and single exit point
- **Directed Edges** represent the ***possibility*** that the program execution proceeds from the end of one node to the beginning of another



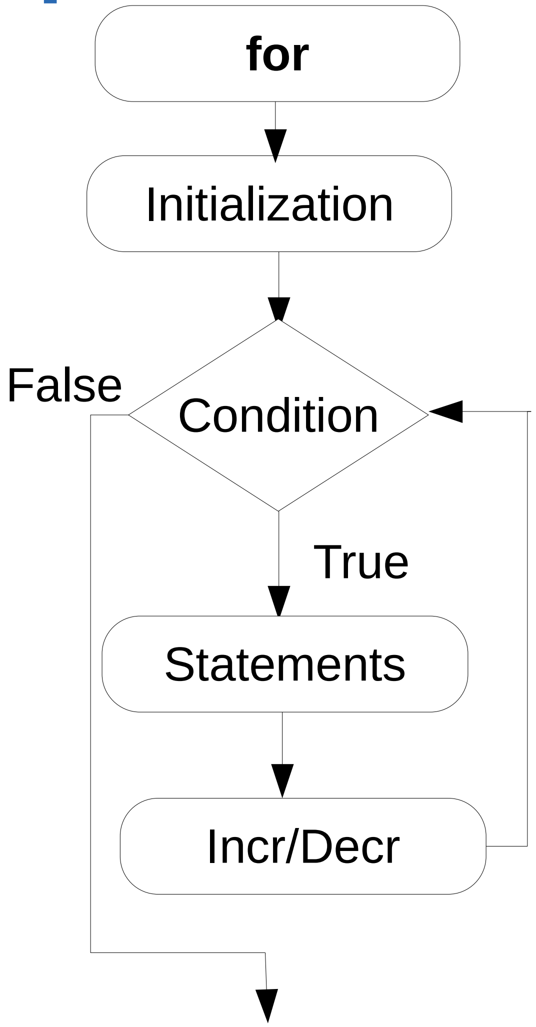
# Control Flow Graph (CFG) Representations:



```
if (condition)
        Then-Block
else
        Else-Block
```



```
While(condition){
        Body
}
```



```
For( initialization; condition; incr/decr){
        Statements
}
```

# Structural Testing

- The CFG can serve as an adequacy criterion for test cases
- The more parts are covered (executed), the higher the chance of a test to uncover a defect
- “parts” can be: nodes, edges, paths, conditions, blocks, statements,....
- Brings us to the idea of ***coverage***

References:

Young, Michal, and Mauro Pezze. "Software Testing and Analysis: Process, Principles and Techniques." (2005).Chapters 5, 12

http://classes.engr.oregonstate.edu/eecs/summer2015/cs362-002/

https://www.st.cs.uni-saarland.de/