# COMP1008
# Testing





## Perfection

- Do your programs work perfectly?
- Are you perfect?

No!!






## Perfection, or Lack of It

- No program is perfect.
- Programs will have errors.
- Often see quotes like:
  - "On average program code has 10 errors per 1000 lines…"






## Two V's

- Verification
  - "Are we building the system right?"
  - Testing code.
- Validation
  - "Are we building the right system?"
  - Testing behaviour against requirements.






## Testing

- Testing is really about trying to find bugs.
  - By actually running the code.
- Testing *cannot* show your program will always work properly — only the deluded believe this can be done!
- But it can remove sufficient bugs to make your program "good enough".
- Testing allows you to gain confidence in your code.






## Testing and Proof

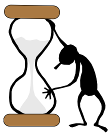
- To prove something we must show:

$$\forall x \bullet P(x)$$

- This implies we have to explore every possible state a program can be in.
- But...

## Testing and Proof (2)

- Take, for example, the sqrt method.
- To “prove” it works we would have to call it with every possible floating point value.
- So if $2^{64}$ = 18446744073709551616 ≈ $10^{19}$ and we do $10^{6}$ operations per second then this is $10^{13}$ seconds, which is *$10^{6}$ years*.








## Testing and Proof (3)

- The philosophy behind all testing should be the finding of errors.
  - Need to identify tests most likely to uncover errors.
- No “proof” can be constructed that no errors exist.
  - Just have the situation that no tests find any errors.
  - The next test you add may find an error…






## Making testing sqrt manageable

- We still have the problem of $10^{19}$ possible values that could give us an error.
- So, we need to focus on floating point values that:
  - Are representative of typical input values.
  - Might cause an error.
- But how do you find them?






## Testing the sqrt Method

- We can start by studying the domain of the method.
  - sqrt partitions the floating point numbers into 3 sets:

    x < 0

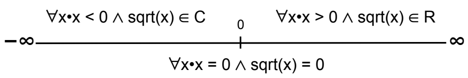
    x = 0

    x > 0
- And by looking at the method to see what the code does and where potential errors might be.






## Equivalence Classes




- Select values that are representative of the distinct classes of input values.
  - x >= 0 looks OK to test.
  - x < 0 is a problem as we need to represent complex numbers...
    - Ignore it.
    - Return an error value.






## Boundary Conditions

- Want to also focus on boundary conditions:
  - 0.0, 1.0, 2.0, 3.0
  - MIN_DOUBLE, MAX_DOUBLE
  - .3, .33, .333, etc.
  - 0.0000000000001, 0.11111111111111, etc.
  - -0.0, -1.0
  - numbers that might cause under/overflow in sqrt algorithm.
- Can use the code itself to help identify boundaries.
  - If and loop statements.
  - Maths expressions.
- But what level of accuracy (decimal places)?

## Running Tests

- Select representatives from each of the sets to construct the test data set.
- Create a test harness — a program to call sqrt with the elements of the data set.
  - Or use a test framework.
- Run the program and compare the results with what was expected (which you need to work out some other way!).






## Most Basic Approach (not recommended in general)

```
public void testSqrt()
{
  System.out.println("sqrt(1.0) = " + Math.sqrt(1.0)) ;
  System.out.println("sqrt(2.0) = " + Math. sqrt(2.0)) ;
  System.out.println("sqrt(3.0) = " + Math. sqrt(3.0)) ;
  System.out.println("sqrt(10.0) = " + Math. sqrt(10.0)) ;
  System.out.println("sqrt(100.0) = " + Math. sqrt(100.0)) ;
  System.out.println("sqrt(1000.0) = " + Math. sqrt(1000.0)) ;
  System.out.println("sqrt(0.0) = " + Math. sqrt(0.0)) ;
  System.out.println("sqrt(-1.0) = " + Math. sqrt(-1.0)) ;
  // etc…
}
```






## Most Basic Approach (2)

sqrt(1.0) = 1.0
sqrt(2.0) = 1.4142135623730951
sqrt(3.0) = 1.7320508075688772
sqrt(10.0) = 3.1622776601683795
sqrt(100.0) = 10.0
sqrt(1000.0) = 31.622776601683793
sqrt(0.0) = 0.0
sqrt(-1.0) = NaN ?!






## NaN?

- Not a Number.
- Value used when result of floating point operation cannot be represented.
- This version of sqrt will return NaN for any argument < 0.
- For this equivalence class, have “solved” problem by updating specification of method.
  - Implies method must check for input less than zero.






## Using a different sqrt method implementation

sqrt(1.0) = 1.0
sqrt(2.0) = 1.4142135623746899
sqrt(3.0) = 1.7320508100147274
sqrt(10.0) = 3.162277665175675
sqrt(100.0) = 10.000000000139897
sqrt(1000.0) = 31.622776601684336
sqrt(0.0) = NaN !!!
sqrt(-1.0) = NaN

Different results!
Which are correct?






## But

- This is quickly going to get boring and error prone.
  - Manual checking process.
  - OK for 10 tests,
  - Tedious for 100 tests,
  - Mind-numbing for 1000 tests.
  - Mistakes will be made.
- Need an automated approach.

## Automate

- Write a test harness program that reads data from data structure or file.
- Get program to run tests and check the results.






```
public void test(double input, double expected, double delta)
{
  double sqrtValue = Math.sqrt(input);
  double diff = Math.abs((sqrtValue - expected));

  if (diff > delta)
  {
    System.out.print("Invalid result for sqrt("+input+"),");
    System.out.print(" expected: " + expected);
    System.out.println(", got: " + sqrtValue);
  }
}
```

Note how doubles are compared.






```
public void testDataSet(double[][] data)
{
  for (int i = 0 ; i < data.length ; i++)
  {
   test(data[i][0],data[i][1],0.00001);
  }
}
public void run()
{
 double[][] d1 = new double[][]
 {{1.0,1.0}, {2.0,1.4142135}, {3.0,1.7320508},
  {10.0,3.1622776}, {100.0,10.0}, {0.1,0.316227},
  {0.0,0.0}, {-1.0,Double.NaN}
 };
 testDataSet(d1);
}
```

Or read data from a file.

But can do better than this using a proper testing framework






## So, just how do you write sqrt anyway...

```
public static double sqrt(final double x) {
 final double precision = 0.0000001;
 if (x < 0.0) {
  return Double.NaN ;
 }
 if (x <= precision) {
  return 0.0 ;
 }
 double a = 1.0 ;
 while (Math.abs(a*a - x) > precision) {
  a = (a + x/a)/2 ;
 }
 return a ;
}
```

Newton Raphson approximation

Obvious boundary conditions

Potential overflow






## Summary

- Test to find errors.
- Use a test harness program.
  - Let it do the repetitive hard work.
- Do enough tests to be confident in your code.






## What does this mean for your code?

- Each method should be tested.
  - Check value returned for given parameter values.
  - For a void method, call a second method to observe the results.
    - e.g., adding an object to a data structure using void add(...), results in the size increasing by one.
- Need accurate specification of what method is meant to do.
- Use method implementation to focus on potential problems.
  - e.g., loop counting one too many/few times.