

- Types of testing
- There are lots of different types of testing that is done on computer systems and programs, depending on the size and complexity, below are some
  - Unit testing – testing that a unit of code functions properly
  - Functional testing – testing to verify that the code functions as specified
  - Integration testing – testing that the various pieces of code function together when they are integrated together
  - Regression testing – testing to verify that the latest code updates do not break the existing code
  - Performance testing – testing to verify that the code is able to perform under various conditions – speed, stability, reliability, resources utilization
  - System testing – testing that the entire system performs correctly in the intended environment
  - Verification testing – does the code do what the requirements specify
  - Validation testing – does the code do what we want it to do
  - And many more

- Types of testing (cont)
- If you take a class in software engineering, you will become familiar with the entire software development process (below is some of it)
  - Requirements development
    - High level requirements
    - Performance requirements
    - Detailed requirements
  - Software design
    - High level design
    - Detailed design
    - Requirements mapping
    - Software development
  - Software verification
    - Develop tests for verification of requirements
    - Develop test data
- If you have a chance to take a software engineering class, I would highly recommend it

- JUnit test
- JUnit is a simple framework to write repeatable tests
  - From <https://junit.org/junit4/>
- JUnit 5 is the current version, <https://junit.org/junit5/>
  - JUnit 5 is integrated into eclipse
  - JUnit 5 tests can also be compiled & run from the command line
- JUnit uses assertions to verify that a result is what it is expected to be
- There are various assertion methods to verify
  - An array matches an expected result array
  - A String result matches the expected String result
  - An object is not null
  - Two objects are not the same
  - An object is null
  - That a list contains certain items
  - There are many more

- JUnit test (cont)
- The general idea is that once we have implemented a class, we create a JUnit test to verify that the class is functioning properly
- We are going to create a significant number of JUnit tests in lab 6 to test program 4
  - Some of the tests we will give you for free, and some you will implement yourself
- In lab 6 we will create tests for the following (19 tests)
  - `getNumberOfEdges()` & `getNumberOfVertices()`
  - `addVertex()` & `addEdge()`
  - `toString()`
  - `isDirected()` for various constructor calls (three of them)
  - `isConnected()` for the four sample graphs
  - The three types of `GraphExceptions` (five tests)
    - Duplicate vertices, duplicate edges, invalid edge

- JUnit test (cont)
- We will start with using JUnit5 with eclipse
- First we need to create a project, along with a class
- Since we are going to be doing this for program 4, we might as well do it for that
  - I started with a project named “program4”, and a class named “garrison\_Graph”
  - I dragged and dropped Edge.java, GraphException.java, and ConnectedGraphFunctions.java into the “program4” folder that garrison\_Graph.java is in
  - If you haven't done anything with program 4 yet, then get the required functions populated with no functionality for the “void” methods and returning false for the “boolean” methods, returning 0 for the “int” methods, and returning “” for the “String” methods
  - Remember to include “implements ConnectedGraphFunctions” in the class definition line

# • JUnit test (cont)

- Once we have the basics of the project created
- On my laptop I had to tell eclipse to use the Java 17 compiler and runtime and set the compiler compliance level to 16
  - Otherwise it did not like the Edge record
  - I did not have this problem on remote140
- Select your graph class and right/double click on it and choose “New JUnit Test Case”
- Select “New JUnit Jupiter test” (this is JUnit5)
- Get rid of the package name
- It should have filled in the name with your graph classes name with “Test” appended to it
- Select @BeforeAll and @BeforeEach
- Class under test should be your graph classes name
- If you select next it will give you a list of methods to create tests for
  - Select addVertex, addEdge, toString
  - We will add the rest manually

- JUnit test (cont)
- Select Finish and OK when asked about adding JUnit 5 to the build path
- Add “**import** org.junit.jupiter.api.Assertions;” to the list of imports (if it is not there)
- We should have something that looks like below

```
1 import static org.junit.jupiter.api.Assertions.*;
2
3 import org.junit.jupiter.api.BeforeAll;
4 import org.junit.jupiter.api.BeforeEach;
5 import org.junit.jupiter.api.Test;
6 import org.junit.jupiter.api.Assertions;
7
8 class garrison_GraphTest {
9
10     @BeforeAll
11     static void setUpBeforeClass() throws Exception {
12     }
13
14     @BeforeEach
15     void setUp() throws Exception {
16     }
17
18     @Test
19     void testAddVertex() {
20         fail("Not yet implemented"); // TODO
21     }
22
23     @Test
24     void testAddEdge() {
25         fail("Not yet implemented"); // TODO
26     }
27
28     @Test
29     void testToString() {
30         fail("Not yet implemented"); // TODO
31     }
32
33 }
```

} Should be executed once prior to the tests

} Should be executed prior to each test, may be named setUpBeforeEach()

# • JUnit test (cont)

- We aren't going to use `@BeforeAll`, but it is not hurting anything by being there
- In the `@BeforeEach` method, add a line to instantiate a copy of your graph class and also include a declaration statement for your graph class prior to the `@BeforeAll`
  - Mine looks like the below
  - If a graph object that is undirected works for a tests, then the “g = new garrison\_Graph();” will have already instantiate one for each of the tests
  - The `@BeforeEach` method is executed before each of the tests

```
class garrison_GraphTest
{
    garrison_Graph g;

    @BeforeAll
    static void setUpBeforeClass() throws Exception
    {
    }

    @BeforeEach
    void setUp() throws Exception
    {
        g = new garrison_Graph();
    }
}
```



- JUnit test (cont)

- Let's add a test for isDirected() for the zero parameter constructor
  - The graph object instantiated in the before each should work for this test, so all we need to do is check that g.isDirected() returns false
  - The assertEquals() method that compares two boolean values will work here, so we can do the following
    - Or we can use the assertFalse() method
  - Add a new @Test as below (you can add either of them)

```
// test that the zero parameter construct makes the graph undirected
@Test
void testIsDirectedForUndirectedGraph()
{
    assertEquals(false, g.isDirected());
}
```

```
// test that the zero parameter construct makes the graph undirected
@Test
void testIsDirectedForUndirectedGraph()
{
    assertFalse(g.isDirected());
}
```

- JUnit test (cont)
- We can do similarly for the one parameter constructor
- Copy the last test, and make two new versions, one for a true test and one for a false test
- I end up with the following

```
// test that the one parameter construct makes the graph directed with true parameter
@Test
void testIsDirectedForDirectedGraph()
{
    g = new garrison_Graph(true);
    assertTrue(g.isDirected());
}

// test that the one parameter construct makes the graph undirected with false parameter
@Test
void testIsDirectedForUndirectedGraph2()
{
    g = new garrison_Graph(false);
    assertFalse(g.isDirected());
}
```

- JUnit test (cont)
- Now we want to test our `getNumberOfVertices()` method
- For this test, let's add vertices for 0, 1, ..., 99 with each time that we add a vertex we will check that the result of `getNumberOfVertices()` is correct
- For this test we will use `assertArrayEquals()`, to verify that the expected result to each call to `getNumberOfVertices()` returns the correct result
- For the test we allocate two int arrays of size 101, one will contain the expected results of `getNumberOfVertices()` and the other will contain the actual results of each call to `getNumberOfVertices()`
- The value at index 0 will be the number prior to adding any vertices, and then the value at index  $i$ ,  $i = 1, 2, \dots, 100$  will be the number of vertices after adding  $i$  vertices (so the value at index  $i$  is  $i$ )

- JUnit test (cont)
- Now we want to test our getNumberOfVertices() method (cont)
- Below is a copy of my code

```
// add vertices 0 - 99, verify there are the correct number of vertices
// each time one is added
@Test
void testGetNumberOfVertices()
{
    int[] getNumberOfVertices = new int[101];
    int[] expectedResultGetNumberOfVertices = new int[getNumberOfVertices.length];
    getNumberOfVertices[0] = g.getNumberOfVertices();
    expectedResultGetNumberOfVertices[0] = 0;
    for( int i = 1; i < getNumberOfVertices.length; i++ )
    {
        expectedResultGetNumberOfVertices[i] = i;
        try
        {
            g.addVertex(i-1);
            getNumberOfVertices[i] = g.getNumberOfVertices();
        }
        catch(GraphException e)
        {
        }
    }

    assertEquals(getNumberOfVertices, expectedResultGetNumberOfVertices);
}
```

} Add vertices 0, 1, 2, ..., 100

- JUnit test (cont)
- Now we want to test our `getNumberOfEdges()` method
- For this test, let's add vertices for 0, 1, ..., 100 and then add edges (i, i+1) for  $i = 0, \dots, 99$  with each time that we add an edge we will check that the result of `getNumberOfEdges()` is correct
- For this test we will use `assertArrayEquals()`, to verify that the expected result to each call to `getNumberOfVertices()` returns the correct result
- For the test we allocate two int arrays of size 101, one will contain the expected results of `getNumberOfEdges()` and the other will contain the actual results of each call to `getNumberOfEdges()`
- The value at index 0 will be the number prior to adding any edges, and then the value at index  $i$ ,  $i = 1, 2, \dots, 100$  will be the number of edges after adding  $i$  vertices (so the value at index  $i$  is  $i$ )

- JUnit test (cont)
- Now we want to test our `getNumberOfEdges()` method (cont)
- Below is a copy of my code

```
// add vertices 0 - 100, then add edges (i,i+1) i = 0, ..., 99,
// verify the number of edges is correct after each one is added
```

```
@Test
```

```
void testGetNumberOfEdges()
```

```
{
```

```
    int[] getNumberOfEdges = new int[101];
```

```
    int[] expectedResultGetNumberOfEdges = new int[getNumberOfEdges.length];
```

```
    getNumberOfEdges[0] = g.getNumberOfEdges();
```

```
    expectedResultGetNumberOfEdges[0] = 0;
```

```
    for( int i = 0; i < getNumberOfEdges.length; i++ )
```

```
    {
```

```
        try
```

```
        {
```

```
            g.addVertex(i);
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

Add vertices 0,  
1, 2, ..., 100

```
    for( int i = 1; i < getNumberOfEdges.length; i++ )
```

```
    {
```

```
        expectedResultGetNumberOfEdges[i] = i;
```

```
        try
```

```
        {
```

```
            g.addEdge(i-1, i);
```

```
            getNumberOfEdges[i] = g.getNumberOfEdges();
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

Add edges (i, i+1), i =  
0, 1, 2, ..., 99

```
    assertEquals(getNumberOfEdges, expectedResultGetNumberOfEdges);
```

```
}
```

- JUnit test (cont)

- Now we want to test our `addVertex()` method
- For this test, let's add vertices for 0, 1, ..., 100 with each time that we add a vertex we will check that the result of `getNumberOfVertices()` is correct
- For this test we will use `assertArrayEquals()`, to verify that the expected result to each call to `getNumberOfVertices()` returns the correct result
- For the test we allocate two int arrays of size 101, one will contain the expected results of `getNumberOfVertices()` and the other will contain the actual results of each call to `getNumberOfVertices()`
- The value at index 0 will be the number prior to adding any vertices, and then the value at index  $i$ ,  $i = 1, 2, \dots, 100$  will be the number of vertices after adding  $i$  vertices (so the value at index  $i$  is  $i$ )
- We add two copies of each vertex, to ensure that we aren't adding duplicates
  - Note we execute `getNumberOfVertices()` prior to adding the duplicate vertex, since if we added the duplicate vertex prior to `getNumberOfVertices()`, `getNumberOfVertices()` would not get executed since the exception would cause the code to transfer control to the catch statement



- JUnit test (cont)
- Now we want to test our addVertex() method (cont)
- Below is what it should look like

```
// add 100 vertices twice and verify that the number of vertices
// is correct each time one is added
@Test
void testAddVertex()
{
    int[] numberOfVertices = new int[101];
    int[] expectedNumberOfVertices = new int[numberOfVertices.length];
    numberOfVertices[0] = g.getNumberOfVertices();
    expectedNumberOfVertices[0] = 0;
    for( int i = 0; i < numberOfVertices.length-1; i++ )
    {
        expectedNumberOfVertices[i+1] = i+1;
        try
        {
            g.addVertex(i);
            numberOfVertices[i+1] = g.getNumberOfVertices();
            g.addVertex(i);
        }
        catch(Exception e)
        {
        }
    }
    assertEquals(numberOfVertices, expectedNumberOfVertices);
}
```

Try adding a second copy of the vertex, this should throw a GraphException and not be added



- JUnit test (cont)
- Now we want to test our addEdge() method
- For this test, let's add vertices for 0, 1, ..., 100 and edges (i, i+1) for i = 0, ..., 99
- Once again, we allocate two arrays of size 101, one for the expected number of edges and one for the actual number of edges
- In this test we add two copies of each edge, to test that we are not adding duplicate edges
- We also add the duplicate edge after executing getNumberOfEdges(), for the same reason that we added the duplicate vertex after getting the number of vertices in the addVertex test

- JUnit test (cont)
- Now we want to test our addEdge() method (cont)
- Below is my code

```
// add 101 vertices and 100 edges twice and verify that the number of edges is correct
// each time an edge is added
@Test
void testAddEdge()
{
    int[] numberOfEdges = new int[101];
    int[] expectedNumberOfEdges = new int[numberOfEdges.length];
    numberOfEdges[0] = g.getNumberOfEdges();
    expectedNumberOfEdges[0] = 0;
    for( int i = 0; i < numberOfEdges.length-1; i++ )
    {
        expectedNumberOfEdges[i+1] = i+1;
        try
        {
            if( i == 0 )
            {
                g.addVertex(i);
            }
            g.addVertex(i+1);
            g.addEdge(i, i+1);
            numberOfEdges[i+1] = g.getNumberOfEdges();
            g.addEdge(i, i+1);
        }
        catch(Exception e)
        {
        }
    }

    assertEquals(numberOfEdges, expectedNumberOfEdges);
}
```

Try adding a second copy of the edge, this should throw a GraphException and not be added

- JUnit test (cont)
- Now we want to test that our GraphException is thrown at the appropriate times
  - When a duplicate vertex is attempted to be added
  - When a duplicate edge is attempted to be added
    - Add (u,v) twice
    - Add (u,v) and (v,u) for an undirected graph
  - When an invalid edge is added
  - And ensure that one is not thrown for a directed graph when edges (u,v) and (v,u) are added, since they are not duplicates

- JUnit test (cont)
- Testing for a GraphException for duplicate vertices
- My code is below, the “( ) -> g.addVertex(0)” is a lambda expression, which we have not discussed yet, but will later in the semester, it is telling JUnit test to apply the command “g.addVertex(0)”

```
// try to add vertex 0 twice, verify there is an exception on the second one
@Test
void testGraphExceptionForDuplicateVertex()
{
    try
    {
        g.addVertex(0);
    }
    catch(GraphException e)
    {
    }
    assertThrows(GraphException.class, () -> g.addVertex(0));
}
```

- JUnit test (cont)
- Testing for a GraphException for duplicate edges
- My code is below, here we are attempting to add the edge (0, 1) twice

```
// try to add edge (0, 1) twice, verify there is an exception on the second one
@Test
void testGraphExceptionForDuplicateEdge()
{
    try
    {
        g.addVertex(0);
        g.addVertex(1);
        g.addEdge(0, 1);
    }
    catch(GraphException e)
    {
    }
    assertThrows(GraphException.class, () -> g.addEdge(0, 1));
}
```

- JUnit test (cont)
- Testing for a GraphException for duplicate edges
- My code is below, here we are attempting to add the edge (0, 1) and then the edge (1, 0)

```
// try to add edge (1, 0) after adding edge (0, 1),  
// verify there is an exception when adding (1, 0)  
@Test  
void testGraphExceptionForDuplicateEdge2()  
{  
    try  
    {  
        g.addVertex(0);  
        g.addVertex(1);  
        g.addEdge(0, 1);  
    }  
    catch(GraphException e)  
    {  
    }  
    assertThrows(GraphException.class, () -> g.addEdge(1, 0));  
}
```

- JUnit test (cont)
- Testing for a GraphException for an invalid edge
- My code is below, here we are attempting to add the edge (0, 1) without ever defining vertices 0 or 1

```
// try to add edge (0, 1) without adding vertices 0 & 1,  
// verify there is an exception  
@Test  
void testGraphExceptionForInvalidEdge()  
{  
    assertThrows(GraphException.class, () -> g.addEdge(0, 1));  
}
```

- JUnit test (cont)

- Testing for a GraphException while adding a reversed edge for a directed graph, which should not throw an exception
- My code is below, here we add edges (0, 1) and (1, 0) and verify that we have two vertices, meaning that we did not throw a GraphException
  - You don't need to implement this test, the one on the next slide is “better”

```
// in a directed graph verify that we can add edge (0, 1) and (1, 0)
@Test
void testGraphExceptionForDuplicateEdge3()
{
    g = new garrison_Graph(true);
    try
    {
        g.addVertex(0);
        g.addVertex(1);
        g.addEdge(0, 1);
        g.addEdge(1, 0);
    }
    catch(GraphException e)
    {
    }
    assertEquals(2, g.getNumberOfEdges());
}
```



- JUnit test (cont)
- Here's a second version of the last test
- When I originally wrote the test, I did not know there was an “assertDoesNotThrow()” assertion
- Below is a second version of the last test using “assertDoesNotThrow()”
- As you use JUnit test more you will notice that there are often multiple ways to perform the same test

```
// in a directed graph verify that we can add edge (0, 1) and (1, 0)
@Test
void testGraphExceptionForDuplicateEdge3b()
{
    g = new garrison_Graph(true);
    try
    {
        g.addVertex(0);
        g.addVertex(1);
        g.addEdge(0, 1);
        assertDoesNotThrow(() -> g.addEdge(1, 0));
    }
    catch(GraphException e)
    {
    }
}
```

- JUnit test (cont)
- We next want to test the isConnected() method
- For this test, we have a lot that we need to do
- We need to
  - Specify a graph, that is, define the vertices & edges
  - Add the vertices and edges to the graph
- My implementation required a large amount of code, since it
  - Parses the vertices and edges from a String that defines them
  - And then adds them to the graph
- Since we have four sample graphs, the code is replicated four times
- The four sample graphs are
  - sample\_directed\_graph\_1.txt – this one is connected
  - sample\_directed\_graph\_2.txt – this one is not connected
  - sample\_undirected\_graph\_1.txt – this one is connected
  - sample\_undirected\_graph\_2.txt – this one is not connect

## • JUnit test (cont)

- Since we need access to the sample graphs within the test methods, we define them in the test class outside of the methods
  - And the expected result of the `getConnectedSet()` for `sample_undirected_graph_2.txt`
- Here's how I have them defined, at the top of my test class prior to any methods

```
// sample_directed_graph_1.txt
String directedAndConnectedVertices = "{1,3,2,4,5,1,2}";
String directedAndConnectedEdges = "{(1,4),(2,1),(2,3),(3,5),(4,5),(5,2)}";

// sample_directed_graph_2.txt
String directedAndNotConnectedVertices = "{1,3,2,4,5,1,2}";
String directedAndNotConnectedEdges = "{(1,4),(2,1),(2,3),(3,5),(4,5),(0,1),(0,7),(1,7)}";

// sample_undirected_graph_1.txt
String undirectedAndConnectedVertices = "{0,1,3,2,4,5,6,7,8,9,0,2}";
String undirectedAndConnectedEdges = "{(0,5),(1,7),(2,4),(3,6),(4,9),(5,8),(6,9),(7,9),(8,9),(5,0)}";

// sample_undirected_graph_2.txt
String undirectedAndNotConnectedVertices = "{0,1,3,2,4,5,6,7,8,9,0,2}";
String undirectedAndNotConnectedEdges = "{(0,5),(1,7),(2,4),(4,9),(5,8),(6,9),(7,9),(8,9),(5,0)}";

// expected output for getConnectedSet(2) for sample_undirected_graph_2.txt
int[] undirectedAndNotConnectedSubset = new int[] {0,1,2,4,5,6,7,8,9};
```

- JUnit test (cont)

- Here's the code for sample\_undirected\_graph\_1.txt

```
// test isConnected() recognizes sample_undirected_graph_1.txt is connected
```

```
@Test
```

```
void testIsConnectedUndirectedAndConnected()  
{
```

undirected graph

```
    g = new garrison_Graph(false);
```

```
    java.util.StringTokenizer st = new java.util.StringTokenizer(undirectedAndConnectedVertices, "{},");
```

```
    while( st.hasMoreTokens() )
```

```
    {  
        int newVertex = Integer.parseInt(st.nextToken());
```

```
        try
```

```
        {
```

```
            g.addVertex(newVertex);
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

Parse the vertices and add them to the graph

```
    st = new java.util.StringTokenizer(undirectedAndConnectedEdges, "{}");
```

```
    String inn = st.nextToken();
```

```
    st = new java.util.StringTokenizer(inn, "(),");
```

```
    while( st.hasMoreTokens() )
```

```
    {  
        int from = Integer.parseInt(st.nextToken());  
        int to = Integer.parseInt(st.nextToken());
```

```
        try
```

```
        {
```

```
            g.addEdge(from, to);
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

Parse the edges and add them to the graph

```
    assertEquals(true, g.isConnected());
```

```
}
```

- JUnit test (cont)

- Here's the code for sample\_undirected\_graph\_2.txt

```
// test isConnected() recognizes sample_undirected_graph_2.txt is not connected
```

```
@Test
```

```
void testIsConnectedUndirectedAndNotConnected()
```

```
{
```

```
    g = new garrison_Graph(false);
```

undirected graph

```
    java.util.StringTokenizer st = new java.util.StringTokenizer(undirectedAndNotConnectedVertices, "{},");
```

```
    while( st.hasMoreTokens() )
```

```
    {
```

```
        int newVertex = Integer.parseInt(st.nextToken());
```

```
        try
```

```
        {
```

```
            g.addVertex(newVertex);
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

Parse the vertices and add them to the graph

```
    st = new java.util.StringTokenizer(undirectedAndNotConnectedEdges, "{}");
```

```
    String inn = st.nextToken();
```

```
    st = new java.util.StringTokenizer(inn, "(),");
```

```
    while( st.hasMoreTokens() )
```

```
    {
```

```
        int from = Integer.parseInt(st.nextToken());
```

```
        int to = Integer.parseInt(st.nextToken());
```

```
        try
```

```
        {
```

```
            g.addEdge(from, to);
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

Parse the edges and add them to the graph

```
    assertEquals(false, g.isConnected());
```

```
}
```



- JUnit test (cont)

- Here's the code for sample\_directed\_graph\_1.txt

```
// test isConnected() recognizes sample_directed_graph_1.txt is connected
```

```
@Test
```

```
void testIsConnecteDirectedAndConnected()
```

```
{
```

```
    g = new garrison_Graph(true);
```

directed graph

```
    java.util.StringTokenizer st = new java.util.StringTokenizer(directedAndConnectedVertices, "{},");
```

```
    while( st.hasMoreTokens() )
```

```
    {
```

```
        int newVertex = Integer.parseInt(st.nextToken());
```

```
        try
```

```
        {
```

```
            g.addVertex(newVertex);
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

```
    st = new java.util.StringTokenizer(directedAndConnectedEdges, "{}");
```

```
    String inn = st.nextToken();
```

```
    st = new java.util.StringTokenizer(inn, "(),");
```

```
    while( st.hasMoreTokens() )
```

```
    {
```

```
        int from = Integer.parseInt(st.nextToken());
```

```
        int to = Integer.parseInt(st.nextToken());
```

```
        try
```

```
        {
```

```
            g.addEdge(from, to);
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

```
    assertEquals(true, g.isConnected());
```

```
}
```

Parse the vertices and add them to the graph

Parse the edges and add them to the graph

- JUnit test (cont)

- Here's the code for sample\_directed\_graph\_2.txt

```
// test isConnected() recognizes sample_directed_graph_2.txt is not connected
```

```
@Test
```

```
void testIsConnecteDirectedAndNotConnected()
```

```
{
```

```
    g = new garrison_Graph(true);
```

```
    java.util.StringTokenizer st = new java.util.StringTokenizer(directedAndNotConnectedVertices, "{},");
```

```
    while( st.hasMoreTokens() )
```

```
    {
```

```
        int newVertex = Integer.parseInt(st.nextToken());
```

```
        try
```

```
        {
```

```
            g.addVertex(newVertex);
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

```
    st = new java.util.StringTokenizer(directedAndNotConnectedEdges, "{}");
```

```
    String inn = st.nextToken();
```

```
    st = new java.util.StringTokenizer(inn, "(),");
```

```
    while( st.hasMoreTokens() )
```

```
    {
```

```
        int from = Integer.parseInt(st.nextToken());
```

```
        int to = Integer.parseInt(st.nextToken());
```

```
        try
```

```
        {
```

```
            g.addEdge(from, to);
```

```
        }
```

```
        catch(GraphException e)
```

```
        {
```

```
        }
```

```
    }
```

```
    assertEquals(false, g.isConnected());
```

```
}
```

directed graph

Parse the vertices and add them to the graph

Parse the edges and add them to the graph

# • JUnit test (cont)

- Here's the code isConnectedSet() for sample\_directed\_graph\_2.txt

// test getConnectedSet() returns correct connected subset for sample\_directed\_graph\_2.txt

@Test

void testGetConnectedSetDirectedAndNotConnected()

{

directed graph

```
g = new garrison_Graph(true);
int startingVertex = -1;
java.util.StringTokenizer st = new java.util.StringTokenizer(directedAndNotConnectedVertices, "{}");
while( st.hasMoreTokens() )
{
    int newVertex = Integer.parseInt(st.nextToken());
    startingVertex = newVertex;

    try
    {
        g.addVertex(newVertex);
    }
    catch(GraphException e)
    {
    }
}
```

Parse the vertices and add them to the graph

```
st = new java.util.StringTokenizer(directedAndNotConnectedEdges, "{}");
String inn = st.nextToken();
st = new java.util.StringTokenizer(inn, "()");
while( st.hasMoreTokens() )
{
    int from = Integer.parseInt(st.nextToken());
    int to = Integer.parseInt(st.nextToken());

    try
    {
        g.addEdge(from, to);
    }
    catch(GraphException e)
    {
    }
}
```

Parse the edges and add them to the graph

```
java.util.HashSet<Integer> connectedSubset = g.getConnectedSet(startingVertex);
boolean correctSize = connectedSubset.size() == 1;
boolean containsStartingVertex = connectedSubset.contains(2);

assertTrue(correctSize && containsStartingVertex);
}
```

Get the connected set containing vertex 2 and check if it is correct



# • JUnit test (cont)

- Here's the code isConnectedSet() for sample\_undirected\_graph\_2.txt

// test getConnectedSet() returns correct connected subset for sample\_undirected\_graph\_2.txt

@Test

void testGetConnectedSetUndirectedAndNotConnected()

{

undirected graph

```
g = new garrison_Graph(false);
int startingVertex = -1;
java.util.StringTokenizer st = new java.util.StringTokenizer(undirectedAndNotConnectedVertices, "{}");
while( st.hasMoreTokens() )
{
    int newVertex = Integer.parseInt(st.nextToken());
    startingVertex = newVertex;

    try
    {
        g.addVertex(newVertex);
    }
    catch(GraphException e)
    {
    }
}
```

Parse the vertices and add them to the graph

```
st = new java.util.StringTokenizer(undirectedAndNotConnectedEdges, "{}");
String inn = st.nextToken();
st = new java.util.StringTokenizer(inn, "(),");
while( st.hasMoreTokens() )
{
    int from = Integer.parseInt(st.nextToken());
    int to = Integer.parseInt(st.nextToken());

    try
    {
        g.addEdge(from, to);
    }
    catch(GraphException e)
    {
    }
}
```

Parse the edges and add them to the graph

```
java.util.HashSet<Integer> connectedSubset = g.getConnectedSet(startingVertex);
```

```
boolean correctSize = connectedSubset.size() == undirectedAndNotConnectedSubset.length;
```

```
boolean containsCorrectVertices = true;
```

```
for( int i = 0; i < undirectedAndNotConnectedSubset.length; i++ )
```

```
{
    containsCorrectVertices = containsCorrectVertices && connectedSubset.contains(undirectedAndNotConnectedSubset[i]);
}
```

```
assertTrue(correctSize && containsCorrectVertices);
```

```
}
```

Get the connected set containing vertex 2 and check if it is correct

- JUnit test (cont)

- And finally, for the toString() test, we create an undirected graph with vertices 0, 1, ..., 9 and edges (0,1), (1,2), ..., (8,9) and then compare the output of the toString() method with the expected result

- Below is my code

```
// add vertices 0 - 9, and edges (i,i+1), i = 0, ..., 8
// verify toString() comes out correct
@Test
void testToString()
{
    String expectedToString = "G = (V, E)\n";
    expectedToString = expectedToString + "V = {0,1,2,3,4,5,6,7,8,9}\n";
    expectedToString = expectedToString + "E = {(0,1),(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9)}";

    for( int i = 0; i < 10; i++ )
    {
        try
        {
            g.addVertex(i);
        }
        catch(GraphException e)
        {
        }
    }

    for( int i = 0; i < 9; i++ )
    {
        try
        {
            g.addEdge(i, i+1);
        }
        catch(GraphException e)
        {
        }
    }

    assertEquals(expectedToString, g.toString());
}
```

Add vertices 0, 1, ..., 9

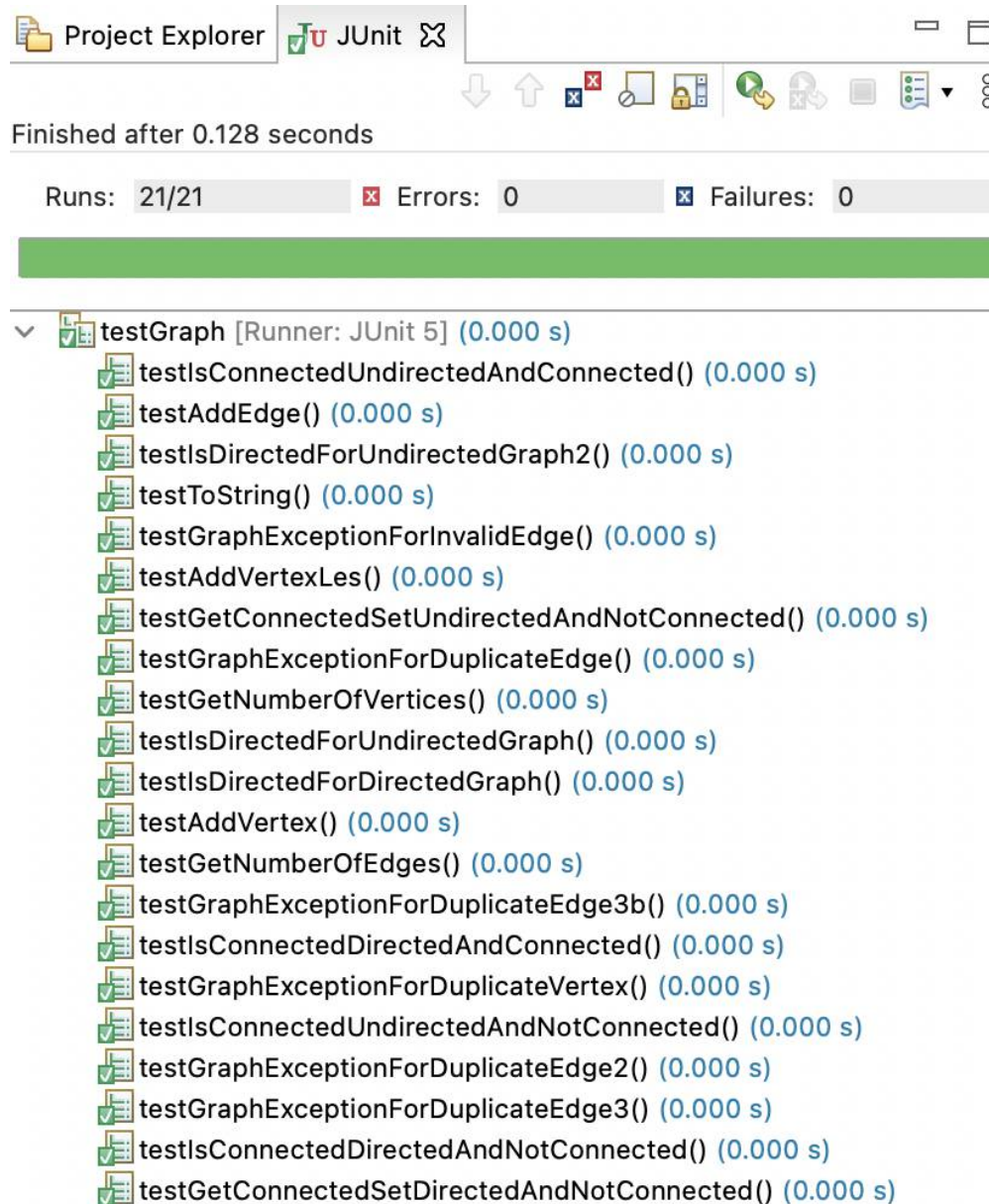
Add edges (0, 1), (1, 2), ..., (8, 9)

- JUnit test (cont)
- To run the JUnit tests
  - Right click on the test program
  - Select run as
  - Select JUnit test
  - A JUnit panel should show up on the tab that Package Explorer is on
  - It should list the number of tests run, how many errors there were, and how many failures there were
    - The next slide shows the JUnit panel
- The first time I tried to run the tests on remote140, I got a popup window saying “The input type of the launch configuration does not exits”
  - I clicked OK
  - I then selected “Run all tests in the selected project, package or source folder:”
    - It filled in “program4”
    - And then I clicked run and it executed the tests

- JUnit test (cont)

- Here's what the JUnit view looks like in eclipse for my implementation

- I have two extra tests



- JUnit test (cont)
- You can also compile and execute JUnit5 tests from the commandline
- With JUnit4 the files that are included with eclipse are all you need to compile and execute tests from the commandline
- With JUnit5 I was able to compile using the files that came with eclipse, but I couldn't figure out how to get it to run from the commandline
- But you can download “junit-platform-console-standalone-1.8.2.jar” which has everything you need to compile and execute test from the commandline
- My testing file is named testGraph.java, and is in the same folder as ConnectedGraphFunctions.java, GraphException.java, garrison\_Graph.java, and Edge.java
- To compile I executed
  - “javac -classpath junit-platform-console-standalone-1.8.2.jar:. testGraph.java”
- To run the tests I executed
  - “java -jar junit-platform-console-standalone-1.8.2.jar --class-path . --select-class testGraph”

- JUnit test (cont)
- Here's the output after running from the commandline
  - Once again, I have two extra tests

Thanks for using JUnit! Support its development at <https://junit.org/sponsoring>

```

JUnit Jupiter ✓
└─ testGraph ✓
   ├── testIsConnectedUndirectedAndConnected() ✓
   ├── testAddEdge() ✓
   ├── testIsDirectedForUndirectedGraph2() ✓
   ├── testToString() ✓
   ├── testGraphExceptionForInvalidEdge() ✓
   ├── testAddVertexLes() ✓
   ├── testGetConnectedSetUndirectedAndNotConnected() ✓
   ├── testGraphExceptionForDuplicateEdge() ✓
   ├── testGetNumberOfVertices() ✓
   ├── testIsDirectedForUndirectedGraph() ✓
   ├── testIsDirectedForDirectedGraph() ✓
   ├── testAddVertex() ✓
   ├── testGetNumberOfEdges() ✓
   ├── testGraphExceptionForDuplicateEdge3b() ✓
   ├── testIsConnectedDirectedAndConnected() ✓
   ├── testGraphExceptionForDuplicateVertex() ✓
   ├── testIsConnectedUndirectedAndNotConnected() ✓
   ├── testGraphExceptionForDuplicateEdge2() ✓
   ├── testGraphExceptionForDuplicateEdge3() ✓
   ├── testIsConnectedDirectedAndNotConnected() ✓
   └─ testGetConnectedSetDirectedAndNotConnected() ✓
JUnit Vintage ✓

```

```

Test run finished after 96 ms
[      3 containers found      ]
[      0 containers skipped    ]
[      3 containers started    ]
[      0 containers aborted    ]
[      3 containers successful  ]
[      0 containers failed     ]
[     21 tests found           ]
[      0 tests skipped         ]
[     21 tests started         ]
[      0 tests aborted         ]
[     21 tests successful      ]
[      0 tests failed          ]

```