JUnit Test Fixture Pattern
The Problem

• In principle, a test fixture for a kernel class should be usable with any kernel class (UUT, or “unit under test”) that purports to implement the same interface.

• However, you actually must modify the test fixture when you change the UUT, because each test case has to call the UUT’s constructor—whose name is that of the UUT!
Example

```java
@Test
public final void testPushEmpty() {
    Stack<String> s = new Stack1L<>();
    Stack<String> sExpected = new Stack1L<>();
    s.push("red");
    sExpected.push("red");
    assertEquals(sExpected, s);
}
```
Example

```java
@Test
dpublic final void testPushEmpty() {
    Stack<String> s = new Stack1L<>();
    Stack<String> sExpected =
        new Stack1L<>();
    s.push("red");
    sExpected.push("red");
    assertEquals(sExpected, s);
}
```

This kind of call of the constructor for the UUT needs to be changed (throughout the test fixture) if you want to change the UUT.
Single-Point Control Over Change

• **Question**: How do we achieve the goal of *single-point control over change* in this situation?
Single-Point Control Over Change

• **Question:** How do we achieve the goal of *single-point control over change* in this situation?

• **Answer:** “Re-route” all UUT constructor calls to another method, which then calls the UUT constructor, so only the body of that one method needs to be changed to accommodate a different UUT.
@Test
public final void testPushEmpty() {
    Stack<String> s = this.constructorTest();
    Stack<String> sExpected = new Stack<>();
    s.push("red");
    sExpected.push("red");
    assertEquals(sExpected, s);
}
Code for constructorTest

Stack<String> constructorTest() {
    return new Stack1L<String>();
}

15 January 2019 OSU CSE
### The Code for a Test Case

```java
@Test
public final void testPushEmpty() {
    Stack<String> s = this.constructorTest();
    Stack<String> sExpected = this.constructorRef();
    s.push("red");
    sExpected.push("red");
    assertEquals(sExpected, s);
}
```

Instead of calling the UUT’s constructor directly here, you call a method (perhaps named constructorRef) ...
The Code for a Test Case

```java
@Test
public final void testPushEmpty() {
    Stack<String> s = this.constructorTest();
    Stack<String> sExpected = this.constructorRef();
    s.push("red");
    sExpected.push("red");
    assertEquals(sExpected, s);
}
```

... which then calls the constructor from either the UUT or a reference implementation of the same interface.
Code for `constructorRef`

- If there is no reference implementation:
  ```java
  Stack<String> constructorRef() {
      return new Stack1L<String>();
  }
  ```

- If, say, `Stack3` is available as a reference implementation:
  ```java
  Stack<String> constructorRef() {
      return new Stack3<String>();
  }
  ```
Isolating This Change Point

test cases

constructorTest
constructorRef

Stack1LTest

StackTest

extends
The Code of StackTest

```java
public abstract class StackTest {

    protected abstract Stack<String> constructorTest();

    protected abstract Stack<String> constructorRef();

    ...

}
```
The Code of **StackTest**

```java
public abstract class StackTest {

    protected abstract Stack<String> constructorTest();

    protected abstract Stack<String> constructorRef();

    StackTest is an abstract class now.

    ...

}
```
public abstract class StackTest {

    protected abstract Stack<String> constructorTest();

    protected abstract Stack<String> constructorRef();

    ...

}

**protected** means that this method *may be called or overridden in a subclass* (which is our intent for **Stack1LTest**), but *may not even be called* from any other class declared outside this package.
The Code of StackTest

public abstract class StackTest {

    protected abstract Stack<String> constructorTest();

    protected abstract Stack<String> constructorRef();

    ...

}

abstract means that this method (called an abstract method) must be overridden in some subclass (which is our intent for Stack1LTest).
The Code of Stack1LTest

public class Stack1LTest extends StackTest {

    @Override
    protected final Stack<String> constructorTest() {
        ...
    }

    @Override
    protected final Stack<String> constructorRef() {
        ...
    }
}
The Code of Stack1LTest

```java
public class Stack1LTest extends StackTest {

    @Override
    protected final Stack<String> constructorTest() {
        ... 
    }

    @Override
    protected final Stack<String> constructorRef() {
        ... 
    }
}
```

Incidentally, **final** here means that this method *may not be overridden* in *any* subclass of Stack1LTest.
public class Stack1LTest extends StackTest {

    @Override
    protected final Stack<String> constructorTest() {
        ...
    }

    @Override
    protected final Stack<String> constructorRef() {
        ...
    }
}
Another Problem

• In a typical test case, it takes many lines of code just to construct the values on which to run a test!

• For instance, suppose you want to test a method with this `Stack<String>` value as input:

  `<"a", "stack", "with", "stuff">`
Another Problem

• In a typical test case, it takes many lines of code just to construct values to run a test!
• For instance, suppose you want to test a method with this `Stack<String>` value as input:
  ```java
  <"a", "stack", "with", "stuff">
  ```

What code must you write to give a variable this value?
A Convenience Method

• Consider this method in `StackTest`:

```java
/**
   * Creates and returns a Stack<String>
   * with the given entries.
   * @ensures
   * `createFromArgsTest = [entries in args]`
   */

private Stack<String> createFromArgsTest(String... args) {/* body */}
```
A Convenience Method

• Consider this method in StackTest:

```java
/**
 * Creates and returns a Stack<String>
 * with the given entries.
 *
 * @ensures
 * createFromArgsTest = [entries in args]
 */

private Stack<String> createFromArgsTest(
    String... args) {/* body */}
```

This special Java **varargs** notation (yes, three dots, or ellipsis, is part of the Java code here!) allows you to call this method with zero or more arguments of type `String`. 
A Convenience Method

• Consider this method:

```java
/**
 * Creates and returns a Stack<String>
 * with the given entries.
 *
 * @ensures
 * @ensures createFromArgsTest = [entries in args]
 */
private Stack<String> createFromArgsTest(String... args) {/* body */}
```

The method body is written as if `args` were an array of Strings of length equal to the number of arguments in the call; and `args` is guaranteed *not to be null.*
A Convenience Method

- Consider this method in `StackTest`:

```java
/**
 * Creates and returns a Stack<String>
 * with the given entries.
 *
 @ensures
 * createFromArgsTest = [entries in args]
 */

private Stack<String> createFromArgsRef(
    String... args) {/* body */}
```

In addition to `createFromArgsTest` you will also want `createFromArgsRef`. 