

Each slide has one of six questions. In each of them participants were asked to select their preferred style for each of dimensions and give feedback

Q1: Between the two unit test styles shown, which one do you prefer based on the following factors? *

	Style 1	Style 2
Readability	<input type="radio"/>	<input type="radio"/>
Maintainability	<input type="radio"/>	<input type="radio"/>
Extensibility	<input type="radio"/>	<input type="radio"/>
Debugging & Issue Diagnosis	<input type="radio"/>	<input type="radio"/>
Robustness	<input type="radio"/>	<input type="radio"/>
Over All	<input type="radio"/>	<input type="radio"/>

Please feel free to share any thought, feedback, or suggestion you had while answering this question.

Short answer text

Style 1

```
@Test public void getByName() {  
    [assertEquals( expected: ">", Entities.getByName("n6t"));  
    [assertEquals( expected: "fj", Entities.getByName("fjlig"));  
    [assertEquals( expected: ">", Entities.getByName("gg"));  
    [assertEquals( expected: "@", Entities.getByName("copy"));  
}
```

Legend

[Represents structurally similar lines/blocks

Style 2

```
@ParameterizedTest  
@CsvSource(value = {  
    ">| n6t",  
    "fj, fjlig",  
    ">, gg",  
    "@, copy"  
})  
public void getByName(String expected, String name) {  
    [assertEquals(expected, Entities.getByName(name));  
}
```

In this Style 2
is our
refactored test

Style 1

```
@ParameterizedTest
@CsvSource(value = {
    "38, amp",
    "62, gt",
    "60, lt",
    "34, quot"
})
public void xhtmlCodepointForName(int expectedCodepoint, String name) {
    [assertEquals(expectedCodepoint, xhtml.codepointForName(name));
}

@ParameterizedTest
@CsvSource(value = {
    "amp, 38",
    "gt, 62",
    "lt, 60",
    "quot, 34"
})
public void xhtmlNameForCodepoint(String expectedName, int codepoint) {
    [assertEquals(expectedName, xhtml.nameForCodepoint(codepoint));
}
```

Style 2

```
@Test
public void xhtml() {
    [assertEquals( expected: 38, xhtml.codepointForName("amp"));
    [assertEquals( expected: 62, xhtml.codepointForName("gt"));
    [assertEquals( expected: 60, xhtml.codepointForName("lt"));
    [assertEquals( expected: 34, xhtml.codepointForName("quot"));

    [assertEquals( expected: "amp", xhtml.nameForCodepoint(38));
    [assertEquals( expected: "gt", xhtml.nameForCodepoint(62));
    [assertEquals( expected: "lt", xhtml.nameForCodepoint(60));
    [assertEquals( expected: "quot", xhtml.nameForCodepoint(34));
}
```

Legend

- [Represents structurally similar lines/blocks
- [Represents another type of structurally similar lines/blocks

In this Style 1
is our
refactored test

Style 1

```
static Stream<Object[]> memorySizeProvider() {  
    return Stream.of(  
        new Object[]{MemorySize.ZERO, 0, 0, 0, 0, 0},  
        new Object[]{new MemorySize( bytes: 955), 955, 0, 0, 0, 0},  
        new Object[]{new MemorySize( bytes: 18500), 18500, 18, 0, 0, 0},  
        new Object[]{new MemorySize( bytes: 15 * 1024 * 1024), 15_728_640, 15_360, 15, 0, 0},  
        new Object[]{new MemorySize( bytes: 2L * 1024 * 1024 * 1024 * 1024 + 10), 2199023255562L,  
            2147483648L, 2097152, 2048, 2}  
    );  
}  
  
no usages new *  
@ParameterizedTest  
@MethodSource("memorySizeProvider")  
void testUnitConversion(MemorySize memorySize, long expectedBytes, long expectedKibiBytes,  
    long expectedMebiBytes, long expectedGibiBytes, long expectedTebiBytes) {  
    assertEquals(expectedBytes, memorySize.getBytes());  
    assertEquals(expectedKibiBytes, memorySize.getKibiBytes());  
    assertEquals(expectedMebiBytes, memorySize.getMebiBytes());  
    assertEquals(expectedGibiBytes, memorySize.getGibiBytes());  
    assertEquals(expectedTebiBytes, memorySize.getTebiBytes());  
}
```

Legend

 Represents structurally similar lines/blocks

In this Style 1
is our
refactored test

Style 2

```
@Test  
public void testUnitConversion() {  
    final MemorySize zero = MemorySize.ZERO;  
    assertEquals( expected: 0, zero.getBytes());  
    assertEquals( expected: 0, zero.getKibiBytes());  
    assertEquals( expected: 0, zero.getMebiBytes());  
    assertEquals( expected: 0, zero.getGibiBytes());  
    assertEquals( expected: 0, zero.getTebiBytes());  
  
    final MemorySize bytes = new MemorySize( bytes: 955);  
    assertEquals( expected: 955, bytes.getBytes());  
    assertEquals( expected: 0, bytes.getKibiBytes());  
    assertEquals( expected: 0, bytes.getMebiBytes());  
    assertEquals( expected: 0, bytes.getGibiBytes());  
    assertEquals( expected: 0, bytes.getTebiBytes());  
  
    final MemorySize kilos = new MemorySize( bytes: 18500);  
    assertEquals( expected: 18500, kilos.getBytes());  
    assertEquals( expected: 18, kilos.getKibiBytes());  
    assertEquals( expected: 0, kilos.getMebiBytes());  
    assertEquals( expected: 0, kilos.getGibiBytes());  
    assertEquals( expected: 0, kilos.getTebiBytes());  
  
    final MemorySize megas = new MemorySize( bytes: 15 * 1024 * 1024);  
    assertEquals( expected: 15_728_640, megas.getBytes());  
    assertEquals( expected: 15_360, megas.getKibiBytes());  
    assertEquals( expected: 15, megas.getMebiBytes());  
    assertEquals( expected: 0, megas.getGibiBytes());  
    assertEquals( expected: 0, megas.getTebiBytes());  
  
    final MemorySize teras = new MemorySize( bytes: 2L * 1024 * 1024 * 1024 * 1024 + 10);  
    assertEquals( expected: 2199023255562L, teras.getBytes());  
    assertEquals( expected: 2147483648L, teras.getKibiBytes());  
    assertEquals( expected: 2097152, teras.getMebiBytes());  
    assertEquals( expected: 2048, teras.getGibiBytes());  
    assertEquals( expected: 2, teras.getTebiBytes());  
}
```

Style 1

```
@Test
public void testHashCode() {
    final TextStringBuilder sb = new TextStringBuilder( initialCapacity: 10);
    final int hc1a = sb.hashCode();
    final int hc1b = sb.hashCode();
    Assertions.assertEquals(hc1a, hc1b);

    final int emptyHc = Arrays.hashCode(sb.getChars(new char[0]));
    assertNotEquals(emptyHc, hc1a);

    final TextStringBuilder sb2 = new TextStringBuilder( initialCapacity: 8000);
    final int h1a = sb2.hashCode();
    final int h1b = sb2.hashCode();
    Assertions.assertEquals(h1a, h1b);

    sb2.append("abc");
    final int hc2b2 = sb2.hashCode();
    final int hc3b2 = sb2.hashCode();
    Assertions.assertEquals(hc2b2, hc3b2);
}
```

Legend

- Represents structurally similar lines/blocks
- Represents another type of structurally similar lines/blocks

Style 2

```
@Test
public void testHashCodeAndEmptyCondition() {
    final TextStringBuilder sb = new TextStringBuilder( initialCapacity: 10);
    final int hc1a = sb.hashCode();
    final int hc1b = sb.hashCode();
    Assertions.assertEquals(hc1a, hc1b);

    final int emptyHc = Arrays.hashCode(sb.getChars(new char[0]));
    assertNotEquals(emptyHc, hc1a);
}

no usages new *

@Test
public void testHashCodeWithAppend() {
    final TextStringBuilder sb2 = new TextStringBuilder( initialCapacity: 8000);
    final int h1a = sb2.hashCode();
    final int h1b = sb2.hashCode();
    Assertions.assertEquals(h1a, h1b);

    sb2.append("abc");
    final int hc2b2 = sb2.hashCode();
    final int hc3b2 = sb2.hashCode();
    Assertions.assertEquals(hc2b2, hc3b2);
}
```

In this Style 2
is our
refactored test

Style 1

```
@Test
public void testHashCode() {
    final TextStringBuilder sb = new TextStringBuilder( initialCapacity: 10);
    final int hc1a = sb.hashCode();
    final int hc1b = sb.hashCode();
    Assertions.assertEquals(hc1a, hc1b);

    sb.append("abc");
    final int hc1b1 = sb.hashCode();
    final int hc1b2 = sb.hashCode();
    Assertions.assertEquals(hc1b1, hc1b2);

    final TextStringBuilder sb2 = new TextStringBuilder( initialCapacity: 2000);
    final int h1a = sb2.hashCode();
    final int h1b = sb2.hashCode();
    Assertions.assertEquals(h1a, h1b);

    sb2.append("123");
    final int hc2b2 = sb2.hashCode();
    final int hc3b2 = sb2.hashCode();
    Assertions.assertEquals(hc2b2, hc3b2);
}
```

Style 2

```
@ParameterizedTest
@CsvSource({
    "10, 'abc'",
    "2000, '123'"
})
public void testHashCode(int capacity, String appendValue) {
    final TextStringBuilder sb = new TextStringBuilder(capacity);
    final int hcBefore = sb.hashCode();
    final int hcBeforeDuplicate = sb.hashCode();
    Assertions.assertEquals(hcBefore, hcBeforeDuplicate);

    sb.append(appendValue);
    final int hcAfter = sb.hashCode();
    final int hcAfterDuplicate = sb.hashCode();
    Assertions.assertEquals(hcAfter, hcAfterDuplicate);
}
```

Legend

 Represents structurally similar lines/blocks

In this Style 2
is our
refactored test

Style 1

```
@Test
public void testHashCode() {
    final TextStringBuilder sb = new TextStringBuilder( initialCapacity: 10);
    final int hc1a = sb.hashCode();
    final int hc1b = sb.hashCode();
    Assertions.assertEquals(hc1a, hc1b);

    sb.append("abc");
    final int hc1b1 = sb.hashCode();
    final int hc1b2 = sb.hashCode();
    Assertions.assertEquals(hc1b1, hc1b2);

    final TextStringBuilder sb2 = new TextStringBuilder( initialCapacity: 2000);
    final int h1a = sb2.hashCode();
    final int h1b = sb2.hashCode();
    Assertions.assertEquals(h1a, h1b);

    sb2.append("123");
    final int hc2b2 = sb2.hashCode();
    final int hc3b2 = sb2.hashCode();
    Assertions.assertEquals(hc2b2, hc3b2);
}
```

Legend

- Represents structurally similar lines/blocks
- Represents another type of structurally similar lines/blocks

In this Style 2
is our
refactored test

Style 2

```
@ParameterizedTest
@CsvSource({
    "10",
    "2000"
})
public void testHashCode(int capacity) {
    final TextStringBuilder sb = new TextStringBuilder(capacity);
    final int hcBefore = sb.hashCode();
    final int hcBeforeDuplicate = sb.hashCode();
    Assertions.assertEquals(hcBefore, hcBeforeDuplicate);
}

no usages new *
@ParameterizedTest
@CsvSource({
    "10, 'abc'",
    "2000, '123'"
})
public void testHashCodeAppendValue(int capacity, String appendValue) {
    final TextStringBuilder sb = new TextStringBuilder(capacity);

    sb.append(appendValue);
    final int hcAfter = sb.hashCode();
    final int hcAfterDuplicate = sb.hashCode();
    Assertions.assertEquals(hcAfter, hcAfterDuplicate);
}
```