CORRECT

Full Stack Web Development

C.O.R.R.E.C.T.

- **C**onformance - Does the value conform to an expected format?
- **O**rdering - Is the set of values ordered or unordered as appropriate?
- **R**ange - Is the value within reasonable minimum and maximum values?
- **R**eference - Does the code reference anything external that isn't under direct control of the code itself?
- **E**xistence - Does the value exist (e.g. non-null, nonzero, present in a set, etc.)?
- **C**ardinality - Are there exactly enough values?
- **T**ime - (absolute and relative) Is everything happening in order? At the right time? In time?

C.O.R.R.E.C.T Thinking

- For each of the CORRECT criteria, consider the impact of data from all possible origins.
- The underlying question to be constantly considered is:
  - *What can go wrong?*
- Once you think of something that could go wrong, write a test for it. Once that test passes, again ask
  - *What else can go wrong?*

- **C**onformance
- **O**rdering
- **R**ange
- **R**eference
- **E**xistence
- **C**ardinality
- **T**ime

[C].O.R.R.E.C.T – [C]onformance

- When data in a specific format is expected → consider what will happen if the data does not conform to the structure.
- e.g. an email address :

  name@somewhere.com

  firstname.lastname@subdomain.somewhere.com

  firstname.lastname%somewhere@subdomain.somewhere.com

  firstname
- How will code react to each of these?
- Similarly, if code is producing data to a specific format, tests must verify that the generated data conforms to desired format.

C.[O].R.R.E.C.T – [O]rdering

- Position of one piece of data within a larger collection.
- A search routine should be tested for conditions where the search target is first or last.
- For a sort routine, what might happen if the set of data is already ordered? Or sorted in precisely reverse order?

```
public void testOrder ()
{
  assertEquals(9, Largest.largest(new int[] { 9, 8, 7 }));
  assertEquals(9, Largest.largest(new int[] { 8, 9, 7 }));
  assertEquals(9, Largest.largest(new int[] { 7, 8, 9 }));
}

public void testDups ()
{
  assertEquals(9, Largest.largest(new int[] { 9, 7, 9, 8 }));
}

public void testOne ()
{
  assertEquals(1, Largest.largest(new int[] { 1 }));
}

public void testNegative ()
{
  int[] negList = new int[] { -9, -8, -7 };
  assertEquals(-7, Largest.largest(negList));
}

public void testEmpty ()
{
  try
  {
    Largest.largest(new int[] {});
    fail("Should have thrown an exception");
  }
  catch (RuntimeException e)
  {
    assertTrue(true);
  }
}
```

C.O.[R].R.E.C.T – [R]ange – Example

- A variable's primitive type may allow it to take on a wider range of values than needed e.g. int age.
- Consider not using primitive types to store bounded-integer values e.g. direction of travel → Bearing.
- Encapsulating a bearing within a class enables you to constrain its range at one point in the system i.e. you can filter out bad data.

```
public class Bearing
{
  public static final int MAX = 359;
  private int value;

  public Bearing(int value)
  {
    if (value < 0 || value > MAX) throw new BearingOutOfRangeException();
      this.value = value;
    }

  public int value() { return value; }

  public int angleBetween(Bearing bearing)
  {
    return value - bearing.value;
  }
}
```

```
public class BearingTest
{
    @Test(expected=BearingOutOfRangeException.class)
    public void throwsOnNegativeNumber()
    {
        new Bearing(-1);
    }

    @Test(expected=BearingOutOfRangeException.class)
    public void throwsWhenBearingTooLarge()
    {
        new Bearing(Bearing.MAX + 1);
    }

    @Test
    public void answersValidBearing()
    {
        assertThat(new Bearing(Bearing.MAX).value(), equalTo(Bearing.MAX));
    }

    @Test
    public void answersAngleBetweenItAndAnotherBearing()
    {
        assertThat(new Bearing(15).angleBetween(new Bearing(12)), equalTo(3));
    }

    @Test
    public void angleBetweenIsNegativeWhenThisBearingSmaller()
    {
        assertThat(new Bearing(12).angleBetween(new Bearing(15)), equalTo(-3));
    }
}
```

C.O.R.[R].E.C.T – [R]eference

- What things does the method-under-test reference that are outside the scope of the method itself?
  - external dependencies
  - object state
  - other conditions
- e.g.
  - a method in a web application to display a customer's account history might require that the customer is first logged on.
  - the method pop() for a stack requires a nonempty stack.
  - shifting the transmission in a car to Park from Drive requires that the car is stopped.

C.O.R.[R].E.C.T– [R]eference

- If assumptions are made about:
  - the state of the class,
  - the state of other objects,
  - the global application,
- Then you need to verify your code is well-behaved if these assumptions/ conditions are not met.

```
@Test
public void remainsInDriveAfterAcceleration()
{
  transmission.shift(Gear.DRIVE);
  car.accelerateTo(35);
  assertThat(transmission.getGear(),
                                  equalTo(Gear.DRIVE));
}

@Test
public void ignoresShiftToParkWhileInDrive()
{
   transmission.shift(Gear.DRIVE);
   car.accelerateTo(30);
   transmission.shift(Gear.PARK);
   assertThat(transmission.getGear(),
                                   equalTo(Gear.DRIVE));
}

@Test
public void allowsShiftToParkWhenNotMoving()
{
   transmission.shift(Gear.DRIVE);
   car.accelerateTo(30);
   car.brakeToStop();
   transmission.shift(Gear.PARK);
   assertThat(transmission.getGear(),
                                  equalTo(Gear.PARK));
}
```

C.O.R.R.[E].C.T – [E]xistence

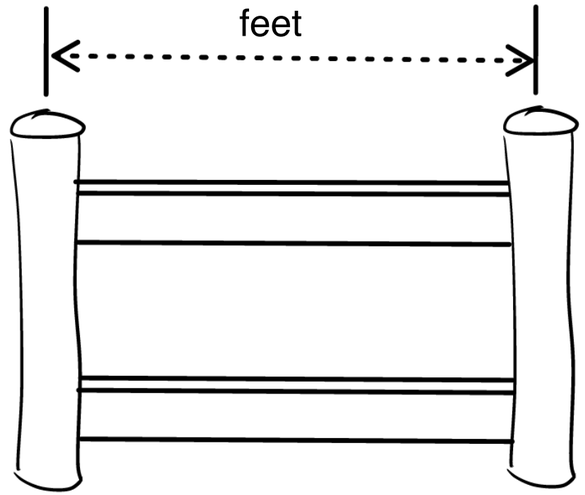
- Make sure the method under test can stand up to nothing!

  Network resource
  files
  URLs
  license keys
  users
  printers…

  - may all disappear without notice.
- Many Java library methods will throw an exception of some sort when faced with non-existent data.
  - Difficulty: hard to debug a generic runtime exception; but easier when your exceptions report a specific message!
- Should unit test with plenty of nulls, zeros, empty strings etc...

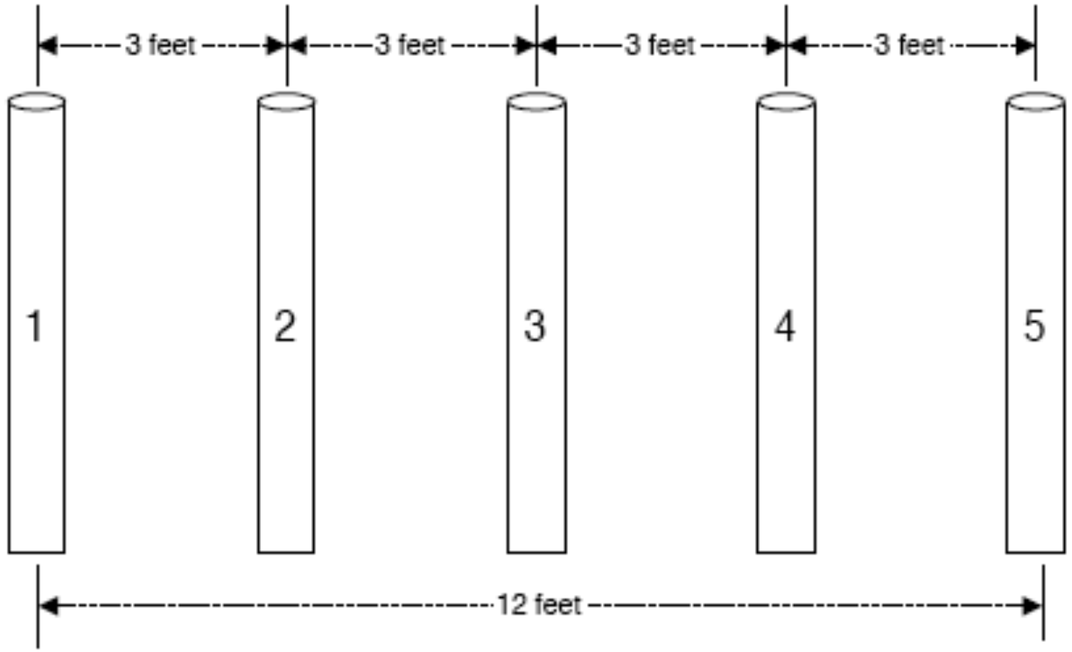
C.O.R.R.E.[C].T – [C]ardinality

Riddle: You have to erect a number of fence sections to cover a straight line 12 feet long. Each section of fencing covers 3 feet, and each end of a section must be held up with a fence post:



How many fence posts do you need?

C.O.R.R.E.[C].T – [C]ardinality



- This problem, and the related common errors, come up so often that they are graced with the name “fencepost errors” or “off-by-one errors”

C.O.R.R.E.[C].T – [C]ardinality

- Related to ***CORR[E]CT: Existence*** i.e. how to make sure there are exactly as many items as needed.
- The count of some set of values is most interesting in these three cases:
  - 1. Zero
  - 2. One
  - 3. More than one
- It's called the “0-1-n-Rule” and it's based on the premise that if method can handle more than one of something, it can probably handle 10, 20, or 1,000.
- Sometimes n may be significant -
  - top 10 results
  - leading 100 users

C.O.R.R.E.[C].T – [C]ardinality

Example: If maintaining a top 10 list of items, tests should consider:

- Producing a report when:
    - there are no items in the list *(zero)*
    - there's only one item in the list *(one)*
    - there aren't yet ten items in the list *(many)*
- Adding an item when:
    - there are no items in the list *(zero)*
    - there's only one item in the list *(one)*
    - there aren't yet ten items in the list *(many)*
    - there are already ten items in the list *(many boundary)*

C.O.R.R.E.C.[T] – [T]ime

- You need to keep several aspects of time in mind:
    - Relative time (ordering in time)
    - Absolute time (elapsed and wall clock)
    - Concurrency issues

C.O.R.R.E.C.[T] – [T]ime – Relative ordering in time

- Some interfaces are inherently stateful:
  - login() will be called before logout().
  - prepareStatement() is called before executeStatement().
  - connect() before read() which is before close().

- Test calling methods out of the expected order try skipping the first, last and middle of a sequence *(i.e. C[O]RRECT – [O]rdering).*

- Relative time can include timeout issues:
  - How long your code is willing to wait for a resource to become available.
  - What happens in your code if the resource never becomes available?

C.O.R.R.E.C.[T] – [T]ime - Absolute

The actual elapsed or “wall clock” time:

- **Elapsed time:** when waiting for a resource, is the elapsed time too long?

- **Wall Clock time:** Most of the time, this makes no difference. However, occasionally, the actual time of day will matter.

  - e.g.: Question: every day of the year is 24 hours long? - true or false?

C.O.R.R.E.C.[T] – [T]ime - Absolute

- Answer: It Depends!
- In UTC (Universal Coordinated Time, the modern version of Greenwich Mean Time, or GMT), the answer is TRUE.
- In areas of the world that does not observe Daylight Savings Time (DST), the answer is TRUE.
- In most of the U.S. (which does observe DST), the answer is FALSE.
  - In April, you'll have a day with 23 hours (spring forward) and in October you'll have a day with 25 (fall back).
  - This means that arithmetic won't always work as you expect two days in the year (you need to test on these two boundary days):
    - 1:45AM plus 30 minutes might equal 1:15AM, rather than 2:15AM.

Question: every day of the year is 24 hours long? - true or false?

C.O.R.R.E.C.[T] – [T]ime - Concurrency

- What will happen if multiple threads use this same object at the same time?
- Are there global or instance level data or methods that need to be synchronized?
- How about external access to files or hardware?
- *If you have concurrency needs, you need to write tests that demonstrate the use of multiple client threads.*

CORRECT

Full Stack Web Development