```
    @Override
    protected boolean matchesSafely(Rectangle rect) {
        return
          Math.abs(rect.origin().x - rect.opposite().x) <= length &&
          Math.abs(rect.origin().y - rect.opposite().y) <= length;
    }

    @Factory
    public static <T> Matcher<Rectangle> constrainsSidesTo(int length) {
        return new ConstrainsSidesTo(length);
    }
}
```

Testing Ranges by Embedding Invariant Methods

The most common ranges you'll test will likely depend on data-structure concerns, not application-domain constraints.

Let's look at a questionable implementation of a sparse array—a data structure designed to save space. The sweet spot for a sparse array is a broad range of indexes where most of the corresponding values are null. It accomplishes this goal by storing only non-null values, using a pair of arrays that work in concert: an array of indexes corresponds to an array of values.

Here's the bulk of the source for the SparseArray class:

iloveyouboss/16/src/util/SparseArray.java

```
public class SparseArray<T> {
    public static final int INITIAL_SIZE = 1000;
    private int[] keys = new int[INITIAL_SIZE];
    private Object[] values = new Object[INITIAL_SIZE];
    private int size = 0;

    public void put(int key, T value) {
        if (value == null) return;

        int index = binarySearch(key, keys, size);
        if (index != -1 && keys[index] == key)
            values[index] = value;
        else
            insertAfter(key, value, index);
    }

    public int size() {
        return size;
    }

    private void insertAfter(int key, T value, int index) {
        int[] newKeys = new int[INITIAL_SIZE];
        Object[] newValues = new Object[INITIAL_SIZE];
```

```
        copyFromBefore(index, newKeys, newValues);

        int newIndex = index + 1;
        newKeys[newIndex] = key;
        newValues[newIndex] = value;

        if (size - newIndex != 0)
            copyFromAfter(index, newKeys, newValues);

        keys = newKeys;
        values = newValues;
    }

    private void copyFromAfter(int index, int[] newKeys, Object[] newValues) {
        int start = index + 1;
        System.arraycopy(keys, start, newKeys, start + 1, size - start);
        System.arraycopy(values, start, newValues, start + 1, size - start);
    }

    private void copyFromBefore(int index, int[] newKeys, Object[] newValues) {
        System.arraycopy(keys, 0, newKeys, 0, index + 1);
        System.arraycopy(values, 0, newValues, 0, index + 1);
    }

    @SuppressWarnings("unchecked")
    public T get(int key) {
        int index = binarySearch(key, keys, size);
        if (index != -1 && keys[index] == key)
            return (T)values[index];
        return null;
    }

    int binarySearch(int n, int[] nums, int size) {
        // ...
    }
}
```

One of the tests we want to write involves ensuring that we can add a couple of entries, then retrieve them both:

iloveyouboss/16/test/util/SparseArrayTest.java

```
@Test
public void handlesInsertionInDescendingOrder() {
    array.put(7, "seven");
    array.put(6, "six");
    assertThat(array.get(6), equalTo("six"));
    assertThat(array.get(7), equalTo("seven"));
}
```

The sparse-array code has some intricacies around tracking and altering the pair of arrays. One way to help prevent errors is to determine what invariants exist for the implementation specifics. In the case of our sparse-array implementation, which accepts only non-null values, the tracked size of the array must match the count of non-null values.

We might consider writing tests that probe at the values stored in the private arrays, but that would require exposing true private implementation details unnecessarily. Instead, we devise a checkInvariants() method that can do the skullduggery for us, throwing an exception if any invariants (well, we have only one so far) fail to hold true.

iloveyouboss/16/src/util/SparseArray.java

```
public void checkInvariants() throws InvariantException {
   long nonNullValues = Arrays.stream(values).filter(Objects::nonNull).count();
   if (nonNullValues != size)
      throw new InvariantException("size " + size +
            " does not match value count of " + nonNullValues);
}
```

(We could also implement invariant failures using the Java assert keyword.)

Now we can scatter checkInvariants() calls in our tests any time we do something to the sparse-array object:

iloveyouboss/16/test/util/SparseArrayTest.java

```
@Test
public void handlesInsertionInDescendingOrder() {
   array.put(7, "seven");
➤  array.checkInvariants();
   array.put(6, "six");
➤  array.checkInvariants();
   assertThat(array.get(6), equalTo("six"));
   assertThat(array.get(7), equalTo("seven"));
}
```

The test errors out with an InvariantException:

```
util.InvariantException: size 0 does not match value count of 1
   at util.SparseArray.checkInvariants(SparseArray.java:48)
   at util.SparseArrayTest
      .handlesInsertionInDescendingOrder(SparseArrayTest.java:65)
   ...
```

Our code indeed has a problem with tracking the internal size. Challenge: where's the defect?

Even though the later parts of the test would fail anyway given the defect, the checkInvariants calls allow us to pinpoint more easily where the code is failing.

Indexing needs present a variety of potential errors. As a parting note on the [R]ange part of the CORRECT mnemonic, here are a few test scenarios to consider when dealing with indexes:

- Start and end index have the same value
- First is greater than last
- Index is negative
- Index is greater than allowed
- Count doesn't match actual number of items

COR[R]ECT: [R]eference

When testing a method, consider:

- What it references outside its scope
- What external dependencies it has
- Whether it depends on the object being in a certain state
- Any other conditions that must exist

A web app that displays a customer's account history might require the customer to be logged on. The `pop()` method for a stack requires a nonempty stack. Shifting your car's transmission from Drive to Park requires you to first stop—if your transmission allowed the shift while the car was moving, it'd likely deliver some hefty damage to your fine Geo Metro.

When you make assumptions about any state, you should verify that your code is reasonably well-behaved when those assumptions are not met. Imagine you're developing the code for your car's microprocessor-controlled transmission. You want tests that demonstrate how the transmission behaves when the car is moving versus when it is not. Our tests for the `Transmission` code cover three critical scenarios: that it remains in Drive after accelerating, that it ignores the damaging shift to Park while in Drive, and that it *does* allow the shift to Park once the car isn't moving:

iloveyouboss/16/test/transmission/TransmissionTest.java

```
@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));
}
```

The *preconditions* for a method represent the state things must be in for it to run. The precondition for putting a transmission in Park is that the car must be at a standstill. We want to ensure that the method behaves gracefully when its precondition isn't met (in our case, we ignore the Park request).

Postconditions state the conditions that you expect the code to make true—essentially, the assert portion of your test. Sometimes this is simply the return value of a called method. You might also need to verify other *side effects*—changes to state that occur as a result of invoking behavior. In the `allowsShiftToParkWhenNotMoving` test case, calling `brakeToStop()` on the `car` instance has the side effect of setting the car's speed to 0.

CORR[E]CT: [E]xistence

You can uncover a good number of potential defects by asking yourself, "Does some given thing exist?" For a given method that accepts an argument or maintains a field, think through what will happen if the value is `null`, zero, or otherwise empty.

Java libraries tend to choke and throw an exception when faced with nonexistent or uninitialized data. Unfortunately, by the time a `null` value reaches the point where something chokes on it, it can be hard to understand the original source of the problem. An exception that reports a specific message, such as "profile name not set," greatly simplifies tracking down the problem.