A circle has only 360 degrees. Rather than store the direction of travel as a native type, create a class named Bearing that encapsulates the direction along with logic to constrain its range. Tests show how it works.

iloveyouboss/16/test/scratch/BearingTest.java

```java
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));
   }
}
```

The constraint is implemented in the constructor of the Bearing class:

iloveyouboss/16/src/scratch/Bearing.java

```java
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; }
}
```

Note that angleBetween() returns an int. We're not placing any range restrictions (such as that it must not be negative) on the result.

The Bearing abstraction makes it impossible for client code to create out-of-range bearings. As long as the rest of the system accepts and works with Bearing objects, the gate on range-related defects is shut.

Other constraints might not be as straightforward. Suppose we have a class that maintains two points, each an x, y integer tuple. The constraint on the range is that the two points must describe a rectangle with no side greater than 100 units. That is, the allowed range of values for both x, y pairs is interdependent.

We want a range assertion for any behavior that can affect a coordinate, to ensure that the resulting range of the x, y pairs remains legitimate—that the *invariant* on the Rectangle holds true.

More formally: an invariant is a condition that holds true throughout the execution of some chunk of code. In this case, we want the invariant to hold true for the lifetime of the Rectangle object—that is, any time its state changes.

We can add invariants, in the form of assertions, to the @After method so that they run upon completion of any test. Here's what an implementation for the invariant for our constrained Rectangle class looks like:

iloveyouboss/16/test/scratch/RectangleTest.java

```
import static org.junit.Assert.*;
import static org.hamcrest.CoreMatchers.*;
import static scratch.ConstrainsSidesTo.constrainsSidesTo;
import org.junit.*;

public class RectangleTest {
   private Rectangle rectangle;

➤  @After
➤  public void ensureInvariant() {
➤     assertThat(rectangle, constrainsSidesTo(100));
➤  }

   @Test
   public void answersArea() {
      rectangle = new Rectangle(new Point(5, 5), new Point (15, 10));
      assertThat(rectangle.area(), equalTo(50));
   }

   @Test
   public void allowsDynamicallyChangingSize() {
      rectangle = new Rectangle(new Point(5, 5));
      rectangle.setOppositeCorner(new Point(130, 130));
      assertThat(rectangle.area(), equalTo(15625));
   }
}
```

For all the tests that manipulate a rectangle instance, we can sleep safely, knowing that JUnit will always check the invariant. The last test, allowsDynamicallyChangingSize, violates the invariant and thus fails.

Creating a Custom Matcher to Verify an Invariant

The assertion in the @After method uses a custom Hamcrest matcher named constrainsSidesTo. The matcher provides an assertion phrasing that reads well left-to-right: assert that (the) rectangle constrains (its) sides to 100.

To implement our custom Hamcrest matcher, we extend a class from org.hamcrest.TypeSafeMatcher bound to the type that we're matching on—Rectangle in our case. By convention, we name the class ConstrainsSidesTo to correspond with the matcher phrasing constrainsSidesTo.

The class must override the matchesSafely() method to be useful. matchesSafely() contains the behavior we're trying to enforce. It returns true as long as both rectangle sides remain in range. A false return fails the constraint. The custom matcher class should override the describeTo() method to provide a meaningful message when the assertion fails.

The custom matcher class should also supply a static factory method that returns the matcher instance. You use this factory method when phrasing an assertion. The constrainsSidesTo() factory method passes the length constraint (100 in the test) to the constructor of the matcher, to be subsequently used by matchesSafely():

iloveyouboss/16/test/scratch/ConstrainsSidesTo.java
```
import org.hamcrest.*;

public class ConstrainsSidesTo extends TypeSafeMatcher<Rectangle> {
   private int length;

   public ConstrainsSidesTo(int length) {
      this.length = length;
   }

   @Override
   public void describeTo(Description description) {
      description.appendText("both sides must be <= " + length);
   }
   @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];
```