```
# ps3.txt
# Dan R. K. Ports <drkp@mit.edu>
# 6.170 PS3, 2004/02/24
# $Id: ps3.txt,v 1.4 2004/02/27 01:20:35 dan Exp $

REQUIREMENTS

The spec does not clearly define what should happen when a Car is
driving along a Route that contains a RoadSegment of zero length. The
fraction of segment travelled no longer has a meaningful
definition. In this implementation, the fraction is zero until drive()
is called, then changed to 1. Hence, it is necessary to call drive()
to arrive at the end of the segment (though this takes zero time)
before calling makeTurn(). This behavior could certainly be changed,
but the most reasonable thing to do would probably be to forbid
segments whose start and end GeoPoints are equal.


DESIGN

1. Route is implemented using an ArrayList to store the segments. An
   alternate implementation might use the front-and-back double-ended
   queue representation presented in lecture to store segments. This
   alternate implementation would be more efficient in situations
   where Routes were modified frequently (using many calls to
   addSegment and popSegment), but is considerably more complicated to
   implement.

2. If the precondition were changed, the new requires clause would be
   weaker and the new specification stronger than before. The
   implementation would have to be modified to deal with RoadSegments
   that are not properly oriented. Probably the simplest way to do
   this would be to modify addSegment to check whether the new segment
   is properly oriented, reverse it if it is not, and proceed
   normally. Nothing else would need to be changed.

3. Two Routes consisting of the same RoadSegments in the same order
   should be considered equal (in the context of equals()), because
   this is what the specification indicates. Since Routes are
   immutable, they are behaviorally equivalent in this case.

4. These Cars should not be judged equal, because they are not
   behaviorally equivalent: if we setSpeed() or drive() one of the
   cars, its speed and position will change, and they will no longer
   be observationally equivalent. (Cars are mutable.)


TESTING

This implementation was tested using the provided GeoPointTest and
RoadSegmentTest, as well as RouteTest and CarTest, and passed all
tests.

Routes were tested by generating some Route objects using a small
collection of RoadSegments. Since all tests required a few simple
Route objects to manipulate, all tests depended on the constructor
and addSegment(); The observers were tested, then the mutators, since
the observers were necsesary to verify the results of the mutator.

Cars were tested by generating a few instances, using the same Route
objects from the Route test as their routes. Again, the observers were
tested, then the simple mutators (setSpeed(), etc), followed by the more
complex drive() and makeTurn() functions. The test case for Car
depended on Route being implemented correctly, as well as RoadSegment
and GeoPoint.


ANALYSIS
```



```
The GeoPoint, RoadSegment, Route, and Car classes were successfully
implemented and tested. There are a few obvious possible improvements:
  - most preconditions are not checked. Though not required, it
    would probably be a good idea to do so.
  - hashCode() functions are not (meaningfully) implemented.
  - a Route could keep track of the total length of its segments;
    this would reduce the getLength() method from O(n) to O(1) time
  - Route could be implemented using an immutable-queue data
    structure for speed.
```

+5

good description

+8

true

although probably not something you want to tell your client

+4

+2

+2

excellent overview! +8

good observations

+5

34/35

```
package ps3;

import java.awt.Color;

/**
 * The <code>Car</code> class represents a vehicle travelling along a
 * <code>Route</code>.<p>
 *
 * A <code>Car</code> has an immutable part and a mutable part.  The
 * immutable fields consist of a name for the car, its color, and the
 * entire <code>Route</code> that the car can follow in its lifetime.
 * This route is called fullRoute.
 *
 * A car's mutable state consists of its current speed (measured in
 * miles per hour) and its current position along the route.  The
 * position is represented by another <code>Route</code>, called
 * remainingRoute, which is the route that the car has left to follow.
 * The remainingRoute is always a suffix of fullRoute.  The actual
 * position of the car is some fraction of the way along the first
 * segment in remainingRoute.<p>
 *
 * A car is moved along its route by two methods: <code>drive()</code>
 * and <code>makeTurn()</code>.  The <code>drive()</code> method moves
 * the car along its current road segment for a given amount of time,
 * but it will not move the car past the end of the current road
 * segment.  The <code>makeTurn()</code> method advances the car to
 * the next road segment in its route.
 *
 * @specfield name : String          // name of car
 * @specfield color : Color          // paint job
 * @specfield fullRoute : Route          // entire route that car can follow in i
ts lifetime
 *
 * @specfield speed : int               // current speed in miles per hour, >= 0
 * @specfield remainingRoute : Route    // route that car has left to follow
 * @derivedfield segment : RoadSegment  // segment on which the car is currently
 found = remainingRoute.first
 * @specfield fraction : real           // fraction of segment that car has driv
en,
 *                                      // in the range [0...1]
 *
 */
public class Car {

    private String name;
    private Color color;
    private Route fullRoute;
    private Route remainingRoute;
    private int speed;
    private double fraction;


    // Abstraction function
    //      name = name
    //      color = color
    //      speed = speed
    //      fullRoute = fullRoute
    //      remainingRoute = remainingRoute
    //      fraction = fraction

    // Representation Invariant:
    //      name, color, fullRoute, remainingRoute != null
    //      AND speed >= 0
    //      AND 0.0 <= fraction <= 1.0

    /**
     * Constructs a new <code>Car</code> and initializes its
     * name, color and route to the specified values.
     * @requires name != null, color != null, and initialRoute != null
```



```
     * @param name  the name of the car
     * @param color the color of the car
     * @param route entire route that car can follow
     * @modifies this
     * @effects constructs a new Car with the given name, color, and
     * initialRoute; zero speed; and positioned at the start of
     * initialRoute's first segment.
     */
    public Car(String name, Color color, Route initialRoute) {
        this.name = name;
    this.color = color;
    this.fullRoute = initialRoute;
    this.remainingRoute = initialRoute;
    this.speed = 0;
    this.fraction = 0;

    checkRep();
    }

    /**
     * Returns the name of the car.
     * @return this.name, the name of the car
     */
    public String getName() {
    checkRep();

        return name;
    }

    /**
     * Returns the color of the car.
     * @return this.color, the color of the car
     */
    public Color getColor() {
    checkRep();

        return color;
    }

    /**
     * Returns the current speed of the car in miles per hour.
     * @return this.speed, the current speed in miles per hour
     */
    public int getSpeed() {
    checkRep();

        return speed;
    }

    /**
     * Returns the entire route that the car can follow in its lifetime.
     * @return this.fullRoute, the entire route that the car can follow
     *         in its lifetime
     */
    public Route getFullRoute() {
    checkRep();

        return fullRoute;
    }

    /**
     * Returns the route that the car has left to follow.
     * @return this.remainingRoute, the route that the car has
     *         left to follow
     */
    public Route getRemainingRoute() {
    checkRep();

        return remainingRoute;
```

```
    }

    /**
     * Returns the segment on which the car can currently be found.
     * @return this.segment, the segment on which the car can
     *         currently be found
     */
    public RoadSegment getCurrentSegment() {
    checkRep();

        return remainingRoute.getFirst();
    }

    /**
     * Returns the fraction of the segment that the car has driven.
     * @return this.fraction, the fraction of the segment that the
     *         car has driven
     */
    public double getFractionOfSegmentTravelled() {
    checkRep();

        return fraction;
    }

    /**
     * Checks whether the car is at the end of the last <code>RoadSegment</code>
     * of its route.
     * @return true iff car is currently at the end of the last
     * <code>RoadSegment</code> of its route; false otherwise
     */
    public boolean hasArrived() {
    checkRep();

        return (fraction == 1.0 && remainingRoute.getSteps() == 1);
    }

    /**
     * Returns the <code>RoadSegment</code> that will be current after
     *         the next <code>makeTurn()</code>.
     * @return the <code>RoadSegment</code> that will be current after
     *         the next <code>makeTurn()</code>
     * @throws EndOfTripException if car is currently in the last
     * RoadSegment of its route
     */
    public RoadSegment getNextTurn() {
    checkRep();

        if (remainingRoute.getSteps() == 1)
        throw new EndOfTripException();

    return remainingRoute.popSegment().getFirst();
    }

    /**
     * Simulates driving the car along the current road segment for a
     * period of time.
     * @requires timeSlice >= 0
     * @param timeSlice amount of time to simulate, in seconds
     * @modifies this.fraction, this.location
     * @effects advances the car along the current road segment at its
     * current speed for (at most) timeSlice seconds.
     * @return amount of timeSlice left over if car reaches end of
     * current segment before timeSlice has fully elapsed; or 0, if
     * timeSlice elapses before car reaches end of current segment.
     */
    public double drive(double timeSlice) {
    checkRep();

    double speedMPS = speed/60.0/60.0; // convert speed to miles
```

magic numbers


```
                        // per second
    double distance = timeSlice*speedMPS;
    double remainingDistance = (1-fraction) * getCurrentSegment().getLength();

    if (distance <= remainingDistance)
    {
        fraction += distance / getCurrentSegment().getLength();
        return 0;
    }
    else
    {
        fraction = 1.0;
        return (distance-remainingDistance) / speedMPS;
    }

    }


    /**
     * Turns car onto the next segment of its route.
     * @requires this.fraction == 1.0
     * @modifies this.fraction, this.remainingRoute, this.segment
     * @effects turns the car from the current segment of its route
     * (which it has completely driven) to the next segment.
     * @throws EndOfTripException if car has finished the last segment
     * of its route
     */
    public void makeTurn() {
    checkRep();

    if (remainingRoute.getSteps() == 1)
        throw new EndOfTripException();

    fraction = 0.0;
    remainingRoute = remainingRoute.popSegment();

    checkRep();
    }

    /**
     * Sets the speed of the car to the specified value.
     * @requires speed >= 0
     * @param speed new speed in miles per hour
     * @modifies this.speed
     * @effects this'.speed = speed
     */
    public void setSpeed(int speed) {
    checkRep();

    this.speed = speed;

    checkRep();
    }

    /**
     * Adjusts car's position along the current road segment.
     * @requires 0.0 <= fraction <= 1.0
     * @param fraction fraction of current segment travelled
     * @modifies this.fraction
     * @effects this'.fraction = fraction
     */
    public void setFractionOfSegmentTravelled(double fraction) {
    checkRep();

        this.fraction = fraction;

    checkRep();
    }
```

```
    /**
     * Compares this <code>Car</code> with the specified <code>Object</code>
     * for equality.
     * @return true iff obj and this are behaviorally equivalent;
     *         otherwise returns false
     */
    public boolean equals(Object obj) {
    checkRep();

    return (obj == this);
    }

    /**
     * Returns a valid hash code for this.
     * @return a valid hash code for this.
     */
    public int hashCode() {
    checkRep();

        return 1;
    }

    /**
     * Returns a string representation of this car.
     * @return a string representation of this.
     */
    public String toString() {
    checkRep();

        return name;
    }

    /**
     * Checks to see if the representation invariant is being violated
     * and if so, throws a RuntimeException
     * @throws RuntimeException if representation invariant is violated
     */
    private void checkRep() throws RuntimeException {
        if (name == null || color == null || fullRoute == null
        || remainingRoute == null)
        throw new RuntimeException("some field was null");

    if (speed < 0)
        throw new RuntimeException("speed was negative");

    if (fraction < 0.0 || fraction > 1.0)
        throw new RuntimeException("fraction out of bounds");
    }

}
```



```
package ps3;

import junit.framework.*;
import java.util.Iterator;
import java.awt.Color;


/**
 * Contains unit test for the Car class
 */

public class CarTest extends TestCase
{
    private static final double TOLERANCE = GeoPointTest.TOLERANCE;

    private GeoPoint gpDowntown;
    private GeoPoint gpWest;
    private GeoPoint gpEast;
    private GeoPoint gpNorth;

    private RoadSegment gsEast;
    private RoadSegment gsWest;
    private RoadSegment gsNorth;
    private RoadSegment gsEast2;
    private RoadSegment gsWest2;
    private RoadSegment gsDiag;
    private RoadSegment gsDiag2;
    private RoadSegment gsZero;

    private Route rDW, rDW2, rDWD, rWDN, rDWN;

    private Car cFoo, cBar, cBaz;

    public CarTest(String name)
    {
    super(name);
    }

    public static Test suite()
    {
    return new TestSuite(CarTest.class);
    }

    protected void setUp() {
        //                              north       east
        // +1 mile                       14488      19579
        gpDowntown = new GeoPoint(42358333, -71060278);
        gpWest = new GeoPoint(42358333, -71079857);
        gpEast = new GeoPoint(42358333, -71040699);
        gpNorth = new GeoPoint(42372821, -71060278);

        gsEast = new RoadSegment("East", gpDowntown, gpEast);
        gsWest = new RoadSegment("West", gpDowntown, gpWest);
        gsNorth = new RoadSegment("North", gpDowntown, gpNorth);
        gsEast2 = new RoadSegment("East", gpEast, gpDowntown);
        gsWest2 = new RoadSegment("West", gpWest, gpDowntown);
        gsDiag = new RoadSegment("NE", gpWest, gpNorth);
        gsDiag2 = new RoadSegment("NorthEast", gpWest, gpNorth);
        gsZero = new RoadSegment("Zero", gpDowntown, gpDowntown);

    rDW = new Route(gsWest);
    rDW2 = new Route(gsWest);
    rDWD = new Route(gsWest).addSegment(gsWest2);
    rWDN = new Route(gsWest2).addSegment(gsNorth);
    rDWN = new Route(gsWest).addSegment(gsDiag);

    cFoo = new Car("Foo", Color.RED, rDW);
    cBar = new Car("Bar", Color.GREEN, rDWD);
    cBaz = new Car("Baz", Color.BLUE, rWDN);
```

```
    }


    // The following set of tests tests the observer methods before
    // any other testing is done. Hence, they're really rather
    // boring, but future tests will require the observers to work, so
    // it's best to catch any errors here if possible. The observers
    // will also be tested further in other tests..

    public void testGetName()
    {
    assertEquals("Foo", cFoo.getName());
    assertEquals("Bar", cBar.getName());
    assertEquals("Baz", cBaz.getName());
    }


    public void testGetColor()
    {
    assertEquals(cFoo.getColor(), Color.RED);
    assertEquals(cBar.getColor(), Color.GREEN);
    assertEquals(cBaz.getColor(), Color.BLUE);
    }


    public void testGetSpeed()
    {
    assertEquals(0.0, cFoo.getSpeed(), TOLERANCE);
    assertEquals(0.0, cBar.getSpeed(), TOLERANCE);
    assertEquals(0.0, cBaz.getSpeed(), TOLERANCE);
    }


    public void testGetFullRoute()
    {
    assertEquals(rDW, cFoo.getFullRoute());
    assertEquals(rDWD, cBar.getFullRoute());
    assertEquals(rWDN, cBaz.getFullRoute());
    }


    public void testGetRemainingRoute()
    {
    assertEquals(rDW, cFoo.getRemainingRoute());
    assertEquals(rDWD, cBar.getRemainingRoute());
    assertEquals(rWDN, cBaz.getRemainingRoute());
    }


    public void testGetCurrentSegment()
    {
    assertEquals(gsWest, cFoo.getCurrentSegment());
    assertEquals(gsWest, cBar.getCurrentSegment());
    assertEquals(gsWest2, cBaz.getCurrentSegment());
    }


    public void testGetFraction()
    {
    assertEquals(0.0, cFoo.getFractionOfSegmentTravelled(), TOLERANCE);
    assertEquals(0.0, cBar.getFractionOfSegmentTravelled(), TOLERANCE);
    assertEquals(0.0, cBaz.getFractionOfSegmentTravelled(), TOLERANCE);
    }


    public void testHasArrived()
    {
    assertEquals(false, cFoo.hasArrived());
```



```
    assertEquals(false, cBar.hasArrived());
    assertEquals(false, cBaz.hasArrived());
    }


    public void testGetNextTurn()
    {
    assertEquals(gsWest2, cBar.getNextTurn());
    assertEquals(gsNorth, cBaz.getNextTurn());

    try
    {
        cFoo.getNextTurn();
        fail("getNextTurn() didn't throw an EndOfTripException");
    }
    catch (EndOfTripException e)
    {
    }

    }


    public void testSetFraction()
    {
    cFoo.setFractionOfSegmentTravelled(0.3);
    assertEquals(0.3, cFoo.getFractionOfSegmentTravelled(), TOLERANCE);

    cFoo.setFractionOfSegmentTravelled(1.0);
    assertEquals(1.0, cFoo.getFractionOfSegmentTravelled(), TOLERANCE);

    cFoo.setFractionOfSegmentTravelled(0.0);
    assertEquals(0.0, cFoo.getFractionOfSegmentTravelled(), TOLERANCE);
    }


    public void testSetSpeed()
    {
    cFoo.setSpeed(20);
    assertEquals(20, cFoo.getSpeed());

    cFoo.setSpeed(100);
    assertEquals(100, cFoo.getSpeed());

    cFoo.setSpeed(0);
    assertEquals(0, cFoo.getSpeed());
    }


    public void testEquals()
    {
    assertEquals("not equal to self", cFoo, cFoo);
    assertTrue("equal to null", !cFoo.equals(null));
    assertTrue("equal to string", !cFoo.equals("test"));
    assertTrue("equal to identical object",
        !cFoo.equals(new Car("Foo", Color.RED, rDW)));
    assertTrue("totally different objects equal", !cFoo.equals(cBar));
    assertTrue("same name and color, different route equal",
        !cFoo.equals(new Car("Foo", Color.RED, rDWD)));
    assertTrue("same name and route, different color equal",
        !cFoo.equals(new Car("Foo", Color.BLUE, rDW)));
    assertTrue("same color and route, different name equal",
        !cFoo.equals(new Car("Bar", Color.RED, rDW)));

    cFoo.setSpeed(10);
    assertTrue("same car, different speed equal",
        !cFoo.equals(new Car("Foo", Color.RED, rDW)));

    cBar.setFractionOfSegmentTravelled(1.0);
    assertTrue("same car, different fraction equal",
```

```
        !cBar.equals(new Car("Bar", Color.GREEN, rDWD)));

    cBar.makeTurn();
    assertTrue("same car, different segment equal",
        !cBar.equals(new Car("Bar", Color.GREEN, rDWD)));
    }


    public void testHashCode()
    {
    assertEquals("hashCode not equal to self", cFoo.hashCode(),
        cFoo.hashCode());

    assertEquals("hashCode not equal to identical object",
        cFoo.hashCode(),
        new Car("Foo", Color.RED, rDW).hashCode());
    }


    public void testDrive()
    {
    // make sure driving at zero speed does nothing
    assertEquals("bad return value for drive()", 0.0, cBar.drive(30),
        TOLERANCE);
    assertEquals("fraction was non-zero after zero-speed drive",
        0.0, cBar.getFractionOfSegmentTravelled(), TOLERANCE);

    // make sure driving for zero time does nothing
    cBar.setSpeed(60); // so 1min = 1mi
    assertEquals("bad return value for drive()",
        0.0, cBar.drive(0), TOLERANCE);
    assertEquals("fraction was non-zero after zero-time drive",
        0.0, cBar.getFractionOfSegmentTravelled(), TOLERANCE);

    // now try actually driving
    assertEquals("bad return value for drive()",
        0.0, cBar.drive(20), TOLERANCE);
    assertEquals("wrong fraction after 20-second drive",
        .333, cBar.getFractionOfSegmentTravelled(), TOLERANCE);

    // make sure driving past the end of the segment does the
    // right thing
    assertEquals("bad return value for 60s drive()",
        20.0, cBar.drive(60), TOLERANCE);
    assertEquals("wrong fraction after 60-second drive",
        1.0, cBar.getFractionOfSegmentTravelled(), TOLERANCE);
    assertEquals("bad return value for 100s drive()",
        100.0, cBar.drive(100), TOLERANCE);
    assertEquals("wrong fraction after 100-second drive",
        1.0, cBar.getFractionOfSegmentTravelled(), TOLERANCE);
    }


    public void testMakeTurn()
    {
    cBar.setSpeed(60);
    assertEquals("bad return value for drive()",
        40.0, cBar.drive(100), TOLERANCE);
    assertEquals("already arrived, at end of first segment",
        false, cBar.hasArrived());

    cBar.makeTurn();
    assertEquals(gsWest2, cBar.getCurrentSegment());
    assertEquals(0.0, cBar.getFractionOfSegmentTravelled(), TOLERANCE);
    assertEquals(false, cBar.hasArrived());
    assertEquals(rDWD, cBar.getFullRoute());

    assertEquals("remainingRoute is wrong",
        1, cBar.getRemainingRoute().getSteps());
```

correct test for equality in your code, but not reflected in your test


```
    assertEquals("remainingRoute is wrong",
        gsWest2, cBar.getRemainingRoute().getFirst());

    assertEquals("bad return value for drive()",
        0.0, cBar.drive(20), TOLERANCE);
    assertEquals("wrong fraction after 20-second drive",
        .333, cBar.getFractionOfSegmentTravelled(), TOLERANCE);

    // make sure driving past the end of the segment does the
    // right thing
    assertEquals("bad return value for 60s drive()",
        20.0, cBar.drive(60), TOLERANCE);
    assertEquals("wrong fraction after 60-second drive",
        1.0, cBar.getFractionOfSegmentTravelled(), TOLERANCE);
    assertEquals("bad return value for 100s drive()",
        100.0, cBar.drive(100), TOLERANCE);

    assertEquals("wrong fraction after 100-second drive",
        1.0, cBar.getFractionOfSegmentTravelled(), TOLERANCE);

    assertEquals(true, cBar.hasArrived());

    try
    {
        cBar.makeTurn();
        fail("makeTurn() should have thrown an EndOfTripException");
    }
    catch (EndOfTripException e)
    {
        // test passed
    }
    }


}
```

```
package ps3;

/**
 * EndOfTripException is an unchecked exception that occurs when a Car
 * is asked for its next turn when it has no more turns to make.
 */
public class EndOfTripException extends RuntimeException {

    /**
     * @effects Constructs an EndOfTripException with no message.
     */
    public EndOfTripException () {
        super ();
    }

    /**
     * @effects Constructs an EndOfTripException with a descriptive
     * error message.
     */
    public EndOfTripException (String message) {
        super (message);
    }
}
```



```
package ps3;

/**
 * <p>A <code>GeoPoint</code> models a point on the earth.
 * <code>GeoPoint</code>s are immutable.
 *
 * <p>North latitudes and east longitudes are represented by positive numbers.
 * South latitudes and west longitudes are represented by negative numbers.
 *
 * <p>The code may assume that the represented points are nearby Boston.
 *
 * <p><b>Implementation hint</b>:<br> Boston is at approximately 42
 * deg. 21 min. 30 sec. N latitude and 71 deg. 03 min. 37 sec. W
 * longitude.  There are 60 minutes per degree, and 60 seconds per
 * minute.  So, in decimal, these corespond to 42.358333
 * latitude and -71.060278 longitude.  The constructor takes
 * integers in millionths of degrees.  To create a new <code>GeoPoint</code>
 * located in Boston, use: <tt>GeoPoint boston = new GeoPoint(42358333,
 * -71060278);</tt>
 *
 * <p>Near Boston, there are approximately 69.023 miles per degree of
 * latitude and 51.075 miles per degree of longitude.  An
 * implementation may use these values when determining distances and
 * headings.
 *
 * @specfield  latitude :  real     // measured in degrees latitude
 * @specfield  longitude : real     // measured in degrees longitude
 *
 * Implemented and passed GeoPointTest today -- drkp 2004/02/22
 *
 **/

public class GeoPoint {

    private int latitude, longitude; // in millionths of a degree

    // Abstraction function
    //     AF(r) = GeoPoint g such that
    //                 g.latitude = r.latitude * 1000000
    //                 g.longitude = r.longitude * 1000000
    //                     (conversion between degrees and millionths
    //                       of degrees in representation)

    // Representation Invariant:
    //     latitude is between +/- 90 * 10^6 inclusive
    //     longitude is between +/- 180 * 10^6 inclusive

    /**
     * Approximation used to determine distances and headings using a
     * "flat earth" simplification.  */
    public static final double MILES_PER_DEGREE_LATITUDE = 69.023;

    /**
     * Approximation used to determine distances and headings using a
     * "flat earth" simplification.  */
    public static final double MILES_PER_DEGREE_LONGITUDE = 51.075;

    // Constructors

    /**
     * Constructs a new <code>GeoPoint</code> representing
     * the location defined by the given latitude and longitude.
     * @requires  the point given by (latitude, longitude) in millionths
     *            of degrees is near Boston
     *
     * @modifies this
     * @effects constructs a <code>GeoPoint</code> from a latitude
     *          and longitude given in millionths of degrees.
     **/
```

```
    public GeoPoint(int latitude, int longitude) {
        checkRep();

        this.latitude = latitude;
    this.longitude = longitude;
    }

    // Observers

    /**
     * Returns the latitude of this <code>GeoPoint</code>,
     * in millionths of degrees.
     * @return the latitude of the <code>GeoPoint</code> object,
     *         in millionths of degrees.
     */
    public int getLatitude() {
        checkRep();

        return latitude;
    }

    /**
     * Returns the longitude of this <code>GeoPoint</code>,
     * in millionths of degrees.
     * @return the longitude of the <code>GeoPoint</code> object,
     *         in millionths of degrees.
     */
    public int getLongitude() {
        checkRep();

        return longitude;
    }

    /**
     * Computes the distance between this <code>GeoPoint</code>
     * and the specified <code>GeoPoint</code>.
     *
     * @requires gp != null
     * @param gp the destination endpoint
     *
     * @return a close approximation of as-the-crow-flies distance, in
     * miles, from this to gp.  As-the-crow-flies is the smaller arc
     * of the great circle connecting the two points.  This is very
     * close to the straight line distance for relatively short
     * distances on the surface of the earth.  An implementation may
     * return the straight-line distance.
     **/
    public double distanceTo(GeoPoint gp) {
        checkRep();

        double latDist, longDist;
    latDist = (gp.latitude - latitude)
        * MILES_PER_DEGREE_LATITUDE / 1000000;
    longDist = (gp.longitude - longitude)
        * MILES_PER_DEGREE_LONGITUDE / 1000000;

    return Math.sqrt(latDist * latDist + longDist * longDist);

    }

    /**
     * Computes the compass heading from this <code>GeoPoint</code>
     * to the specified <code>GeoPoint</code>.
     *
     * @requires gp != null && !this.equals(gp)
     * @param gp the destination endpoint
     *
     * @return a close approximation of compass heading h from this to
     * gp, in degrees, using the flat-surface, near Boston
```

watch for mathing


```
     * approximation, such that 0 &lt;= h &lt; 360.  In compass
     * headings, north = 0, east = 90, south = 180, and west = 270.
     **/
    public double headingTo(GeoPoint gp) {
        checkRep();

    double theta;
        double latDist, longDist;

    latDist = (gp.latitude - latitude)
        * MILES_PER_DEGREE_LATITUDE / 1000000;
    longDist = (gp.longitude - longitude)
        * MILES_PER_DEGREE_LONGITUDE / 1000000;

    theta = Math.atan2(latDist, longDist);
    theta = Math.toDegrees(theta);

    if (theta > 90)
        return 450-theta;
    else
        return 90-theta;
    }

    /**
     * Compares this <code>GeoPoint</code>
     * to the specified <code>Object</code> for equality.
     *
     * @return obj != null && (obj instanceof <code>GeoPoint</code>)
     *         && obj.latitude = this.latitude && obj.longitude = this.longitude
     **/
    public boolean equals(Object obj) {
        checkRep();

        if (obj == null || !(obj instanceof GeoPoint))
        return false;

    GeoPoint o = (GeoPoint) obj;

    return (o.latitude == latitude && o.longitude == longitude);
    }

    /**
     * Returns a hash code for this <code>GeoPoint</code>.
     * @return a valid hashcode for this.
     **/
    public int hashCode() {
        checkRep();

        // This implementation will work, but you may want to modify it later
        // for improved performance.  If you do change the implementation, make
        // sure it holds the hashCode invariant.  That is, if equals returns
        // true for two objects, then they must have the same hashCode.
        return 1;
    }

    /**
     * Returns a string representation of this <code>GeoPoint</code>.
     * @return a string representation of this.
     **/
    public String toString() {
        checkRep();

    String str;

    str = latitude + " " + longitude;

    return str;
    }
```

```
    /**
     * Checks to see if the representation invariant is being
     * violated and if so, throws a RuntimeException.
     * @throws RuntimeException if representation invariant is violated
     */
    private void checkRep() throws RuntimeException {
    if (Math.abs(latitude) > 90000000)
        throw new RuntimeException("latitude out of bounds");

    if (Math.abs(longitude) > 180000000)
        throw new RuntimeException("longitude out of bounds");
    }


} // GeoPoint
```



```
package ps3;

import junit.framework.*;

/**
 * Contains suite of unit tests for the <code>GeoPoint</code> class.
 */
public class GeoPointTest extends TestCase {

    public static final double TOLERANCE = 0.01;

    private GeoPoint gpDowntown;
    private GeoPoint gpWest;
    private GeoPoint gpEast;
    private GeoPoint gpSouthEast;
    private GeoPoint gpNorth;
    private GeoPoint gpSouth;
    private GeoPoint gpMit;

    public GeoPointTest(String name) {
        super(name);
    }

    // Tell JUnit to run all the test inside this class
    public static Test suite() {
        return new TestSuite(GeoPointTest.class);
    }

    protected void setUp() {

        gpDowntown = new GeoPoint(42358333, -71060278);
        gpWest = new GeoPoint(42358333, -71079857);
        gpEast = new GeoPoint(42358333, -71040699);
        gpNorth = new GeoPoint(42372821, -71060278);
        gpSouthEast = new GeoPoint(42343845, -71040699);
        gpSouth = new GeoPoint(42343845, -71060278);
    }

    public void testEquals() {
        assertTrue("instance not equal to itself.", gpNorth.equals(gpNorth));

        assertTrue(
            "instance not equal to copy.",
            gpNorth.equals(new GeoPoint(42372821, -71060278)));

        assertTrue(
            "totally different objects are equal.",
            !gpNorth.equals(gpEast));
    }

    public void testEquals2() {
        assertTrue("equals returns true for null.", !gpNorth.equals(null));
        assertTrue("equals returns true for a String.", !gpNorth.equals("foo"));
    }

    public void testImmutability() {

        GeoPoint gpReceiver = new GeoPoint(42358333, -71060278);
        GeoPoint gpArgument = new GeoPoint(42343845, -71060278);

        gpReceiver.distanceTo(gpArgument);
        assertTrue(
            "distanceTo mutates receiver.",
            gpReceiver.equals(gpDowntown));
        assertTrue("distanceTo mutates argument.", gpArgument.equals(gpSouth));

        gpReceiver.hashCode();
        assertTrue(
            "hashCode mutates receiver.",
```

```
            gpReceiver.equals(gpDowntown));

        gpReceiver.headingTo(gpArgument);
        assertTrue(
            "headingTo mutates receiver.",
            gpReceiver.equals(gpDowntown));
        assertTrue("headingTo mutates argument.", gpArgument.equals(gpSouth));

        gpReceiver.toString();
        assertTrue("toString mutates receiver.", gpReceiver.equals(gpDowntown));

    }

    public void testDistanceTo() {
        assertEquals(
            "From North to Downtown 1 mile",
            1.0,
            gpNorth.distanceTo(gpDowntown),
            TOLERANCE);
        assertEquals(
            "From North to Downtown equal to Downtown to North",
            gpDowntown.distanceTo(gpNorth),
            gpNorth.distanceTo(gpDowntown),
            TOLERANCE);
        assertEquals(
            "From North to North 0 miles",
            0.0,
            gpNorth.distanceTo(gpNorth),
            TOLERANCE);
    }

    public void testHeadingTo() {
        assertEquals(
            "Downtown headingTo East should be 90",
            90.0,
            gpDowntown.headingTo(gpEast),
            TOLERANCE);
        assertEquals(
            "Downtown headingTo West should be 270",
            270.0,
            gpDowntown.headingTo(gpWest),
            TOLERANCE);

        double h = gpDowntown.headingTo(gpNorth);
        assertTrue(
            "Downtown headingTo North should be 0",
            ((h - 0.0) <= TOLERANCE) || ((360.0 - h) <= TOLERANCE));

        assertEquals(
            "Downtown headingTo South should be 180",
            180.0,
            gpDowntown.headingTo(gpSouth),
            TOLERANCE);
        assertEquals(
            "West headingTo North should be 45",
            45.0,
            gpWest.headingTo(gpNorth),
            TOLERANCE);
        assertEquals(
            "West headingTo South should be 135",
            135.0,
            gpWest.headingTo(gpSouth),
            TOLERANCE);
        assertEquals(
            "East headingTo North should be 315",
            315.0,
            gpEast.headingTo(gpNorth),
            TOLERANCE);
        assertEquals(
```



```
            "East headingTo South should be 225",
            225.0,
            gpEast.headingTo(gpSouth),
            TOLERANCE);

    }

    public void testHashCode() {
        GeoPoint gpDowntown2 = new GeoPoint(42358333, -71060278);

        assertTrue(
            ".equals() objects must have the same .hashCode()",
            gpDowntown.hashCode() == gpDowntown2.hashCode());
    }

}
```

```
package ps3;

/**
 * A <code>RoadSegment</code> models a straight line segment on the earth.
 * <code>RoadSegment</code>s are immutable.<p>
 *
 * <p>
 *
 * A compass heading is a nonnegative real number less than 360.
 * In compass headings, north = 0, east = 90, south = 180, and west = 270.<p>
 *
 * When used in a map, a <code>RoadSegment</code> might represent part of a stre
et or
 * a boundary.
 * As an example usage, this map
 * <pre>
 *  Penny Lane  a
 *              |
 *              i--j--k  Abbey Road
 *              |
 *              z
 * </pre>
 * could be represented by the following <code>RoadSegment</code>s:
 * ("Penny Lane", a, i), ("Penny Lane", z, i),
 * ("Abbey Road", i, j), and ("Abbey Road", j, k).
 *
 * @specfield  name : String       // name of the route part identified
 * @specfield  p1 : GeoPoint       // first endpoint of the segment
 * @specfield  p2 : GeoPoint       // second endpoint of the segment
 * @derivedfield length : real     // straight-line distance between p1
 *                                    and p2, in miles
 * @derivedfield  heading : angle  // compass heading from p1 to p2, in degrees
 *
 * Implemented and passed RoadSegmentTest today -- drkp 2004/02/22
 *
 **/
public class RoadSegment {

    private String name;
    private GeoPoint p1, p2;

    // Abstraction function
    //      name = name
    //      p1 = p1
    //      p2 = p2

    // Representation Invariant:
    //      name, p1, p2 != null

    // Constructors

    /**
     * Constructs a new <code>RoadSegment</code>
     * initializing its name and endpoints to the specified values.
     *
     * @requires name != null && p1 != null && p2 != null
     *
     * @param name the name of the route part
     * @param p1   first endpoint of the segment
     * @param p2   second endpoint of the segment
     *
     * @modifies this
     * @effects constructs a new RoadSegment with the specified name
     * and endpoints
     **/
    public RoadSegment(String name, GeoPoint p1, GeoPoint p2) {
        this.name = name;
    this.p1 = p1;
    this.p2 = p2;
```

indentation


```
    }

    // Producers

    /**
     * Returns a new <code>RoadSegment</code> like this one but with its
     * endpoints reversed.
     * @return a new <code>RoadSegment</code> gs such that
     *       gs.name = this.name
     *    && gs.p1 = this.p2
     *    && gs.p2 = this.p1
     **/
    public RoadSegment reverse() {
    checkRep();

    return new RoadSegment(name, p2, p1);
    }

    // Observers

    /**
     * Returns the name of this <code>RoadSegment</code>.
     * @return the name of this <code>RoadSegment</code>.
     * A name given to a <code>RoadSegment</code> object that can differentiate
     * between two <code>RoadSegment</code> objects with identical GeoPoint endp
oints.
     * Equality between <code>RoadSegment</code> objects requires that the names
 be
     * equal String objects.
     */
    public String getName() {
    checkRep();

    return name;
    }

    /**
     * Returns the first endpoint of the segment.
     * @return first endpoint of the segment.
     */
    public GeoPoint getP1() {
        checkRep();

    return p1;
    }

    /**
     * Returns the second endpoint of the segment.
     * @return second endpoint of the segment.
     */
    public GeoPoint getP2() {
    checkRep();

        return p2;
    }

    /**
     * Returns the length of the segment.
     * @return the length of the segment.
     */
    public double getLength() {
    checkRep();

    return p1.distanceTo(p2);
    }

    /**
     * Returns a point at some fractional distance along the segment.
     *
```

more indentation

corridor caching

```
     * @requires 0 <= fraction <= 1
     * @param fraction fraction of distance along segment
     *
     * @return point p such that p lies on the segment and
     *         p.distanceTo (p1) / length == fraction
     */
    public GeoPoint interpolate(double fraction) {
        checkRep();

    return new GeoPoint((int) (p1.getLatitude()
        + fraction * (p2.getLatitude() - p1.getLatitude())),
        (int) (p1.getLongitude()
        + fraction * (p2.getLongitude() - p1.getLongitude())));
    }

    /**
     * Returns the heading of this segment.
     * @requires this.length != 0
     *
     * @return the compass heading from p1 to p2, in degrees.
     **/
    public double getHeading() {
    checkRep();

        return p1.headingTo(p2);
    }

    /**
     * Compares the specified <code>Object</code> with
     * this <code>RoadSegment</code> for equality.
     * @return    gs != null && (gs instanceof <code>RoadSegment</code>)
     *          && gs.name = this.name && gs.p1 = this.p1 && gs.p2 = this.p2
     **/
    public boolean equals(Object obj) {
    checkRep();

    if (obj == null || !(obj instanceof RoadSegment))
        return false;

    RoadSegment gs = (RoadSegment) obj;

    return gs.name.equals(this.name) && gs.p1.equals(this.p1)
        && gs.p2.equals(this.p2);
    }

    /**
     * Returns a valid hashcode for this.
     * @return a valid hashcode for this.
     **/
    public int hashCode() {
        // This implementation will work, but you may want to modify it later
        // for improved performance.  If you do change the implementation, make
        // sure it holds the hashCode invariant.  That is, if equals returns
        // true for two objects, then they must have the same hashCode.
    checkRep();
        return 1;
    }

    /**
     * Returns a string representation of this <code>RoadSegment</code>.
     * @return a string representation of this.
     **/
    public String toString() {
    checkRep();
        return name;
    }

    /**
     * Checks to see if the representation invariant is being violated
```

consider caching

indenting


```
     * and if so, throws a RuntimeException
     * @throws RuntimeException if representation invariant is violated
     */
    private void checkRep() throws RuntimeException {
    if (name == null || p1 == null || p2 == null)
        throw new RuntimeException("RoadSegment's GeoPoint is null");
    }


} // RoadSegment
```

```
package ps3;

import junit.framework.*;

/**
 * Contains suite of unit tests for the <code>RoadSegment</code> class.
 */
public class RoadSegmentTest extends TestCase {

    private static final double TOLERANCE = GeoPointTest.TOLERANCE;

    private GeoPoint gpDowntown;
    private GeoPoint gpWest;
    private GeoPoint gpEast;
    private GeoPoint gpNorth;

    private RoadSegment gsEast;
    private RoadSegment gsWest;
    private RoadSegment gsNorth;
    private RoadSegment gsEast2;
    private RoadSegment gsWest2;
    private RoadSegment gsDiag;
    private RoadSegment gsDiag2;
    private RoadSegment gsZero;

    public RoadSegmentTest(String name) {
        super(name);
    }

    // Tell JUnit to run all the tests in this class
    public static Test suite() {
        return new TestSuite(RoadSegmentTest.class);
    }

    // JUnit calls setUp() before each test__ method is run
    protected void setUp() {
        //                          north       east
        // +1 mile                    14488       19579
        gpDowntown = new GeoPoint(42358333, -71060278);
        gpWest = new GeoPoint(42358333, -71079857);
        gpEast = new GeoPoint(42358333, -71040699);
        gpNorth = new GeoPoint(42372821, -71060278);

        gsEast = new RoadSegment("East", gpDowntown, gpEast);
        gsWest = new RoadSegment("West", gpDowntown, gpWest);
        gsNorth = new RoadSegment("North", gpDowntown, gpNorth);
        gsEast2 = new RoadSegment("East", gpEast, gpDowntown);
        gsWest2 = new RoadSegment("West", gpWest, gpDowntown);
        gsDiag = new RoadSegment("NE", gpWest, gpNorth);
        gsDiag2 = new RoadSegment("NorthEast", gpWest, gpNorth);
        gsZero = new RoadSegment("Zero", gpDowntown, gpDowntown);
    }

    public void testEquals() {
        assertEquals("Self equality", gsNorth, gsNorth);

        assertTrue("equals returns true for null", !gsNorth.equals(null));
        assertTrue(
            "equals returns true for String",
            !gsNorth.equals(new String("foo")));

        assertEquals(
            "not equal to copy",
            gsNorth,
            new RoadSegment("North", gpDowntown, gpNorth));
        assertTrue(
            "totally different objects are equal.",
            !gsNorth.equals(gsEast));
        assertTrue(
```



```
            "same points, different name are equal.",
            !gsDiag.equals(gsDiag2));
        assertTrue(
            "same name, different points are equal.",
            !gsEast.equals(gsEast2));

    }

    public void testEquals2() {
        // make segment components which are equal by value, but which are
        // not the same object
        String north = "Nor";
        north += "th";
        GeoPoint gpDowntown2 = new GeoPoint(42358333, -71060278);
        GeoPoint gpNorth2 = new GeoPoint(42372821, -71060278);
        RoadSegment gsNorth2 = new RoadSegment(north, gpDowntown2, gpNorth2);

        assertTrue(
            "Segment equality should use value equality, not reference equality",
            gsNorth.equals(gsNorth2));

        assertTrue(
            "equals(non-RoadSegment) should be false",
            !gsNorth2.equals("aString"));

        assertTrue("equals(null) should be false", !gsNorth2.equals(null));
    }

    public void testReverse() {
        assertTrue(
            "Reversed segment is not equal same segment reversed.",
            gsEast.reverse().equals(gsEast.reverse()));
        assertTrue(
            "Twice reversed segment is not same as initial.",
            gsEast.reverse().reverse().equals(gsEast));
        assertTrue(
            "New reversed item is not equal to its reversal.",
            gsWest.equals(gsWest2.reverse()));
        assertTrue(
            "New reversed item is not equal to its reversal.",
            gsWest.reverse().equals(gsWest2));
        assertTrue(
            "Segment equal to its reversal.",
            !gsEast.reverse().equals(gsEast));
        assertTrue(
            "Segment reversed twice is equal to its reversal.",
            !gsEast.reverse().reverse().equals(gsEast.reverse()));
        assertTrue(
            "Reversed segment reversed equals reversed.",
            !gsWest.reverse().equals(gsWest2.reverse()));
        assertTrue(
            "Reversed zero segment doesn't equal itself.",
            gsZero.reverse().equals(gsZero));
    }

    public void testName() {
        assertEquals("name() doesn't work.", "East", gsEast.getName());
    }

    public void testP1() {
        assertEquals(gpDowntown, gsEast.getP1());
    }

    public void testP2() {
        assertEquals(gpDowntown, gsEast2.getP2());
    }

    public void testLength() {
        assertEquals("East 1 mile", 1.0, gsEast.getLength(), TOLERANCE);
```

```
        assertEquals("West 1 mile", 1.0, gsWest.getLength(), TOLERANCE);
        assertEquals("North 1 mile", 1.0, gsNorth.getLength(), TOLERANCE);
        assertEquals("1.414 miles", 1.414, gsDiag.getLength(), TOLERANCE);
        assertEquals("0 miles", 0, gsZero.getLength(), TOLERANCE);
    }

    public void testHeading() {
        assertEquals("East should be 90", 90.0, gsEast.getHeading(), TOLERANCE);
        assertEquals(
            "West should be 270",
            270.0,
            gsWest.getHeading(),
            TOLERANCE);
        double nh = gsNorth.getHeading();
        assertTrue(
            "North heading (" + nh + ") is less than zero.",
            !(nh < 0.0));
        assertTrue(
            "North heading (" + nh + ") is greater or equal to 360.",
            !(nh >= 360.0));
        if (nh > TOLERANCE) {
            // we know nh is in [0..360); maybe it's just under 360, which is ok
ay too
            double delta = Math.abs(360.0 - nh);
            if (delta > TOLERANCE)
                fail("North heading expected: 0 or 359.999 but got " + nh);
        }
        assertEquals(
            "South heading should be 180",
            180.0,
            gsNorth.reverse().getHeading(),
            TOLERANCE);
    }


    public void testInterpolate() {

        GeoPoint gpWestModif = new GeoPoint(42358333, -71079858);
        // gpNorth is (42372821, -71060278);
        // Delta latitude is: 14488; Delta longitude is 19580
        RoadSegment gsDiagModif = new RoadSegment("DiagModif",gpWestModif,gpNorth
);

        // Testing interpolation at fraction 0.0
        GeoPoint gpInter = gsDiag.interpolate(0.0);
        assertTrue("Interpolating a 0.0 fraction doesn't return correct GeoPoint",
                   (Math.abs(gpInter.getLatitude()-gpWestModif.getLatitude()) <=
 2));
        assertTrue("Interpolating a 0.0 fraction doesn't return correct GeoPoint",
                   (Math.abs(gpInter.getLongitude()-gpWestModif.getLongitude())
<= 2));

        // Testing interpolation at fraction 1.0
        gpInter = gsDiag.interpolate(1.0);
        assertTrue("Interpolating a 1.0 fraction doesn't return correct GeoPoint",
                   (Math.abs(gpInter.getLatitude()-gpNorth.getLatitude()) <= 2))
;
        assertTrue("Interpolating a 1.0 fraction doesn't return correct GeoPoint",
                   (Math.abs(gpInter.getLongitude()-gpNorth.getLongitude()) <= 2
));

        // Testing interpolation at fraction 0.25
        gpInter = gsDiag.interpolate(0.25);
        // At 1/4 we need +3622 and +4895
        GeoPoint gpQuarter = new GeoPoint(42361955,-71074963);
        assertTrue("Interpolating a 0.25 fraction doesn't return correct GeoPoint",
                   (Math.abs(gpInter.getLatitude()-gpQuarter.getLatitude()) <= 2
));
        assertTrue("Interpolating a 0.25 fraction doesn't return correct GeoPoint",
```



```
                   (Math.abs(gpInter.getLongitude()-gpQuarter.getLongitude()) <=
 2));


    }

    public void testHashCode() {
        GeoPoint gpDowntown2 = new GeoPoint(42358333, -71060278);
        GeoPoint gpNorth2 = new GeoPoint(42372821, -71060278);
        RoadSegment gsNorth2 = new RoadSegment("North", gpDowntown2, gpNorth2);

        assertTrue(
            ".equals() objects must have the same .hashCode()",
            gsNorth.hashCode() == gsNorth2.hashCode());
    }

}
```

```
package ps3;

import java.util.Iterator;
import java.util.ListIterator;
import java.util.ArrayList;
import java.util.Collections;

/**
 * The <code>Route</code> class represents an immutable sequence of
 * <code>RoadSegments</code>.  <p>
 *
 * While each <code>Route</code> instance is immutable, a new
 * <code>Route</code> can be constructed by adding a segment to the
 * end of a <code>Route</code>.  An added segment must be properly
 * oriented; that is, its p1 field must correspond to the end of the
 * original <code>Route</code>, and its p2 field corresponds to the
 * end of the new <code>Route</code>.<p>
 *
 * A new <code>Route</code> can also be constructed by popping off the
 * first segment on the route.  This capability is used when a vehicle
 * is following the route; once it has driven the first segment, it
 * pops it off to get the remainder of the route it should follow.
 * The new <code>Route</code> must have at least one segment, so only
 * a <code>Route</code> with more than one segment can be popped.<p>
 *
 * Because a <code>Route</code> is not necessarily straight, its
 * length -- the distance traveled by following the path from start to
 * end -- is not necessarily the same as the distance along a straight
 * line between its endpoints.<p>
 *
 *
 * @specfield segments : sequence       // a sequence of RoadSegments that make u
p this Route
 * @derivedfield first : RoadSegment    // first segment on route = segments[0]
 * @derivedfield last : RoadSegment     // last segment on route = segments[segme
nts.length - 1]
 * @derivedfield steps : int            // number of segments in the route = segm
ents.length
 * @derivedfield start : GeoPoint       // location of the start of the route = f
irst.p1
 * @derivedfield end : GeoPoint         // location of the end of the route = las
t.p2
 * @derivedfield length : real          // total length of the route, in miles =
sum_i segments[i].length
 *
 * Implemented and passed RouteTest today -- drkp 2004/02/23
 *
 */
public class Route {

    ArrayList segments;

    // Abstraction function
    //      segments = segments

    // Representation Invariant:
    //      segments != NULL
    //      AND segments contains at least one element
    //      AND segments contains only RoadSegment elements
    //      AND \forall i>0, segments(i).P1 = segments(i-1).P2

    // Constructor

    /**
     * Constructs a new <code>Route</code> consisting only of the
     * specified segment.
     *
     * @requires seg != null
     * @param seg the only road segment that will be part of this route
```

OK, could be more descriptive

good


```
     * @modifies this
     * @effects constructs a new Route consisting of one segment.
     **/
    public Route(RoadSegment seg) {
        segments = new ArrayList();
    segments.add(seg);

    checkRep();
    }

    private Route(ArrayList l)
    {
    segments = (ArrayList) l.clone();
    }

    // Producers

    /**
     * Constructs a new <code>Route</code> by adding a segment to
     * the end of this <code>Route</code>.
     * @requires seg != null && seg.p1 == this.end
     * @param seg the road segment to be added
     * @return a new <code>Route</code> r such that
     *         r.segments = this.segments + gs
     **/
    public Route addSegment(RoadSegment seg) {
    checkRep();

    Route r = new Route(segments);
    r.segments.add(seg);
    r.checkRep();

    return r;
    }

    /**
     * Constructs a new <code>Route</code> by removing the first segment
     * from this <code>Route</code>.
     *
     * @requires this.steps > 1
     * @return a new <code>Route</code> r such that
     *         this.segments = this.first + r.segments
     */
    public Route popSegment() {
        checkRep();

    Route r = new Route(segments);
    r.segments.remove(0);

    r.checkRep();

    return r;
    }


    // Observers

    /**
     * Returns the location of the start of the route.
     * @return this.start, the location of the start of the route
     */
    public GeoPoint getStart() {
    checkRep();

    return ((RoadSegment) segments.get(0)).getP1();
    }

    /**
```

prefer addAll() to clone()

```
    * Returns the location of the end of this route.
    * @return this.end, the location of the end of the route
    */
   public GeoPoint getEnd() {
   checkRep();

   return ((RoadSegment) segments.get(segments.size()-1)).getP2();
   }

   /**
    * Returns the first segment on this route.
    * @return this.first, the first segment on the route
    */
   public RoadSegment getFirst() {
   checkRep();

   return (RoadSegment) segments.get(0);
   }

   /**
    * Returns the last segment on this route.
    * @return this.last, the last segment on the route
    */
   public RoadSegment getLast() {
   checkRep();

   return (RoadSegment) segments.get(segments.size()-1);
   }

   /**
    * Returns the number of segments on this route.
    * @return this.steps, the number of segments on the route
    */
   public int getSteps() {
   return segments.size();
   }

   /**
    * Returns the total length of this route, in miles.
    * @return this.length, the total length of the route, in miles
    */
   public double getLength() {
   // Theta(n) implementation; could be reduced to Theta(1) by
   // maintaining the length as a separate private field

   checkRep();

   double length = 0;

   for (Iterator iter = segments.iterator(); iter.hasNext();)
   {
       RoadSegment seg = (RoadSegment) iter.next();
       length += seg.getLength();
   }

   return length;
   }

   /**
    * Returns an <code>Iterator</code> of <code>RoadSegments</code>
    *         that yields the segments sequence, in order.
    * @return an <code>Iterator</code> of <code>RoadSegments</code>
    *         that yields the segments sequence, in order.
    */
   public Iterator getSegments() {
   return Collections.unmodifiableList(segments).iterator();
   }

   /**
```

consider caching


```
    * Compares this <code>Route</code> to the specified <code>Object</code>
    * for equality.
    * @return true iff (o instanceof <code>Route</code>) && (o.segments and thi
s.segments
    * contain the same elements in the same order); false otherwise
    */
   public boolean equals(Object o) {
   if (!(o instanceof Route))
       return false;

   Route r = (Route) o;
   return segments.equals(r.segments);
   }

   /**
    * Returns a valid hash code for this.
    * @return a valid hash code for this.
    */
   public int hashCode() {
       return 1;
   }

   /**
    * Returns a string representation of this <code>Route</code>.
    * @return a string representation of this.
    */
   public String toString() {
       throw new RuntimeException ("not implemented");
   }

   /**
    * Checks to see if the representation invariant is being violated
    * and if so, throws a RuntimeException.
    * @throws RuntimeException if representation invariant is violated
    */
   private void checkRep() throws RuntimeException {
   if (segments == null || segments.isEmpty())
       throw new RuntimeException("segments null or empty");

   GeoPoint lastStepEnd = null;

   // Make sure all elements in segments are RoadSegments, and
   // they connect: segment(i).P1 = segment(i-1).P1

   for (ListIterator iter = segments.listIterator(); iter.hasNext(); )
   {
       Object o = iter.next();

       if (!(o instanceof RoadSegment))
       throw new RuntimeException("segments contains non-RoadSegment");
       RoadSegment seg = (RoadSegment) o;
       if (lastStepEnd != null) // skip this check for the first
                   // segment
       {
       if (!(seg.getP1().equals(lastStepEnd)))
           throw new RuntimeException("segments contains segments that do not connect");
       }
       lastStepEnd = seg.getP2();
   }

   }

}
```

```
package ps3;

import junit.framework.*;
import java.util.Iterator;


/**
 * Contains unit test suite for the Route class
 */

public class RouteTest extends TestCase
{
    private static final double TOLERANCE = GeoPointTest.TOLERANCE;

    private GeoPoint gpDowntown;
    private GeoPoint gpWest;
    private GeoPoint gpEast;
    private GeoPoint gpNorth;

    private RoadSegment gsEast;
    private RoadSegment gsWest;
    private RoadSegment gsNorth;
    private RoadSegment gsEast2;
    private RoadSegment gsWest2;
    private RoadSegment gsDiag;
    private RoadSegment gsDiag2;
    private RoadSegment gsZero;

    private Route rDW, rDW2, rDWD, rWDN, rDWN;


    public RouteTest(String name)
    {
    super(name);
    }

    public static Test suite()
    {
    return new TestSuite(RouteTest.class);
    }

    protected void setUp() {
        //                          north       east
        // +1 mile                    14488      19579
        gpDowntown = new GeoPoint(42358333, -71060278);
        gpWest = new GeoPoint(42358333, -71079857);
        gpEast = new GeoPoint(42358333, -71040699);
        gpNorth = new GeoPoint(42372821, -71060278);

        gsEast = new RoadSegment("East", gpDowntown, gpEast);
        gsWest = new RoadSegment("West", gpDowntown, gpWest);
        gsNorth = new RoadSegment("North", gpDowntown, gpNorth);
        gsEast2 = new RoadSegment("East", gpEast, gpDowntown);
        gsWest2 = new RoadSegment("West", gpWest, gpDowntown);
        gsDiag = new RoadSegment("NE", gpWest, gpNorth);
        gsDiag2 = new RoadSegment("NorthEast", gpWest, gpNorth);
        gsZero = new RoadSegment("Zero", gpDowntown, gpDowntown);

    rDW = new Route(gsWest);
    rDW2 = new Route(gsWest);
    rDWD = new Route(gsWest).addSegment(gsWest2);
    rWDN = new Route(gsWest2).addSegment(gsNorth);
    rDWN = new Route(gsWest).addSegment(gsDiag);
    }


    public void testEquals()
    {
    assertEquals("Not equal to self", rDW, rDW);
```



```
    assertTrue("Equal to null", !rDW.equals(null));
    assertTrue("Equal to string", !rDW.equals("test"));
    assertEquals("Not equal to identical object", rDW, rDW2);
    assertTrue("Equal to different object", !rDW.equals(rWDN));
    assertTrue("Equal to different object", !rDW.equals(rDWN));
    assertTrue("Equal to different object", !rWDN.equals(rDWN));
    }


    public void testHashCode()
    {
    assertEquals("Hash code not equal to self",
        rDW.hashCode(), rDW.hashCode());
    assertEquals("Hash code not equal to identical object",
        rDW.hashCode(), rDW2.hashCode());
    }

    public void testGetEnd()
    {
    assertEquals(gpWest, rDW.getEnd());
    assertEquals(gpNorth, rDWN.getEnd());
    }


    public void testGetFirst()
    {
    assertEquals(gsWest, rDW.getFirst());
    assertEquals(gsWest2, rWDN.getFirst());
    assertEquals(gsWest, rDWN.getFirst());
    }


    public void testGetLast()
    {
    assertEquals(gsWest, rDW.getLast());
    assertEquals(gsWest2, rDWD.getLast());
    assertEquals(gsNorth, rWDN.getLast());
    }


    public void testGetLength()
    {
    assertEquals(1.0, rDW.getLength(), TOLERANCE);
    assertEquals(2.0, rDWD.getLength(), TOLERANCE);
    assertEquals(2.414, rDWN.getLength(), TOLERANCE);
    }


    public void testGetSegments()
    {
    Iterator iter = rDWN.getSegments();

    assertTrue(iter.hasNext());
    assertEquals(gsWest, iter.next());
    assertTrue(iter.hasNext());
    assertEquals(gsDiag, iter.next());
    assertFalse(iter.hasNext());
    }


    public void testGetStart()
    {
    assertEquals(gpDowntown, rDW.getStart());
    assertEquals(gpDowntown, rDWD.getStart());
    assertEquals(gpWest, rWDN.getStart());
    }


    public void testGetSteps()
```

test for rep exposure

```
    {
    assertEquals(1, rDW.getSteps());
    assertEquals(2, rDWD.getSteps());
    assertEquals(2, rDWN.getSteps());
    }


    public void testAddSegment()
    {
    Route r = rDW.addSegment(gsWest2);
    assertEquals(r, rDWD);
    r = r.addSegment(gsZero);
    r = r.addSegment(gsEast);
    assertEquals(4, r.getSteps());
    assertEquals(3.0, r.getLength(), TOLERANCE);
    assertEquals(gsEast, r.getLast());
    assertEquals(gsWest, r.getFirst());
    }


    public void testPopSegment()
    {
    Route r = rWDN.popSegment();
    assertEquals(gsNorth, r.getLast());
    assertEquals(gsNorth, r.getFirst());
    assertEquals(1, r.getSteps());
    assertEquals(1.0, r.getLength(), TOLERANCE);
    }

}
```

should test to ensure immutability is preserved

Dan

Overall: 96/100

```
SUMMARY FOR ps3.stafftests.RouteTest
Pass : 13 (100%)
Fail : 0 (0%)
Error: 0 (0%)
Total: 13
SUMMARY FOR ps3.stafftests.CarTest
Pass : 9 (100%)
Fail : 0 (0%)
Error: 0 (0%)
Total: 9
SUMMARY FOR ps3.stafftests.RoadSegmentTest
Pass : 10 (100%)
Fail : 0 (0%)
Error: 0 (0%)
Total: 10
SUMMARY FOR ps3.stafftests.GeoPointTest
Pass : 6 (100%)
Fail : 0 (0%)
Error: 0 (0%)
Total: 6
SUMMARY FOR ALL TESTS
Pass : 38 (100%)
Fail : 0 (0%)
Error: 0 (0%)
Total: 38
GRADES
ps3.stafftests.RouteTest: 11.0/11.0
ps3.stafftests.CarTest: 16.0/16.0
ps3.stafftests.RoadSegmentTest: 5.0/5.0
ps3.stafftests.GeoPointTest: 3.0/3.0
GRADES TOTAL: 35.0/35.0
```

~~34~~ 35/35

44/45

code looks very good, but please

- format your code, there is a lot of funny indentation stuff in your printout
- be a little more descriptive in AF, R1

(you did a pretty good job, this is just a minor point)