1. **hashCode**

   (a) Consider the following `hashCode` method for `SimplePoint2`:

```
@Override
public int hashCode() {
  return (int) (10 * this.x) + (int) (this.y);
}
```

   Compute the hash code for the following `SimplePoint2` objects:

```
SimplePoint2 p1 = new SimplePoint2(0f, 0f);

SimplePoint2 p2 = new SimplePoint2(-5.1f, -2f);

SimplePoint2 p3 = new SimplePoint2(0f, 87.5f);

SimplePoint2 p4 = new SimplePoint2(-6.5f, 60f);

SimplePoint2 p5 = new SimplePoint2(1.9f, 1.5f);
```

   Solution: Remember that casting a double value to an int simply discards everything after the decimal.

```
0 + 0 = 0

-51 - 2 = -53

0 + 87 = 87

-65 + 60 = -5

19 + 1 = 20
```

   (b) Draw a hash table having 10 buckets. Label the buckets 0 through 9.

   Solution:

0	1	2	3	4	5	6	7	8	9

   (c) Suppose that the hash table uses the following method to compute the bucket number:

```
private int bucket(Object obj) {
  int hash = obj.hashCode();
  return (int) (Math.abs(hash) / 10);
  // NOTE: This is not a general purpose algorithm for computing
  // a bucket number.
}
```

Fill in the hash table with the labels `p1`, `p2`, `p3`, `p4`, `p5` indicating which buckets the points from Question 1(a) are placed into.

Solution:

p1 0	1	p5 2	3	4	p2 p4 5	6	7	p3 8	9

2. **Information hiding**

Consider the following implementation of class that represents time on a 24-hour clock:

```
public class HourMin {
    private int hour;         // between 0 and 23
    private int minute;       // between 0 and 59

    public HourMin(String time) {        // assumes strings like "15:35"
        String parts[] = time.split(":");
        int hour = Integer.parseInt(parts[0]);
        int minute = Integer.parseInt(parts[1]);
        this.setHour(hour);
        this.setMinute(minute);
    }

    public final int getHour() {      // returns a value between 0 and 23
        return this.hour;
    }

    public final int getMinute() {    // returns a value between 0 and 59
        return this.minute;
    }

    public final void setHour(int hour) {
        this.hour = hour;       // precondition: hour is between 0 and 23
    }

    public final void setMinute(int minute) {
        this.minute = minute;   // precondition: minute is between 0 and 59
    }

    public String toString() {
        return this.getHour() + ":" + this.getMinute();
    }
}
```

Suppose we change the implementation so that we use only one field `minuteFromMidnight` equal to the number of minutes from midnight. Which of the following features of the class do we have to change when we modify the implementation? (check all that need to change):

-the constructor

-`getHour()`

-`getMinute()`

-`setHour(int)`

-`setMinute(int)`

-`toString()`

Solution: Any constructor or method that uses a field directly will be affected:

-the constructor ✓

`-getHour()` ✓

`-getMinute()` ✓

`-setHour(int)` ✓

`-setMinute(int)` ✓

`-toString()`

3. **Information hiding**

   Re-implement `HourMin` so that it uses a single field `minuteFromMidnight` equal to the number of minutes from midnight.

```
public class HourMin {
    private int minuteFromMidnight;

    public HourMin(String time) {        // assumes strings like "15:35"
        String parts[] = time.split(":");
        int hour = Integer.parseInt(parts[0]);
        int minute = Integer.parseInt(parts[1]);


        this.minuteFromMidnight = 60 * hour + minute;

    }

    public final int getHour() {      // returns a value between 0 and 23

        return this.minuteFromMidnight / 60;

    }

    public final int getMinute() {    // returns a value between 0 and 59

        return this.minuteFromMidnight % 60;

    }

    public final void setHour(int hour) {
        // precondition: hour is between 0 and 23

        int minute = this.getMinute();  // get the current minute
        this.minuteFromMidnight = 60 * hour + minute;

    }

    public final void setMinute(int minute) {
        // precondition: minute is between 0 and 59

        int hour = this.getHour();   // get the current hour
        this.minuteFromMidnight = 60 * hour + minute;

    }

    public String toString() {
        return this.getHour() + ":" + this.getMinute();
    }
}
```

4. **Immutability**

   Immutability has many advantages but it has some disadvantages, too. Consider the following method

in `Point2` (which is mutable) that moves the point by an amount `dx` in the x direction and an amount `dy` in the y direction:

```
public class Point2 {
    private double x;
    private double y;

    // constructors and other methods not shown

    public void move(double dx, double dy) {
        this.x += dx;
        this.y += dy;
    }
}
```

We cannot add an identical method to `IPoint2` because `IPoint2` objects are immutable. The best we can do is to add a method that returns a new `IPoint2` object with the desired coordinates. Implement such a method for `IPoint2`:

```
public class IPoint2 {
    private final double x;
    private final double y;

    // constructors and other methods not shown

    // Returns a new point whose coordinates are equal to
    // this point shifted by dx and dy.
    public IPoint2 moveCopy(double dx, double dy) {

        return new IPoint2(this.x + dx, this.y + dy);

    }
}
```

Why is the method named `moveCopy` instead of simply `move`?

Solution: Because a method named `move` already exists with the same signature.

5. **Class invariants** Examine your answer for Question 3.

   (a) What are the class invariants for `HourMin`? Your answer should involve conditions on the value of `minuteFromMidnight`.

   Solution:

   ```
   this.minuteFromMidnight >= 0
   this.minuteFromMidnight < 60 * 24
   ```

   (b) Which of the following features of the class do we have to change to ensure that the class invariants are always true? (check all that need to change):

   -the constructor

   -`getHour()`

   -`getMinute()`

   -`setHour(int)`

   -`setMinute(int)`

   -`toString()`

   Solution: Any constructor or method that changes `this.minuteFromMidnight` might have to validate its arguments:

   -the constructor ✓

   -`getHour()`

   -`getMinute()`

   -`setHour(int)`✓

   -`setMinute(int)`✓

   -`toString()`

   (c) For each feature you checked in Part (b), what needs to change to to ensure that the class invariants are always true?

   Solution: The constructor must validate the given hour and minute.

   `setHour` must validate the given hour to ensure that `hour >=0 && hour < 24` is true

`setMinute` must validate the given minute to ensure that `minute >=0 && minute < 60` is true

(d) Bonus question: How can you minimize the amount of code duplication when you implement the changes you described in Part (c)?

Solution: Implement a private method named `set`:

```
private void set(int hour, int minute) {
   if (hour < 0 || hour > 23) {
       throw new IllegalArgumentException("bad hour");
   }
   if (minute < 0 || minute > 59) {
       throw new IllegalArgumentException("bad minute");
   }
   this.minuteFromMidnight = 60 * hour + minute;
}
```

The constructor, `setHour`, and `setMinute` can all use `set`.