MIT
6.031: Software Construction
Prof. Rob Miller and Max Goldman

6.031 Spring 2020 Quiz 1

Your Kerberos username: `_____`

You have 50 minutes to complete this quiz. There are 14 problems. The quiz is closed-book and closed-notes, but you are allowed one two-sided page of notes.

This page automatically saves your answers as you work. Saved answers are marked with a green check. If you see a red exclamation mark, or a red notification that you are disconnected, your answers are not being saved: try reloading the page right away, before continuing to work on the quiz.

If you want to ask a clarification question, please put yourself on the online help queue and a staff member will talk to you in a text chat.

Good luck!

Before you begin, please sign this honor statement.

I affirm that this is a closed-book quiz, which means:

- I will not reference any materials aside from my 1-page 2-sided cribsheet (including my class notes, Eclipse, the course website, any files stored on my laptop, and any resources on the internet except this quiz itself and the online help queue).
- I will not communicate with classmates or anyone else (other than 6.031 staff members) about anything related to this quiz until the solutions are officially released.

By entering your full name below (first name and last name), you agree to this honor statement.

The code for this quiz has an ADT `Semester` representing an academic semester. Every year has two semesters, fall and spring. Fall semester 2018 is a different semester than fall semester 2019.

The code is provided at the bottom of this page, and you can open all the code in a separate tab.

---

1. (4 points) In `Semester`, find one example of each kind of operation, or put NONE if the type has no such operation.

Creator:

Observer:

Mutator:

Producer:

---

2. (10 points) Write an abstraction function and rep invariant for `Semester`, consistent with the rep and code given.

Abstraction function:

Rep invariant:

---

3. (10 points) Implement `sameValue()` and `hashCode()`. Your `hashCode()` implementation may not be a constant function.

```
public boolean sameValue(Semester that) {
```

`}`

```
// note: hashCode() must NOT be a constant function
public int hashCode() {
```

`}`

---

4. (6 points) Write a *different* rep for `Semester`, consisting of an `int` and an `enum` type. Make sure to declare your `enum` type here too.

```
public class Semester {
```

`}`

5. (10 points) Reimplement `isFall()` and `next()` for your new rep. Assume that the private constructor `Semester(int)` has been changed to take your new rep as its arguments instead of `int`.

```
public boolean isFall() {

}
```

```
public Semester next() {

}
```

Here is another possible rep for `Semester`:

```
public class Semester {
    Date date;
    // AF: TBD
    // RI: date.getMonth() is not 0, 5, 6, or 7
   ...
}
```

You can refer to the API documentation for `Date`, which is included in the code at the bottom of this quiz (open all the code in a separate tab).

Problems 6-10 all refer to this new rep.

6. (5 points) Write an abstraction function for this new rep, consistent with the rep invariant shown.

7. (5 points) Write `checkRep()` for this new rep.

```
private void checkRep() {

}
```

8. (10 points) Reimplement `isFall()` and `next()` for this new rep. Assume that the private constructor `Semester(int)` has been changed to `Semester(Date)` instead.

```
public boolean isFall() {
```

```
}
```

```
public Semester next() {
```

```
}
```

---

9. (10 points) Implement `sameValue()` and `hashCode()` for this new rep. Your `hashCode()` implementation must not be a constant function.

```
public boolean sameValue(Semester that) {
```

```
}
```

```
// note: hashCode() must NOT be a constant function
public int hashCode() {
```

```
}
```

---

10. (5 points) Suppose the `Semester(Date)` constructor looks like this:

```
public Semester(Date date) {
    this.date = date;
}
```

In one sentence, state why this code is not safe from bugs:

Consider this new operation for `Semester` that returns the length of an interval spanned by two semesters:

```
/**
  * @param that  a semester
  * @return number of semesters in the interval
  *         spanned by this and that, inclusive
  */
public int span(Semester that)
```

For example, if `fall2018` and `spring2020` are `Semester` values corresponding to

their names, then `fall2018.span(spring2020)` is `4`.

11. (5 points) What is the precondition of the spec above?

12. (10 points) For each of the alternative specs below, state whether it is STRONGER than, WEAKER than, or INCOMPARABLE to the spec above, and **give an example** (using self-describing names like `fall2018` and `spring2020`) to explain your answer.

```
/** @param that  a semester, must be earlier in time than this
  * @return number of semesters in the interval
  *         spanned by this and that, inclusive
  */
public int span(Semester that)
```

```
/** @param that  a semester
  * @return 2 * (the year of that - the year of this)
  */
public int span(Semester that)
```

13. (5 points) Write a useful, three-subdomain partition for the original spec of `span()`.

14. (5 points) Give three test cases that cover your partition. Do not write JUnit code, just write the inputs and expected output for each test case.

`Semester`

```
/**
 * Semester represents an academic semester.
 * Every year has 2 semesters, fall and spring.
 * Fall semester 2018 is a different semester than fall semester 2019.
 */
public class Semester {
    private int num;

    // AF: TBD
    // RI: TBD
    // Safety from rep exposure: TBD

    /** Make a semester from a season and year, in string form.
     *  @param s  string of the form ([Ff]all|[Ss]pring)\d+ */
    public Semester(String s) {
        // TBD
```

```
    }

    private Semester(int semesterNumber) {
        this.num = semesterNumber;
    }

    /** @return true if and only if this is a fall semester */
    public boolean isFall() {
        return num % 2 == 0;
    }

    /** @return the next semester after this */
    public Semester next() {
        return new Semester(num+1);
    }

    //
    // Semester may have other operations, not shown
    //

    @Override public boolean equals(Object that) {
        return that instanceof Semester && sameValue((Semester)that);
    }

    /** @return true iff this and that represent the same semester */
    private boolean sameValue(Semester that) {
        // TBD
    }

    @Override public int hashCode() {
        // TBD
    }
}
```

Date

```
/**
 * The class Date represents a specific instant in time.
 */
public class Date {

    /**
     * Allocates a Date object and initializes it to the current time.
     */
```

```
    public Date() { ... }

    /**
     * Allocates a Date object and initializes it so that
     * it represents midnight, local time, at the beginning of the day
     * specified by the `year`, `month`, and
     * `date` arguments.
     *
     * @param year  the year minus 1900.
     * @param month the month between 0-11.
     * @param date  the day of the month between 1-31.
     */
    public Date(int year, int month, int date) { ... }

    /**
     * Allocates a Date object and initializes it so that
     * it represents the instant at the start of the minute specified by
     * the `year`, `month`, `date`,
     * `hrs`, and `min` arguments, in the local
     * time zone.
     *
     * @param year  the year minus 1900.
     * @param month the month between 0-11.
     * @param date  the day of the month between 1-31.
     * @param hrs   the hours between 0-23.
     * @param min   the minutes between 0-59.
     */
    public Date(int year, int month, int date, int hrs, int min) { ... }

    /**
     * Returns a value that is the result of subtracting 1900 from the
     * year that contains or begins with the instant in time represented
     * by this Date object, as interpreted in the local
     * time zone.
     *
     * @return the year represented by this date, minus 1900.
     */
    public int getYear() { ... }

    /**
     * Sets the year of this Date object to be the specified
     * value plus 1900. This Date object is modified so
     * that it represents a point in time within the specified year,
     * with the month, date, hour, minute, and second the same as
     * before, as interpreted in the local time zone. (Of course, if
```

```
     * the date was February 29, for example, and the year is set to a
     * non-leap year, then the new date will be treated as if it were
     * on March 1.)
     *
     * @param year the year value.
     */
    public void setYear(int year) { ... }

    /**
     * Returns a number representing the month that contains or begins
     * with the instant in time represented by this Date object.
     * The value returned is between `0` and `11`,
     * with the value `0` representing January.
     *
     * @return the month represented by this date.
     */
    public int getMonth() { ... }

    /**
     * Sets the month of this date to the specified value. This
     * Date object is modified so that it represents a point
     * in time within the specified month, with the year, date, hour,
     * minute, and second the same as before, as interpreted in the
     * local time zone. If the date was October 31, for example, and
     * the month is set to June, then the new date will be treated as
     * if it were on July 1, because June has only 30 days.
     *
     * @param month the month value between 0-11.
     */
    public void setMonth(int month) { ... }

    /**
     * Returns the day of the month represented by this Date object.
     * The value returned is between 1 and 31
     * representing the day of the month that contains or begins with the
     * instant in time represented by this Date object, as
     * interpreted in the local time zone.
     *
     * @return the day of the month represented by this date.
     */
    public int getDate() { ... }

    /**
     * Sets the day of the month of this Date object to the
     * specified value. This Date object is modified so that
```

```
     * it represents a point in time within the specified day of the
     * month, with the year, month, hour, minute, and second the same
     * as before, as interpreted in the local time zone. If the date
     * was April 30, for example, and the date is set to 31, then it
     * will be treated as if it were on May 1, because April has only
     * 30 days.
     *
     * @param date the day of the month value between 1-31.
     */
    public void setDate(int date) { ... }

    /**
     * Returns the day of the week represented by this date. The
     * returned value (0 = Sunday, 1 = Monday,
     * 2 = Tuesday, 3 = Wednesday, 4 =
     * Thursday, 5 = Friday, 6 = Saturday)
     * represents the day of the week that contains or begins with
     * the instant in time represented by this Date object,
     * as interpreted in the local time zone.
     *
     * @return the day of the week represented by this date.
     */
    public int getDay() { ... }

    /**
     * Returns the hour represented by this Date object. The
     * returned value is a number (0 through 23)
     * representing the hour within the day that contains or begins
     * with the instant in time represented by this Date
     * object, as interpreted in the local time zone.
     *
     * @return the hour represented by this date.
     */
    public int getHours() { ... }

    /**
     * Sets the hour of this Date object to the specified value.
     * This Date object is modified so that it represents a point
     * in time within the specified hour of the day, with the year, month,
     * date, minute, and second the same as before, as interpreted in the
     * local time zone.
     *
     * @param hours the hour value.
     */
    public void setHours(int hours) { ... }
```

```
    /**
     * Returns the number of minutes past the hour represented by this date
     * as interpreted in the local time zone.
     * The value returned is between 0 and 59.
     *
     * @return the number of minutes past the hour represented by this date
     */
    public int getMinutes() { ... }

    /**
     * Sets the minutes of this Date object to the specified value.
     * This Date object is modified so that it represents a point
     * in time within the specified minute of the hour, with the year, mont
     * date, hour, and second the same as before, as interpreted in the
     * local time zone.
     *
     * @param minutes the value of the minutes.
     */
    public void setMinutes(int minutes) { ... }

    /**
     * Returns the number of seconds past the minute represented by this da
     * The value returned is between 0 and 61. The
     * values 60 and 61 can only occur on those
     * Java Virtual Machines that take leap seconds into account.
     *
     * @return the number of seconds past the minute represented by this da
     */
    public int getSeconds() { ... }

    /**
     * Sets the seconds of this Date to the specified value.
     * This Date object is modified so that it represents a
     * point in time within the specified second of the minute, with
     * the year, month, date, hour, and minute the same as before, as
     * interpreted in the local time zone.
     *
     * @param seconds the seconds value.
     */
    public void setSeconds(int seconds) { ... }

}
```