design and code a class TimeOfDay providing a time of day and associated operations

abstract data type: data + operations

development steps following the Design by Contract pattern

1. Informal requirement analysis;
2. Design class interface;
3. Add contracts;
4. Design internal representation;
5. Implement methods.
Creation and initialization to 00:00:00;

Queries/Inspection
- current time in integers representing hours, minutes and seconds;
- current time in seconds since 00:00:00;
- current time in string of the form hh:mm:ss;

Commands/Modification
- reset to 00:00:00;
- set to given hours, minutes, seconds;
- set to given seconds since 00:00:00;
- advance time by a number of seconds ("modulo length of day");
- tick, i.e. advance time by one second ("modulo length of day");
operations grouped according to their main role

▶ construction and initialization

```java
public class TimeOfDay {
    // Constructors
    public TimeOfDay() {...}
}
```

▶ queries ("observable entities"), i.e. inspection of the current state

```java
public int hours() {...}
public int minutes() {...}
public int seconds() {...}
public int totalSeconds() {...}
public String toString() {...}
```

▶ commands, i.e. changes of the current state

```java
public void setHMS(int h, int m, int s) {...}
public void setTotalSeconds(int ts) {...}
public void advance(int s) {...}
public void reset() {...}
public void tick() {...}
```
hours, minutes, seconds, and totalSeconds

- primitive queries
- restriction on admissible values expressed in *invariants*
  - I0: $0 \leq hours \&\& hours < 24$
  - I1: $0 \leq minutes \&\& minutes < 60$
  - I2: $0 \leq seconds \&\& seconds < 60$
  - I3: $0 \leq totalSeconds \&\& totalSeconds < \text{SECONDSINDAY}$
- values related expressed by invariant
  - I4: $totalSeconds == hours*60*60 + minutes*60 + seconds$

auxiliary validator function to express condition I0

```java
// ret: 0 <= h && h < 24
public boolean validHours(int h){...}
```

// I0: validHours(hours)

likewise for I1 and I2
constructor TimeOfDay(): create a TimeOfDay object and initialize it to 00:00:00

    // pre: true
    // post: totalSeconds == 0
    public TimeOfDay() {...}

command setHMS(int h, int m, int s) should not violate I0 through I2

    // pre: validHours(h), validMinutes(m), validSeconds(s)
    // post: hours == h, minutes == m, seconds == s
    public void setHMS(int h, int m, int s){...}
command advance(int s) should advance the current time by s seconds, modulo the length of the day (in seconds)

// pre: 0 <= s
// post: totalSeconds' == (totalSeconds + s) % SECONDSINDAY
public void advance(int s){...}

where
'totalSeconds: the value of totalSeconds before the operation
totalSeconds': the value of totalSeconds after the operation

alternative specification with specification constant C

// pre: 0 <= s && totalSeconds == C
// post: totalSeconds == (C + s) % SECONDSINDAY
public void advance(int s){...}
// Commands

// pre: validHours(h), validMinutes(m), validSeconds(s)
// post: hours == h, minutes == m, seconds == s
public void setHMS(int h, int m, int s){...}

// pre: 0 <= ts < SECONDSINDAY
// post: totalSeconds == ts
public void setTotalSeconds(int ts){...}

// pre: 0 <= s
// post: totalSeconds' == (totalSeconds + s) % SECONDSINDAY
public void advance(int s){...}

// pre: true
// post: totalSeconds == 0
public void reset(){...}

// pre: true
// post: totalSeconds' == (totalSeconds + 1) % SECONDSINDAY
public void tick(){...}
given the contracts some operations could already be implemented by using other operations (even before an internal representation is chosen)

```java
// pre: true
// post: totalSeconds == 0
public void reset()
{
    setTotalSeconds(0);
}

// pre: true
// post: totalSeconds' == (totalSeconds + 1) % SECONDSINDAY
public void tick()
{
    advance(1);
}

or the other way round (not very efficient)
```

```java
public void advance(int s)
{
    for (int i = 0; i != s; i++)
    {
        tick();
    }
}
```
two obvious alternatives for an internal representation

1. three private instance variables representing hours, minutes and seconds:

   ```java
   private int fSeconds;
   private int fMinutes;
   private int fHours;
   ```

   // private invariants
   // Pri0: validHours(fHours)
   // Pri1: validMinutes(fMinutes)
   // Pri2: validSeconds(fSeconds)

2. one private field representing the total seconds:

   ```java
   private int fTotalSeconds;
   ```

   // private invariants
   // Pri0: 0 <= fTotalSeconds && fTotalSeconds < SECONDSINDAY
choice between alternatives to be made on basis of

- ease of coding operations
- efficiency of operations (computation time)
- memory used by internal representation

operation advance(s): far more easily coded using alternative 2
other operations: no real difference between the alternatives

alternative 2 is chosen for the internal representation
public class TimeOfDay {

    // Constants
    public static final int SECONDSINDAY = 24 * 60 * 60;

    // Private part
    private int fTotalSeconds;

    // Private invariants
    // Pri0: 0 <= fTotalSeconds && fTotalSeconds < SECONDSINDAY

    // Constructors
    public TimeOfDay() {
        reset();
    }
// Queries

public int hours() {
    return fTotalSeconds / 3600;
}

public int minutes() {
    return (fTotalSeconds % 3600) / 60;
}

public int seconds() {
    return fTotalSeconds % 60;
}

public int totalSeconds() {
    return fTotalSeconds;
}

// ret: (hours, minutes, seconds) in string
// of the form  hh:mm:ss
public String toString() {
    return String.format("%02d:%02d:%02d", hours(), minutes(), seconds());
}
// Commands

// pre: validHours(h), validMinutes(m), validSeconds(s)
// post: hours == h, minutes == m, seconds == s
public void setHMS(int h, int m, int s) {
    // check pre
    assert validHours(h) && validMinutes(m) && validSeconds(s):
    String.format("setHMS.pre.failed;\(h,m,s\)\=\(\%02d:%02d:%02d\", h, m, s);

    fTotalSeconds = h * 3600 + m * 60 + s;
}

// pre: 0 <= ts < SECONDSINDAY
// post: totalSeconds == ts
public void setTotalSeconds(int ts) {
    // check pre
    assert 0 <= ts && ts < SECONDSINDAY:
    String.format("setTotalSeconds.pre.failed;\(ts\)\=\(%d\", ts);

    fTotalSeconds = ts;
}
// pre: 0 <= s
// post: totalSeconds' == (totalSeconds + s) % SECONDSINDAY
public void advance(int s){
    // check pre
    assertEquals(0, s);

    fTotalSeconds = (fTotalSeconds + s) % SECONDSINDAY;
}

// pre: true
// post: totalSeconds == 0
public void reset(){
    setTotalSeconds(0);
}

// pre: true
// post: totalSeconds' == (totalSeconds + 1) % SECONDSINDAY
public void tick(){
    advance(1);
}
- definition of class TimeOfDay in file TimeOfDay.java
- a class TimeOfDayTester defined in file TimeOfDayTester.java residing in the same directory can use the class TimeOfDay

```java
public class TimeOfDayTester {
    public static void main(String[] args) {
        TimeOfDay t; // variable for holding a reference
        // to a TimeOfDay object
        t = new TimeOfDay(); // new TimeOfDay object created;
        // t contains reference to it
        // t represents 00:00:00
        // calls of instance methods of object referenced by t
        t.tick(); // t represents 00:00:01
        t.advance(3659); // t represents 01:01:00

        System.out.println(t.toString()); // prints "01:01:00"
        t.setHMS(23, 59, 30); // t represents 23:59:30
        t.advance(3600); // t represents 00:59:30
    }
}
```
consider the following function added to class TimeOfDay

```java
// pre: true
// return: whether the time represented by this is earlier on the day than the time represented by t
public boolean isEarlierThan(TimeOfDay t) {
    return this.totalSeconds() < t.totalSeconds();
}
```

it can be used as follows

```java
TimeOfDay t1 = new TimeOfDay();
TimeOfDay t2 = new TimeOfDay();

... // code containing several instance method calls
... // on the objects referenced by t1 and t2

if (t1.isEarlierThan(t2)) {
    ...
} else {
    ...
}
```