Project Title: Implementing a Date Abstract Data Type (DateADT)

Objective: The goal of this project is to implement a Date Abstract Data Type (DateADT) without using any built-in date/time libraries. Students must design and implement a class that represents a specific instant in time, supporting fundamental date and time operations, while manually handling aspects like leap years, valid ranges, and conversions to milliseconds.

Overview

In many real-world applications, developers rely on date/time libraries to manage date operations, leap years, and conversions. This project aims to deepen understanding by creating a custom DateADT from scratch. You will implement core methods for constructing, getting, setting, comparing, and stringifying date/time values, using only basic arithmetic.

Key Features

- **No built-in date/time libraries:** All computations must be done manually.
- **Millisecond epoch handling:** Provide `getTime()` and `setTime(long ms)` to convert between date/time fields and an epoch-based millisecond representation.
- **Basic date/time fields:** year, month, day, hours, minutes, seconds.
- **Validation and leap years:** Validate inputs (e.g., month in [0..11], day in [1..maxDays]) and properly handle leap-year rules.
- **Comparison methods:** `before()` and `after()` to compare two DateADT instances.
- **String parsing/formatting:** Support constructing from (and returning) a string like `YYYY-MM-DD HH:MM:SS`.

Requirements

1. **Constructor Support**
   - `DateADT(int year, int month, int day)`
   - `DateADT(int year, int month, int day, int hrs, int min, int sec)`
   - `DateADT(String dateStr)` (e.g., `"2024-02-29 15:45:30"`)
2. **Getter/Setter Methods**
   - `getYear()` , `getMonth()` , `getDay()` , `getHours()` , `getMinutes()` , `getSeconds()`
   - `setYear()` , `setMonth()` , `setDay()` , `setHours()` , `setMinutes()` , `setSeconds()`

   - `getTime()` , `setTime(long ms)`

3. **Comparison Methods**
   - `boolean before(DateADT other)`
   - `boolean after(DateADT other)`
4. **String Representation**
   - `String toString()` returns `"YYYY-MM-DD HH:MM:SS"` format.

Implementation Guidelines

1. **No external libraries:** Instead, use manual arithmetic for leap-year checks (e.g., `(year % 4 == 0 && year % 100 != 0) || year % 400 == 0`) and day-of-month calculations.
2. **Epoch logic:** Internally define an epoch such as 1970-01-01 00:00:00. `getTime()` calculates total milliseconds from this epoch; `setTime(ms)` does the reverse.
3. **Month and day ranges:**
   - Month is 0-based (`0` = January, `11` = December).
   - Day ranges from `1` to `28/29/30/31` depending on month and leap year.
4. **Seconds range:** Typically `0–59` or up to `60` to account for leap seconds.
5. **Validation:** Throw an error (or raise an exception) for invalid inputs.
6. **Optional expansions:**
   - Compute day differences between two dates.
   - Add or subtract days, months, or years.
   - Timezone adjustments.

Sample Implementations

Below are reference implementations in **Python** and **Java** that satisfy the above requirements. **Note**: Each includes constructor overloads, manual date arithmetic, `before()` / `after()` , and string parsing.

Python Reference (`date_adt.py`)

```
class DateADT:
    def __init__(self, *args):
        """
        Constructor overloads:
        1) DateADT(year, month, day)
        2) DateADT(year, month, day, hours, minutes, seconds)
        3) DateADT(date_string)
        """
        # ... Implementation details (see code for full version)

    def before(self, other):
        return self.getTime() < other.getTime()

    def after(self, other):
        return self.getTime() > other.getTime()

    def getTime(self):
        # Convert this date/time to ms since epoch.
        pass  # Detailed logic in reference solution

    def setTime(self, ms):
        # Convert ms since epoch to date/time fields.
        pass

    def toString(self):
        return f"{self.year:04d}-{self.month:02d}-{self.day:02d} " \
               f"{self.hours:02d}:{self.minutes:02d}:{self.seconds:02d}"

# Example usage:
if __name__ == "__main__":
    date1 = DateADT(2023, 2, 28)
    date2 = DateADT("2024-01-15 13:45:50")
    print(date1.toString(), date2.toString())
    print("Is date1 before date2?", date1.before(date2))
```

Java Reference (`DateADT.java`)

```
public class DateADT {
    private int year, month, day, hours, minutes, seconds;

    public DateADT(int year, int month, int day) {
        // ...
    }

    public DateADT(int year, int month, int day, int hours, int minutes, int seconds
        // ...
    }

    public DateADT(String dateStr) {
        // Parse "YYYY-MM-DD HH:MM:SS"
        // ...
    }

    public boolean before(DateADT other) {
        return this.getTime() < other.getTime();
    }
    public boolean after(DateADT other) {
        return this.getTime() > other.getTime();
    }

    public long getTime() {
        // Convert to ms from epoch (1970-01-01).
    }

    public void setTime(long ms) {
        // Convert from ms to date/time fields.
    }

    @Override
    public String toString() {
        return String.format("%04d-%02d-%02d %02d:%02d:%02d", year, month, day, hour
    }

    public static void main(String[] args) {
        DateADT date1 = new DateADT(2023, 2, 28);
        DateADT date2 = new DateADT("2024-01-15 13:45:50");
        System.out.println("Date1: " + date1);
        System.out.println("Date2: " + date2);
        System.out.println("Is date1 before date2? " + date1.before(date2));
    }
}
```

Testing & Validation

1. **Unit Tests:**
    - Verify constructors for various valid/invalid dates.
    - Check leap-year calculations (e.g., 2024, 1900, 2000).
    - Test `before()` and `after()` between known dates.
    - Confirm `getTime()` returns consistent results.
    - Confirm `setTime()` properly reconstructs date/time.
2. **Edge Cases:**
    - Dates near boundaries: Dec 31, Feb 29 in leap year, invalid Feb 29.
    - Negative or extremely large year values, if desired.
    - Handling of seconds = 60 for leap seconds.

Submission Requirements

1. **Source Code:** Provide `date_adt.py` or `DateADT.java` .
2. **Readme/Documentation:** Summarize your approach, known limitations, and special cases.
3. **Testing Evidence:** Document or include test outputs demonstrating successful checks.

Evaluation Criteria

- **Correctness:** All core requirements and date/time calculations are accurate.
- **Robustness:** Proper input validation and error handling.
- **Code Quality:** Readable, well-structured, and follows best practices.
- **Performance:** Reasonable efficiency for typical date/time operations.

Possible Extensions

- **differenceInDays(DateADT other):** Return the integer day difference.
- **Time Zone Support:** Storing offsets or performing conversions.
- **Add/Subtract Methods:** e.g., `addDays(int n)` , `addMonths(int n)` , etc.
- **String Parsing Enhancements:** Handling multiple formats or localized strings.