CSC1016S Assignment 4: Encapsulation, UML Modelling and Classes

Assignment Instructions

This assignment has two parts. Part I deals with class declarations with encapsulation. Part II deals with UML modelling and classes. Furthermore, each part has a separate scenario.

Part I: Encapsulation

The first part of the assignment is a continuation of Assignment 3. It involves building, testing and debugging Java programs that create and manipulate composition of classes that model simple types of object; and that employ the full gamut of implementing technologies: fields, constructors, methods and access modifiers.

The Scenario

The scenario remains the same as for Assignment 3: the simulation of a pay-to-stay car paid. In this part of the assignment we bring it to a conclusion by adding in the missing element: payment.

To recap, the kind of car park in question has a ticket machine at the entrance and a cashier at the exit. A driver, on entering the car park receives a ticket stamped with the arrival time. (The arrival time is also recorded on the magnetic strip on the back.) On exit, the driver gives the ticket to the cashier, the duration of the stay is calculated and from that, how much must be paid.

For this part of the assignment, the focus is on developing a class of "tariff table" that can be used to determine the cost of a stay.

Here is sample I/O from the final program:

```
Car Park Simulator
The current time is 00:00:00.
Commands: tariffs, advance {minutes}, arrive, depart, quit.
>tariffs
[0 minutes .. 30 minutes] : R10.00
[30 minutes .. 1 hour] : R15.00
[1 hour .. 3 hours] : R20.00
[3 hours .. 4 hours] : R30.00
[4 hours .. 5 hours] : R40.00
[5 hours .. 6 hours] : R50.00
[6 hours .. 8 hours] : R60.00
[8 hours .. 10 hours] : R70.00
[10 hours .. 12 hours] : R90.00
[12 hours .. 1 day] : R100.00
>arrive
Ticket issued: Ticket[id=80000001, time=00:00:00].
>advance 20
The current time is 00:20:00.
>arrive
Ticket issued: Ticket[id=80000002, time=00:20:00].
>advance 15
The current time is 00:35:00.
>depart 80000001
Ticket details: Ticket[id=80000001, time=00:00:00].
Current time: 00:35:00.
```

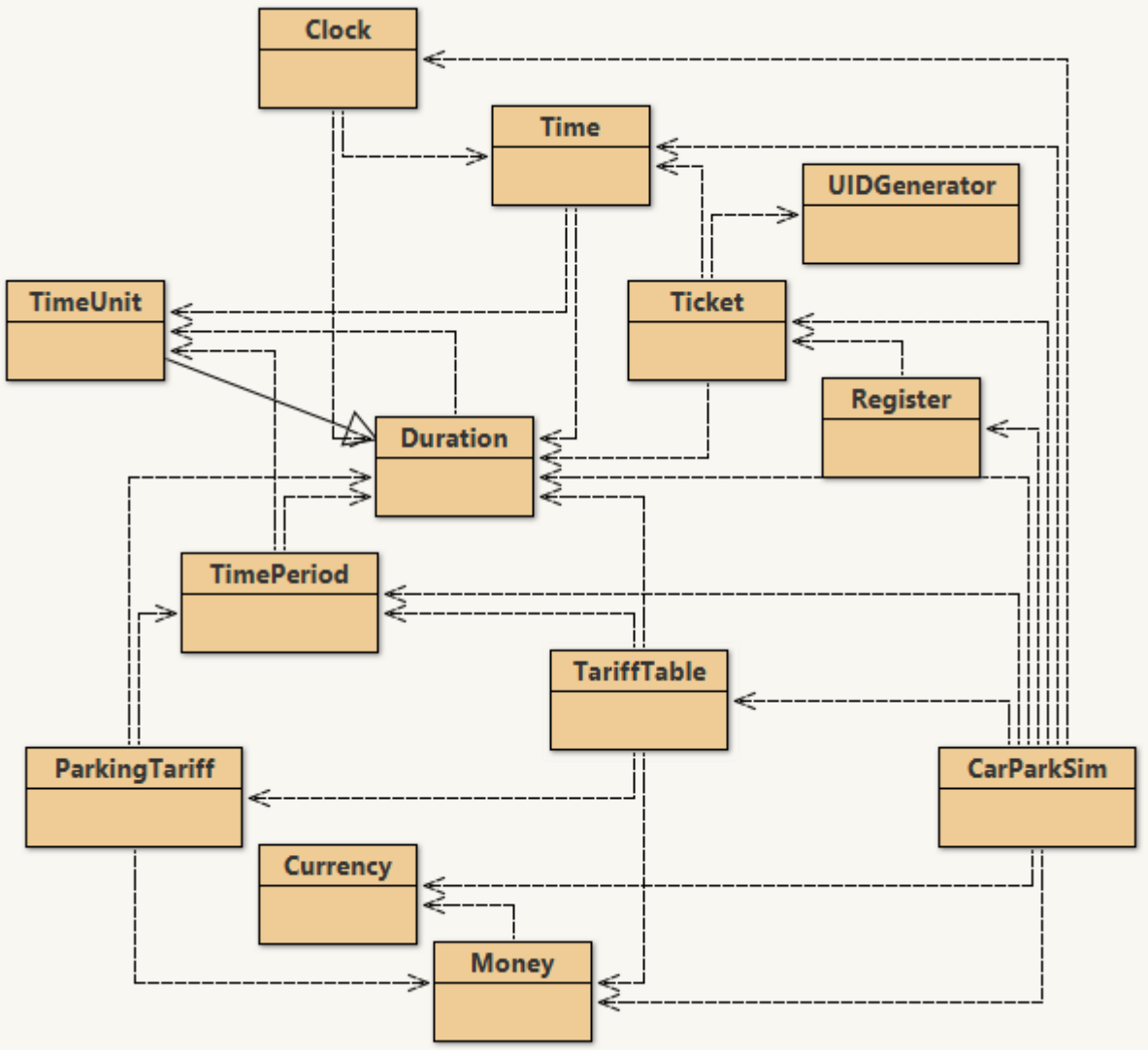
```
Duration of stay: 35 minutes.
Cost of stay : R15.00.
>advance 6
The current time is 00:41:00.
>depart 80000002
Ticket details: Ticket[id=80000002, time=00:20:00].
Current time: 00:41:00.
Duration of stay: 21 minutes.
Cost of stay : R10.00.
>quit
Goodbye.
```

Items in bold represent user input.

As you can see, where it differs from the previous version is that:

- There's a 'tariff' command that prints a list of parking tariffs.
- When a vehicle departs, the cost of stay is printed after the duration.

The following diagram depicts the classes that form the program:



We have tweaked the `Duration` and `Time` classes, and created a new `TimeUnit` class. You should use these versions for this assignment, found on the Vula page. (The big change is the addition of some string formatting for Duration.)

The new components are a `TimePeriod` class (supplied), a `TariffTable` class, and a `ParkingTariff` class. `Money` and `Currency` (from assignment 1) make a reappearance. The `CarParkSim` class is modified to accommodate the need to, given the duration of a stay, look up the tariff.

A TimePeriod is a duration range. It has an inclusive lower bound, l, and an exclusive upper bound, u. A time duration, d, falls within the range if $l \leq d < u$.

For example, a TimePeriod might be 30 up to (but not including) 90 minutes, where 30 is the inclusive lower bound, and 90 is the exclusive upper bound. A duration of 90 minutes does not fall within the period, a duration of 89 minutes does.

A ParkingTariff represents the parking tariff, t, for stays that fall within a given time period, p.

For example, if you park your car in the Cavendish Square shopping mall (in Claremont, Cape Town), stays from 2 hours up to (but not including) 3 hours cost R15.

A TariffTable represents a series of parking tariffs in ascending order of time period.

For example, (taken from Cavendish Square again)

Time Period	*Tariff*
0 – 2 Hours	R13
2 – 3 Hours	R15
3 – 4 Hours	R20
4 – 5 Hours	R30
5 – 6 Hours	R50
6 – 7 Hours	R70

The `TimePeriod` class has been provided for you. It has the following specification:

Class TimePeriod

A TimePeriod is a Duration range. It has an inclusive lower bound, l, and an exclusive upper bound, u.
A Duration, d, falls within the range if l<=d<u.

Constructors

public TimePeriod(Duration lowerBound, Duration upperBound)
// Create a TimePeriod with the given inclusive lower bound and exclusive upper bound.

Methods

public Duration lowerBound()
// Obtain the lower bound for this time period.
public Duration upperBound()
// Obtain the upper bound for this time period.
public boolean includes(Duration duration)
// Determine whether the given duration falls within this time period i.e. whether
// lowerBound()≤duration<upperBound().
public boolean precedes(TimePeriod other)
// Determine whether this time period precedes the other time period i.e. whether
// this.upperBound()≤other.lowerBound().

```
public boolean adjacent(TimePeriod other)
        // Determine whether this time period is adjacent to the other time period i.e. whether
        // this.upperBound() is equal to other.lowerBound(), or this.lowerBound() is equal to other.upperBound()
public String toString()
        // Obtain a String representation of this TimePeriod object in the form:
        // "[<duration> .. <duration>]" where durations are given as a series of non-zero quantities of time // units, the
        smallest being minutes.
```

Your task is to construct the `TariffTable` class (exercise 2) which in turn requires the design and construction of a `ParkingTariff` class (exercise 1). In exercise 3, we revise and construct the `CarParkSim` class.

For this assignment we expect to see the use of access modifiers to properly implement classes. In this regard, class specifications are properly abstract – they describe a type of object's attributes and behaviour but not how it's implemented i.e. there are no details of instance variables and how they are manipulated.

The assignment will be part automatically marked and part manually marked. The tutors will review your implementation decisions; your choice of instance variables and data types.

Exercise 1 [10 marks]

Your first task is to develop a `ParkingTariff` class. The primary objective is actually to develop a TariffTable class, however, there's a design issue that must be solved first.

A TariffTable must store a series of parking tariffs. Each is an association between a time period and a cost. You could implement a TariffTable by using two arrays, one containing TimePeriod objects and the other containing Money objects, where the TimePeriod at index *i* in the first array is associated with the Money object at index *i* in the second array.

That's not a particularly nice solution since there's nothing in the resulting variable declarations to indicate the relationship, and there's potential for mismatch. A better solution is to have a TariffTable contain a collection of ParkingTariff objects, where each ParkingTariff object stores a TimePeriod and Money.

Develop a ParkingTariff class of object.

- Your class will have instance variable(s), constructor(s), and method(s).
- You should aim to move beyond simple get and set methods to ones that offer greater functionality.

With respect to the second point, you may wish to develop the ParkingTariff class in conjunction with the TariffTable class (exercise 2):

- Think about what a TariffTable object must do.
- Think about what could be delegated to ParkingTariff objects

Your solution will be manually marked by the tutors.

Exercise 2 [30 marks]

Develop a TariffTable class that meets the following specification:

Class TariffTable

A TariffTable records parking tariffs for a pay-to-stay car park.

Constructors

public TariffTable(int maxSize)
// Create a TariffTable with the given maximum number of entries.

Methods

public void addTariff(TimePeriod period, Money tariff)
// Add the tariff for the given period to the table. The period must directly follow, and be
// adjacent to, that for the previous tariff entered.
// If the period does not follow or is not adjacent then an IllegalArgumentException is thrown.
public Money getTariff(Duration lengthOfStay)
// Obtain the tariff for the given length of stay.
public String toString()
// Obtain a String representation of this TariffTable in the form:
// <period$_0$> : <tariff$_0$>
// …
// <period$_n$> : <tariff$_n$>

Here's a snippet of code to illustrate behaviour:

```
//...
final Currency currency = new Currency("R", "ZAR", 100);
final TimePeriod pOne = new TimePeriod(new Duration("hour", 1), new
Duration("hour", 2));
final TimePeriod pTwo = new TimePeriod(new Duration("hour", 2), new
Duration("hour", 3));

final TariffTable tariffTable = new TariffTable(2);
tariffTable.addTariff(pOne, new Money("R2", currency));
tariffTable.addTariff(pTwo, new Money("R5", currency));

System.out.println(tariffTable);

System.out.println(tariffTable.getTariff(new Duration("minute",
65)));
System.out.println(tariffTable.getTariff(new Duration("hour", 2)));
//...
```

The output from the fragment would be:

```
[60..120 minutes] : R2.00
[120..180 minutes] : R5.00
R2.00
R5.00
```

NOTE:

- The `toString()` method must return a string containing a series of lines, one for each parking tariff. All but the last line must end with a newline character '\n'.

- The addTariff() method must check that the period, p_n, for the new tariff follows on from the previous one, p_p entered i.e. that p_0 precedes and is adjacent to p_n. If this condition is not met then an exception should be thrown. Here' s the expression your code should use:

```
throw new IllegalArgumentExeption("TimePeriod:addTariff():
precondition not met.");
```

Evaluate your work: by (i) writing a test program, or (ii) using the jGrasp interactive feature (as demonstrated in assignment 3).

Exercise 3 [20 marks]

The CarParkSim class contains the main program method. It creates the Register, Clock and TariffTable objects, and handles user input/output.

Extending the CarParSim solution that you constructed for the previous assignment, you need to:

- Add code to create and populate a TariffTable object.
- Extend the code for the 'depart' command to include printing the tariff that applies to a stay of that duration.
- Add code for an additional 'tariffs' command: When the user enters the 'tariffs' command, a list of parking tariffs will be printed.

Here is some more sample I/O:

```
Car Park Simulator
The current time is 00:00:00.
Commands: tariffs, advance {minutes}, arrive, depart, quit.
>advance 10
The current time is 00:10:00.
>arrive
Ticket issued: Ticket[id=80000001, time=00:10:00].
>advance 1
The current time is 00:11:00.
>arrive
Ticket issued: Ticket[id=80000002, time=00:11:00].
>arrive
Ticket issued: Ticket[id=80000003, time=00:11:00].
>depart 80000003
Ticket details: Ticket[id=80000003, time=00:11:00].
Current time: 00:11:00.
Duration of stay: 0 minutes.
Cost of stay : R10.00.
>depart 80000006
Invalid ticket ID.
>depart 80000001
Ticket details: Ticket[id=80000001, time=00:10:00].
Current time: 00:11:00.
Duration of stay: 1 minute.
Cost of stay : R10.00.
>tariffs
[0 minutes .. 30 minutes] : R10.00
[30 minutes .. 1 hour] : R15.00
```

```
[1 hour .. 3 hours] : R20.00
[3 hours .. 4 hours] : R30.00
[4 hours .. 5 hours] : R40.00
[5 hours .. 6 hours] : R50.00
[6 hours .. 8 hours] : R60.00
[8 hours .. 10 hours] : R70.00
[10 hours .. 12 hours] : R90.00
[12 hours .. 1 day] : R100.00
>quit
Goodbye.
```

Part II: UML Modelling and Classes

The second part of the assignment concerns the development of UML models, and the construction of classes in Java using object composition i.e. write class declarations that define types of object that contain and manipulate other objects.

This part of the assignment is based on the following scenario:

Scenario

The Fynbos Flower company is flexible on the hours that employees work, and as a consequence, timekeeping is an important administrative practice. The company records the dates and times of shifts (for want of a better word). Employees sign-in when they arrive and sign-out when they leave. The start of a shift and the end of a shift always fall within the same week.

This information supports a range of enquiries

- whether an employee is present,
- whether an employee was present on a given day,
- the details of the shift worked on a given day,
- the total hours worked during a given week.
- the shifts worked during a given week.

Though, currently, none of the employees bear the same name, the company assigns each a unique ID number that may be used to disambiguate.

The concept of a "week" bears elaboration. A business week starts on a Monday and ends on a Sunday. The weeks are numbered. The first week of the year is the one that contains the first Thursday of the year.

In the appendix you will find a set of specifications for an `Employee` class and classes that it depends on: `Shift`, `Week`, `CalendarTime`, `Date`, `Time`, `Duration` and `TimeUnit`.

Exercise 4 concerns constructing a UML class diagram for the classes, showing attributes, constructors, methods, and associations.

On the Vula assignment page you will find a ZIP file containing Java code for `Shift`, `Week`, `CalendarTime`, `Date`, `Time`, `Duration` and `TimeUnit`.

Exercise 5 concerns implementing the `Employee` class.

Exercise 4 [10 marks]

Drawing on the described scenario, your task is to study the class specifications in the appendix, and the supplied code, and develop an equivalent UML class diagram.

- Your diagram should depict all the classes.
- For each class you should give public attributes and operations, including those that are static.
- Your diagram should show relationships. Specifically, instances of generalisation, association, and aggregation.
- Associations and aggregations should be annotated with multiplicity.

Exercise 5 [30 marks]

Implement the Employee class as specified in the appendix. To evaluate your work, you should use the interactive features of JGrasp and/or construct test code.

The following (extensive) code fragment demonstrates class behaviour:

```
//
Employee employee = new Employee("Sivuyile Ngesi", "01010125");
System.out.println(employee.name());
System.out.println(employee.UID());
System.out.println(employee.present());
//
System.out.println();
employee.signIn(new Date(1, 9, 2019), new Time(6,00));
System.out.println(employee.present());
employee.signOut(new Date(1, 9, 2019), new Time(18,00));
System.out.println(employee.present());
//
System.out.println();
employee.signIn(new Date(2, 9, 2019), new Time(16, 30));
employee.signOut(new Date(3, 9, 2019), new Time(2, 30));
//
System.out.println();
employee.signIn(new Date(3, 9, 2019), new Time(18,00));
employee.signOut(new Date(4, 9, 2019), new Time(4,00));
//
System.out.println();
System.out.println(employee.worked(new Date(31, 8, 2019)));
System.out.println(employee.worked(new Date(1, 8, 2019)));
//
System.out.println();
System.out.println(employee.worked(new Week(34, 2019)));
System.out.println(employee.worked(new Week(35, 2019)));
System.out.println(employee.worked(new Week(36, 2019)));
//
System.out.println();
List<Shift> shifts = employee.get(new Date(1, 9, 2019));
for(Shift shift : shifts) { System.out.println(shift); }
System.out.println();
shifts = employee.get(new Date(2, 9, 2019));
for(Shift shift : shifts) { System.out.println(shift); }
```

```
System.out.println();
shifts = employee.get(new Date(3, 9, 2019));
for(Shift shift : shifts) { System.out.println(shift); }
//
System.out.println();
shifts = employee.get(new Week(35, 2019));
for(Shift shift : shifts) { System.out.println(shift); }
//
System.out.println();
System.out.println(Duration.format(employee.hours(new Week(35,
2019)), "minute"));
```

The code produces the following output:

```
Sivuyile Ngesi
01010125
false

true
false

false
false

false
true
true

1/9/2019%06:00:00 - 1/9/2019%18:00:00

2/9/2019%16:30:00 - 3/9/2019%02:30:00

2/9/2019%16:30:00 - 3/9/2019%02:30:00
3/9/2019%18:00:00 - 4/9/2019%04:00:00

1/9/2019%06:00:00 - 1/9/2019%18:00:00

12 hours
```

NOTE: Each employee has a collection of shifts. You will need some type of collection object. An ArrayList is recommended.

Marking and Submission

Submit your UML class diagram and the following classes; `ParkingTariff.java`, `TariffTable.java`, `CarParkSim.java` and `Employee.java` classes all contained within a single .ZIP folder to the automatic marker. The zipped folder should have the following naming convention:

```
yourstudentnumber.zip
```

Appendices

Class Employee

An object of this class represents an Employee from the perspective of time keeping. It records the employee name and ID, and logs sign-ins and sign-outs.

Constructors

public Employee(String name, String uid)
// Create an Employee representing the employee with given name and UID.

Methods

public String name()
// Obtain this employee's name.

public void UID()
// Obtain this Employee's ID.

public void signIn(Date d, Time t)
// Record that this employee has begun a shift on the given date and at the given time.

public void signOut(Date d, Time t)
// Record that this employee has completed a shift on the given date and at the given time.

public boolean present()
// Determine whether this employee is present i.e. has signed-in and not yet signed-out.

public boolean worked(Date d)
// Determine whether this employee worked a shift that at least partly occurred on the given date.

public boolean worked(Week w)
// Determine whether this employee worked at least one shift during the given week.

public List<Shift> get(Date d)
// Obtain the shift(s) worked by this employee that at least partly occur on the given date.

public List<Shift> get(Week w)
// Obtain a list of the shifts worked by this employee during the given week.

public Duration hours(Week w)
// Returns the total time (hours and minutes) worked during the given week.

Class Shift

An object of this class represents a work shift. A shift begins on a given date at a given time and ends on a given date at a given time.

Constructors

public Shift(CalendarTime start, CalendarTime finish)
// Create a Shift object representing a shift worked between the given date times.

Methods

public CalendarTime start()
// Obtain the start date and time for this shift.

public CalendarTime finish()
// Obtain the end date and time for this shift.

public boolean inWeek(Week w)
// Determine whether this shift occurred within the given week.

public boolean includesDate(Date date)
// Determine whether this shift at least partly occurred on the given date.

public Duration length()
// Obtain the length of this shift.

public String toString()
// Obtain a string representation of this shift of the form "<date>%:<time>-<date>%:<time>".

Class Week

An object of this class represents a business week. A business week starts on a Monday and ends on a Sunday. The business weeks of a year are numbered. The first week of the year is the one that contains the first Thursday of the year.

Constructors

public Week(int n, int y)
// Create a Week object that represents business week number n in year y.

public Week(String string)
// Create a Week object from a string of the form "<number>/<year>" where "<number>" is up to two digits long, and "<year>" is a 4-digit number.

Methods

public boolean includes(Date d)
// Determine whether this business week includes the given date d.

public String toString()
// Obtain a String representation of this Week of the form "<number>/<year>".

Class CalendarTime

An object of this class represents a calendar time i.e. a date and time.

Instance variables

private Date date;
private Time time;

Constructors

public CalendarTime(Date d, Time t)
// Create a CalendarTime object that represents the given date and time.

Methods

public int compareTo(CalendarTime other)
// Returns a value of -1, 0, or 1 depending on whether this calendar time precedes, is equivalent
// to, or follows the given calendar time, other.

public Date date()
// Returns the date component.

public Time time()
// Returns the time component.

public Duration subtract(CalendarTime other)
// Subtract the given CalendarTime from this CalendarTime.

public String toString()
// Obtain a string representation of this CalendarTime in the form "<date>%<time>".

Class Date

A Date object represents Gregorian calendar date.

Constructors

public Date(int d, int m, int y)
// Create a Date representing day d, month m, and year y.

public Date(String string)
// Create a Date from a string of the form "[d]d/[m]m/yyyy".

Methods

public int compareTo(Date other)
// Compare this Date to other date, returning a -ve number if this precedes other, a zero if they
// are coincident, and a positive number if other precedes this.

public Duration subtract(Date other)
// Obtain the time from the other date to this date.

public String toString()
// Obtain a String representation of this date in the form [d]d/[m]m/yyyy.

Class Time

A Time object represents a twenty-four-hour clock reading composed of hours, minutes and seconds.

Constructors

public Time(int h, int m)

// Create a Time representing the h^{th} hour *and m^{th} minute of the day.*

public Time(String reading)

// Create a Time object from a string representation of a twenty-four-hour clock reading
// of the form 'hh:mm[:ss]' e.g. "03:25", "17:55:05".

Methods

public Duration subtract(Time other)

// Set the first, middle and last names of this Student object.

public String toString()

// Obtain a String representation of this Time object in the form "HH:MM:SS".

Class Duration

A Duration object represents a length of time (with millisecond accuracy).

Constructors

public Duration(String timeUnit, long quantity)

// Create a Duration object that represents the given quantity of the given time unit.
// Permissible time units are: "millisecond", "second", "minute", "hour", "day" ,"week".

Methods

public int compareTo(Duration other)

// Returns a negative, zero, or positive value, depending on whether this duration is smaller, equal
// to, or greater than the other duration.

public boolean equals(Object o)

// Determine whether object o is equivalent to this object i.e. if it is a Duration and of the same
// value as this Duration.

public static String format(final Duration duration, final String smallestUnit)

// Obtain a formatted string that expresses the given duration as a series of no-zero quantities of
// the given time units.
// For example: Given Duration d=new Duration("second", 88893);, the expression
// Duration(d, "second") returns the string "1 day 41 minutes 33 seconds", while
// Duration(d, "minute") returns the string "1 day 41 minutes".

Class TimeUnit

A TimeUnit is a (sub) type of duration that has a name, such as ‘hour’, ‘minute’, ‘week.

Fields (Constants)

```
public final static TimeUnit MILLISECOND;
public final static TimeUnit SECOND;
public final static TimeUnit MINUTE;
public final static TimeUnit HOUR;
public final static TimeUnit DAY;
public final static TimeUnit WEEK;
```

Methods

```
public String getName()
        // Obtain this time unit’s name..

public static TimeUnit[] values()
        // Obtain an array of all the defined time units.

public static TimeUnit parse(String string)
        // Obtain the TimeUnit with the given name. Returns null if the string does not match any of
        // the defined units.
```

END