1 Time for Java API Proposal

This document is a PDF version of the Time for Java API Proposal, which can be found at http://java.time.free.fr/ .

2 Package time

Package Contents *Page*

Interfaces

Classes

This package provide a proposition of classes to replace the `Date` and `Calendar` from `java.util`. It is also available in PDF format (at doc-files/Time API Proposal jor Java.pdf).

Indeed, thoses classes are not very well designed in Java. Their main drawbacks are:

- They should be immutable, so that they can be used as keys in `Map`
- They should be *thread-safe*
- A `java.util.date` does not represent a date in the common meaning of the word.
- A `java.util.calendar` does not represent neither a calendar, but rather a date in a particular calendar.

Several discutions have already point out thoses problems: here (at http://forums.java.net/jive/thread.jspa?threadID=319) here (at http://forums.java.net/jive/thread.jspa?threadID=198) and here (at http://forums.java.net/jive/thread.jspa?threadID=150) for example.

Proposal

The goal of this document is NOT to propose yet another datetime library for Java.

It is rather to explain some concepts that, maybe, should be present in a future release that Sun would one day hopefully provide. Because we *need* standardization about dates in Java.

The main ideas are:

- To separate concepts like:
  - Instant
  - Date
  - Calendar system
- Good integration with Java Collections Framework (at http://java.sun.com/j2se/1.5.0/docs/guide/collection
  - As any Java programmer knows this framework, it should help to use a new date/time package like this one.
- Take advantage of new capabilities of Java 1.5 :
  - Generics
  - Varargs
  - Autoboxing/Unboxing
  - Typesafe Enums
- To provide only few key classes.

Open discussions

As said, the goal of this package is to provide a basis for discussions between Java programmers, and what they would like to have in a decent date/time API. The current proposal defines that:

- A Chronology (at Chronology.html) have some TimeFields (example: YEAR, MONTH, HOUR...)
- A TimeField (at TimeField.html) have only a limited set of possible values (example: 0-23 for hour..., JANUARY..DECEMBER for month)
- A TimeMask (at TimeMask.html) groups several TimeField (so it is a collection of TimeField), and defines a time format
- A Time (at Time.html) object is simply a map which keys are TimeField and values the value of those TimeField.

There is no relation between a Time object and the number of milliseconds since January 1, 1970, 00:00:00 GMT, and this is very important. The idea is to use *theorical* Time objects, that are the same for everyone. Indeed, the only important point with the 20th of October 2005, is that it is between the 19th and the 21st, wether you live in US or in Europe, or anywhere else. We should NOT care about the number millisecond since January 1, 1970, 00:00:00 GMT at this point.

Computations should be done through Chronology methods that allows:

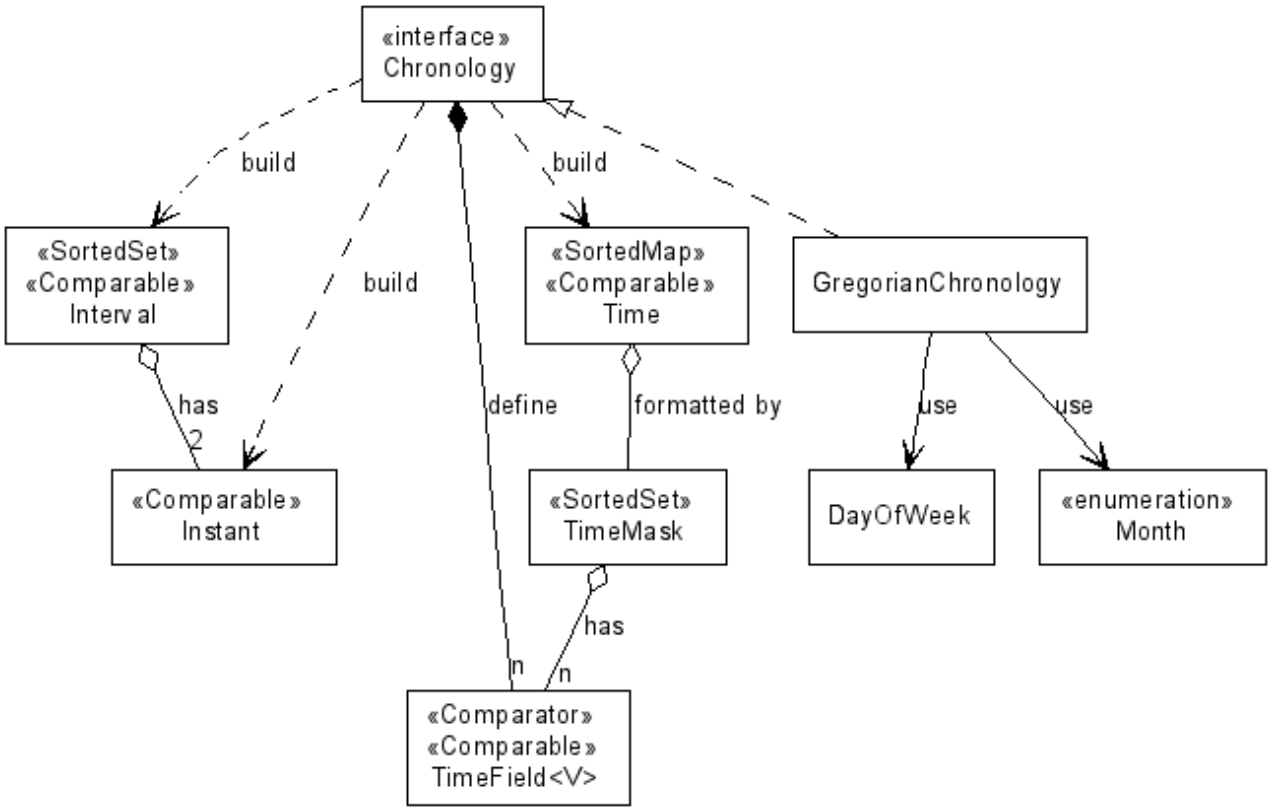
- asSortedSet (at Chronology.html) : to manipulate the Chronology as a `SortedSet` of all possible `Time` objects.
- asList (at Chronology.html) : to manipulate the Chronology as a `List` of all possible `Time` objects.
- split (at Chronology.html) : to manipulate a `Time` object as a `SortedSet` of its components. Examples:
  - a year as a *set of its months*
  - a year as a *set of its weeks*
  - a month as a *set of days*
  - a day is a *set of hours*
  - ...

Note than some operations can be done on `Time` objects, without using a specific chronology:

- compareTo (at Time.html) : to compare chronologicaly `Time` objects.
- entrySet (at Time.html) : useful to test in a `Time` object is part of another.

Of course, a Chronology provides methods to convert such a Time object into Interval (at Interval.html) , that allows to compute the number millisecond since January 1, 1970, 00:00:00 GMT and to provide compatibility with existing Java code.

Class diagram



Example of Code

If you want, here you can have a look at a code sample of usage of this API (at doc-files/sample1.html).

Future

If something important for you is missing in that proposal, feel free to discuss about it, so that this proposal get modified to take your need into account.

Please also note that today, the only implementation of Chronology is GregorianChronology, and is not finished.

Historical

2005, 15th of November	First public release

Copyleft notice

2.1 Interface Chronology

This class represents a system used by humans to describe time.

A chronology provides several `TimeField` that have a meaning in this chronology. One can merge several `TimeField` to create `TimeMask` objects.

Those `TimeMask` objects define *time format* that can be used to describe a precise `Time` object.

If a `Time` object is well defined, it will describe a peace of time (for example, with a `TimeMask` with YEAR, MONTH and DAY). Such a `Time` object is said *consistent for interval* as it can be safely converted in an `Interval` object.

A `Time` object does not necessary have to be *consistent for interval*. For example, it is possible to define a Time object like "9 hour and 15 minutes". This `Time` object is not *consistent for interval*, because it does not define a precise day, just a time in a day. But it still can be usefull, as it may be used to build other `Time` objects. Indeed, you can combinate it with another `Time` object that only defines a day, and the combinaison of both gives a `Time` object that defines day *and* hour/minute.

Please note that having a good `TimeMask` is not sufficient for beeing *consistent for interval*. For example, a `Time` object which value is [31st February of 2006] is *not* consistent for interval.

If a `Time` object provide enough information to be millisecond accurate, it is said *consistent for instant* as it can be safely converted in an `Instant` object. Generally, being *consistent for instant* implies to be *consistent for interval* (which one millisecond interval duration).

A Chronology provides the following services:

- Methods to test `Time` object against *consistence for instant* and *consistence for interval*.
- Methods to make conversion between `Time` and `Instant`/`Interval`
- Methods that provided `java.util.Collection` views of this chronology.

2.1.1 Declaration

public interface Chronology

2.1.2 All known subinterfaces

GregorianChronology (in 2.3, page 13)

2.1.3 All classes known to implement interface

GregorianChronology (in 2.3, page 13)

2.1.4 Methods

- **asList**
  `java.util.List` **asList(** `Time` **startingTime,** `int` **startingPosition** **)**
  - **Description**
    Return a immutable *view* of some `Time` objects of this chronology starting at a specified `Time`. Such a *view* provides a bridge with *Java Collections Framework*.

All `Time` objects of the returned list have the same `TimeMask` as `startingTime`. The first element of the list is `startingTime` if `startingPosition=0`. Otherwise, if `startingPosition`>0, it specifies the position of `startingTime` in the returned list. So we have:

```
asList(myTime, x).get(x).equals(myTime)==true
```

If `startingPosition`<0, the returned list does *not* contains `startingTime` and starts with the `(-startingPosition)` Time objects that follows logically `startingTime`.

To sumurize, `startingPosition` is always the logically (even if it's negative) position of `startingTime` in the returned list.

To have the 100th day following `day1`, call:

```
asList(day1, 0).get(100);
```
or:
```
asList(day1, 100).get(0);
```

Please note than unlike `Set`, a `List` cannot have more than `Integer.MAX_VALUE`.

– **Parameters**
  * `startingTime` – an absolute reference for `Time`
  * `startingPosition` – the logical position of `startingTime` in the returned list
– **Returns** – a immutable *view* of some `Time` objects of this chronology.
– **Throws**
  * `java.lang.NullPointerException` – if `startingTime` is `null`
  * `java.lang.ClassCastException` – if `startingTime` is not compatible with this chronology
  * `java.lang.IllegalArgumentException` – if `startingTime` is not *consistent with interval*
  * `java.lang.IndexOutOfBoundsException` – if the returned List would have more than `Integer.MAX_VALUE` elements.

- **asList**
`java.util.List` **asList**`( TimeMask` **mask** `)`

– **Description**
Return a immutable *view* of all `Time` objects of this chronology with a specified `TimeMask`. Such a *view* provides a bridge with *Java Collections Framework*.

For example, assuming that `myTime` is a `Time` object, calling:

```
asList(myTime.getTimeMask()).contains(myTime)
```

returns `true` if `myTime` is *consistent for interval*.

To have the 100th day following `day1`, call:

```
int idx = asList(dayMask).indexOf(day1);
asList(dayMask).get(idx+100);
```

Please note than unlike `Set`, a `List` cannot have more than `Integer.MAX_VALUE`. So some `mask` that would have define a too big list cannot be used with this method.

– **Parameters**

  - ∗ `mask` – the specified `TimeMask`
- **Returns** – a immutable *view* of all `Time` objects of this chronology with a specified `TimeMask`.
- **Throws**
  - ∗ `java.lang.NullPointerException` – if `mask` is `null`
  - ∗ `java.lang.ClassCastException` – if `mask` is not compatible with this chronology
  - ∗ `java.lang.IllegalArgumentException` – if `mask` is not *consistent with interval*
  - ∗ `java.lang.IndexOutOfBoundsException` – if the returned List would have more than `Integer.MAX_VALUE` elements.

- **asSortedSet**
  `java.util.SortedSet` **asSortedSet(** `TimeMask` **mask)**
  - **Description**
    Return a immutable *view* of all `Time` objects of this chronology with a specified `TimeMask`. Such a *view* provides a bridge with *Java Collections Framework*.
    For example, assuming that `myTime` is a `Time` object, calling:

    ```
    asSortedSet(myTime.getTimeMask()).contains(myTime)
    ```

    returns `true` if `myTime` is *consistent for interval*.
    Another example, assuming that `dayMask` is a `TimeMask` object like [YEAR, MONTH, DAY_OF_MONTH], and that `day1` and `day2` are two differents days, calling:

    ```
    asSortedSet(dayMask).subSet(day1, day2).size()
    ```

    returns the number of days between `day1` and `day2` according to this chronology.
    To have the day following `day1`, call:

    ```
    Iterator it = asSortedSet(day1.getTimeMask()).tailSet(day1).iterator();
    it.next(); // To skip day1
    return it.next()
    ```

    To have the day before `day1`, call:

    ```
    asSortedSet(dayMask).headSet(day1).last()
    ```

  - **Parameters**
    - ∗ `mask` – the specified `TimeMask`
  - **Returns** – a immutable *view* of all `Time` objects of this chronology with a specified `TimeMask`.
  - **Throws**
    - ∗ `java.lang.NullPointerException` – if `mask` is `null`
    - ∗ `java.lang.ClassCastException` – if `mask` is not compatible with this chronology
    - ∗ `java.lang.IllegalArgumentException` – if `mask` is not *consistent with interval*

- **getInstant**
  `Instant` **getInstant(** `Time` **time)** `throws java.lang.ClassCastException`
  - **Description**
    Convert a `Time` object into `Instant` according to this chronology object.
  - **Parameters**

    * `time` – the `Time` object to be converted.
  - **Returns** – `Instant` corresponding to the `t` object.
  - **Throws**
    * `java.lang.ClassCastException` – if `time` does not have a `TimeMask` compatible with this `Chronology` object.
    * `java.lang.IllegalArgumentException` – if `time` is not consistent for instant.
    * `java.lang.NullPointerException` – if `time` is `null`

- **getInterval**
  `Interval` **getInterval**`( Time` **time** `) throws java.lang.ClassCastException`
  - **Description**
    Convert a `Time` object into an `Interval` object, according to this chronology.
  - **Parameters**
    * `time` – the `Time` object to be converted
  - **Returns** – the `Interval` corresponding to `t`
  - **Throws**
    * `java.lang.ClassCastException` – if `time` is not compatible with this chronology.
    * `java.lang.IllegalArgumentException` – if `time` is not consistent for interval.
    * `java.lang.NullPointerException` – if `time` is `null`

- **getTime**
  `Time` **getTime**`( Instant` **instant** `)`
  - **Description**
    Convert an `Instant` object into a `Time` object with all known `TimeField` of this chronology.
  - **Parameters**
    * `instant` – the `Instant` object to be converted.
  - **Returns** – a `Time` object with all known `TimeField` of this chronology.

- **getTime**
  `Time` **getTime**`( Instant` **instant**`, TimeMask` **mask** `) throws java.lang.ClassCastException`
  - **Description**
    Convert an `Instant` object into a `Time` object with a precise `TimeMask` according to this chronology.
    Calling this method should be equivalent as the following:

    ```
    getTime(instant).subMap(mask)
    ```

  - **Parameters**
    * `instant` – the `Instant` object to be converted.
    * `mask` – the `TimeMask` to use.
  - **Returns** – a `Time` object with `mask` as TimeMask.
  - **Throws**
    * `java.lang.ClassCastException` – if `mask` is not compatible with this chronology.
    * `java.lang.NullPointerException` – if `instant` or `mask` is `null`

- **getTime**
  `Time` **getTime**`( Interval` **interval** `)`
  - **Description**
    Convert an `Interval` object into a `Time` object with all known `TimeField` of this chronology.
  - **Parameters**
    * `interval` – the `Interval` object to be converted.
  - **Returns** – a `Time` object with all known `TimeField` of this chronology.
  - **Throws**
    * `java.lang.IllegalArgumentException` – if `interval` cannot be converted into a `Time` object according to this chronology.
    * `java.lang.NullPointerException` – if `interval` is `null`

- **getTime**
  `Time` **getTime**`( Interval` **interval**`, TimeMask` **mask** `) throws java.lang.ClassCastException`
  - **Description**
    Convert an `Interval` object into a `Time` object with a precise `TimeMask` according to this chronology.
    Calling this method should be equivalent as the following:

    ```
    getTime(interval).subMap(mask)
    ```

  - **Parameters**
    * `interval` – the `Interval` object to be converted.
    * `mask` – the `TimeMask` to use.
  - **Returns** – a `Time` object with `mask` as TimeMask..
  - **Throws**
    * `java.lang.IllegalArgumentException` – if `interval` cannot be converted into a `Time` object according to this chronology.
    * `java.lang.ClassCastException` – if `mask` is not compatible with this chronology.
    * `java.lang.NullPointerException` – if `interval` or `mask` is `null`

- **isConsistentForInstant**
  `boolean` **isConsistentForInstant**`( Time` **time** `)`
  - **Description**
    Test if a `Time` object is *consistent for instant*.
    A `Time` object is *consistent for instant* if it provides enough information to point a unique `Instant` in the whole Time.
  - **Parameters**
    * `time` – the `Time` object to be tested.
  - **Returns** – `true` if `time` is *consistant for instant*.
  - **Throws**
    * `java.lang.NullPointerException` – if `time` is `null`

- **isConsistentForInterval**
  `boolean` **isConsistentForInterval**`( Time` **time** `)`

- **Description**
  Test if a `Time` object is *consistent for interval*.
  A `Time` object is *consistent for interval*, if it provides enough information to point a unique `Interval` in the whole Time (for example, a day, a week, a month...).
- **Parameters**
  * `time` – the `Time` object to be tested.
- **Returns** – `true` if `time` is *consistant for interval*.
- **Throws**
  * `java.lang.NullPointerException` – if `time` is `null`

- **knownFields**
  `java.util.SortedSet` **knownFields**`( )`
  - **Description**
    Return all `TimeField` of this chronology.
    All `TimeMask` and `Time` object used with this chronology should only use one or several of thoses `TimeField` objects, otherwise a `ClassCastException` will be thrown by the other methods of this chronology when such an unappropriate `TimeField` will be used.
  - **Returns** – all `TimeField` of this chronology

- **next**
  `Time` **next**`( Time` **time** `) throws java.lang.IllegalArgumentException, java.lang.ClassCastException`
  - **Description**
    Return the next `Time` object that directly follows a specified `Time` object according to this chronology.
    More formally, this method returns a `Time` object that have the same `TimeMask` than `time`, with the following property: the ending instant (according to this chronoloy) of `time` is the starting instant (according to this chronology) of the resulted `Time` object.
    If no exception is thrown, the following test is always `true`:

    ```
    getInterval(time).getEnd().equals(next(time).getStart())
    ```

    This is equivalent of the following call:

    ```
    Iterator it = asSortedSet(time.getTimeMask()).tailSet(time).iterator();
    it.next(); // To skip time
    return it.next()
    ```

  - **Parameters**
    * `time` – the time whose next time is return
  - **Returns** – the `Time` object that directly follows `time`.
  - **Throws**
    * `java.lang.ClassCastException` – if `time` does not have a `TimeMask` compatible with this `Chronology` object.
    * `java.lang.IllegalArgumentException` – if `time` does not have a following Time.

- **previous**
  `Time` **previous**`( Time` **time** `) throws java.lang.IllegalArgumentException, java.lang.ClassCastException`

– **Description**

Return the previous `Time` object that is just before a specified `Time` object according to this chronology.

More formally, this method returns a `Time` object that have the same `TimeMask` than `time`, with the following property: the starting instant (according to this chronoloy) of `time` is the ending instant (according to this chronology) of the resulted `Time` object.

If no exception is thrown, the following test is always `true`:

```
getInterval(time).getStart().equals(previous(time).getEnd())
```

This is equivalent of the following call:

```
asSortedSet(time.getTimeMask()).headSet(time).last()
```

– **Parameters**

∗ `time` – the time whose previous time is return

– **Returns** – the `Time` object that is just before `time`.

– **Throws**

∗ `java.lang.ClassCastException` – if `time` does not have a `TimeMask` compatible with this `Chronology` object.

∗ `java.lang.IllegalArgumentException` – if `time` does not have a previous Time.

• **split**
`java.util.SortedSet` **split(** `Time` **timeToSplit,** `TimeField[]` **fieldsToAdd**)

– **Description**

Return a immutable *view* with `Time` objects by adding some `TimeField` to an existing `Time` object.

The provided `timeToSplit` must be *consistent for interval*. The returned set is a view of `timeToSplit` as set of its component. For example:

∗ a year as a *set of its months*
∗ a year as a *set of its weeks*
∗ a month as a *set of days*
∗ a day is a *set of hours*
∗ ...

The returned set contains `Time` objects whose `TimeMask` is the time mask of `timeToExplode` merged with all `fieldsToAdd`. All possibles values are contained in the returned set.

For example, if one have:

```
timeToSplit = new Time(DAY_MASK, 2005, Month.NOVEMBER, 11);
```

calling the following method:

```
split(timeToSplit, HOUR, MINUTE);
```

return a set that contains all hours/minutes from 00:00 to 23:59.
[2005, NOVEMBER, 11, 0 , 0] ... [2005, NOVEMBER, 11, 0 , 59],
[2005, NOVEMBER, 11, 1 , 0] ... [2005, NOVEMBER, 11, 1 , 59],
[2005, NOVEMBER, 11, 2 , 0] ... [2005, NOVEMBER, 11, 2 , 59],
...
[2005, NOVEMBER, 11, 22 , 0] ... [2005, NOVEMBER, 11, 22 , 59],

[2005, NOVEMBER, 11, 23 , 0] ... [2005, NOVEMBER, 11, 23 , 59]

Another example, if you want to know the number of days in a month:

```
Time myMonth = new Time(MONTH_MASK, 2005, Month.DECEMBER);
int nbDays = myChronology.split(myMonth, DAY_OF_MONTH).size();
```

- **Parameters**
  - `timeToSplit` – the time to split.
  - `fieldsToAdd` – all TimeField to add.
- **Returns** – an immutable *view* of `Time` objects.

2.2 Class DayOfWeek

Definition a day of the week.

Unlike month, this class is not an `Enum`, because `DayOfWeek` does not have a *natural order*. More precisely, even if everyone agrees on the order of weekdays, the first day of the week depends on the local usage.

This class provides also static method to have several appropriate `Comparator` or `SortedSet` where the first conventionnal day of the week has been set.

2.2.1 Declaration

public final class DayOfWeek
extends java.lang.Object

2.2.2 Fields

- public static final DayOfWeek **SUNDAY**
- public static final DayOfWeek **MONDAY**
- public static final DayOfWeek **TUESDAY**
- public static final DayOfWeek **WEDNESDAY**
- public static final DayOfWeek **THURSDAY**
- public static final DayOfWeek **FRIDAY**
- public static final DayOfWeek **SATURDAY**

2.2.3 Methods

- **fromGcValue**
  `public static DayOfWeek` **fromGcValue**`( int gc )`
  - **Description**
    Returns the `DayOfWeek` object with the specified `java.util.Calendar` integer constant.
  - **Parameters**
    - `gc` – a integer from `java.util.Calendar` weekday constants.

- **Returns** – the `DayOfWeek` object with the specified `java.util.Calendar` integer constant.
- **Throws**
  - `java.lang.IllegalArgumentException` – if `gc` is not a valid integer constant.

- **getGcValue**
  `public int` **getGcValue**`( )`
  - **Description**
    Return the `java.util.Calendar` integer constant corresponding to this `DayWeek` object.
    This method should be use only for compatibility with `java.util.Calendar`.
  - **Returns** – the `java.util.Calendar` integer constant corresponding to this `DayWeek` object.

- **specialComparator**
  `public static java.util.Comparator` **specialComparator**`( DayOfWeek` **firstWeekDay** `)`
  - **Description**
    Return a `Comparator` for sorting `DayOfWeek`.
    The conventionnal first week of the day must be provided, so that the `Comparator` can do his job.
  - **Parameters**
    - `firstWeekDay` – conventionnal first day of week
  - **Returns** – a `Comparator` that compare `DayOfWeek` objects using a precise first day of week

- **specialSet**
  `public static java.util.SortedSet` **specialSet**`( DayOfWeek` **firstWeekDay** `)`
  - **Description**
    Return a immutable `SortedSet` with all seven `DayOfWeek`.
    The conventionnal first week of the day must be provided.
  - **Parameters**
    - `firstWeekDay` – conventionnal first day of week
  - **Returns** – a `SortedSet` with all seven `DayOfWeek`, starting with `firstWeekDay`

- **toString**
  `public java.lang.String` **toString**`( )`
  - **Description**
    Return a `String` describing the `DayOfWeek`
  - **Returns** – a `String` describing the `DayOfWeek`

- **valueOf**
  `public static DayOfWeek` **valueOf**`( java.lang.String` **name** `)`
  - **Description**
    Returns the `DayOfWeek` object with the specified name.
  - **Parameters**
    - `name` – the name of the weekday to be returned
  - **Returns** – the `DayOfWeek` object with the specified name.
  - **Throws**
    - `java.lang.IllegalArgumentException` – if `name` is not a valid weekday name.

2.3 Class GregorianChronology

This class provides a concrete gregorian chronology. It is backed by a `java.util.GregorianChronology`. It is written to make a example of Chronology. This class is immutable and *thread-safe*.

Please note that this current implementation is not still under work, and not finished.

2.3.1 Declaration

public final class GregorianChronology
extends java.lang.Object
implements Chronology

2.3.2 Fields

- public static final TimeField **YEAR**
  - Year, between -9999 and +9999. Unlike GregorianCalendar, there is no concept of ERA in this chronology. It means that "1 BC" Year in Gregorian calendar is "0" in this chronology, and "2 BC" is "-1"...
- public static final TimeField **MONTH**
  - Month, which values must be a `Month` (at Month.html) object, from JANUARY to DECEMBER
- public static final TimeField **DAY_OF_MONTH**
  - Day of the month, from 1 to 31
- public static final TimeField **DAY_OF_YEAR**
  - Day of year, from 1 to 366
- public static final TimeField **WEEK_OF_YEAR**
  - Week of year, from 1 to 53
- public static final TimeField **HOUR_OF_DAY**
  - Hour of the day, from 0 to 23
- public static final TimeField **MINUTE**
  - Minute of the hour, from 0 to 59
- public static final TimeField **SECOND**
  - Second, from 0 to 59
- public static final TimeField **MILLISECOND**
  - Millisecond, from 0 to 999
- public final TimeField **DAY_OF_WEEK**
  - Day of the week, which values if a `DayOfWeek` (at DayOfWeek.html). Unlike other `TimeField` of the GregorianChronology, this field may vary upon different instance of object, as the order of weekday depends on the Locale.

    On the other hand, this implementation garantees that if two different `GrogorianChronology` have the same first dayweek, they will share the same `DAY_OF_WEEK` field.

2.3.3 Constructors

- **GregorianChronology**
  `public` **GregorianChronology**`( )`
  - **Description**
    Build a new `GregorianChronology` backed by a default `GregorianCalendar` using the default time zone and the default locale.
- **GregorianChronology**
  `public` **GregorianChronology**`( java.util.GregorianCalendar gc )`
  - **Description**
    Build a new `GregorianChronology` backed by a specific `GregorianCalendar`.
    To garantee immutability, `gc` is cloned.
  - **Parameters**
    * `gc` – calendar used by this Chronology.

2.3.4 Methods

- **asList**
  `java.util.List` **asList**`( Time startingTime, int startingPosition )`
  - **Description copied from Chronology (in 2.1, page 4)**
    Return a immutable *view* of some `Time` objects of this chronology starting at a specified `Time`. Such a *view* provides a bridge with *Java Collections Framework*.
    All `Time` objects of the returned list have the same `TimeMask` as `startingTime`. The first element of the list is `startingTime` if `startingPosition=0`. Otherwise, if `startingPosition>0`, it specifies the position of `startingTime` in the returned list. So we have:

    ```
    asList(myTime, x).get(x).equals(myTime)==true
    ```

    If `startingPosition<0`, the returned list does *not* contains `startingTime` and starts with the `(-startingPosition)` Time objects that follows logically `startingTime`.
    To sumurize, `startingPosition` is always the logically (even if it's negative) position of `startingTime` in the returned list.
    To have the 100th day following `day1`, call:

    ```
    asList(day1, 0).get(100);
    ```
    or:
    ```
    asList(day1, 100).get(0);
    ```

    Please note than unlike `Set`, a `List` cannot have more than `Integer.MAX_VALUE`.
  - **Parameters**
    * `startingTime` – an absolute reference for `Time`
    * `startingPosition` – the logical position of `startingTime` in the returned list
  - **Returns** – a immutable *view* of some `Time` objects of this chronology.
  - **Throws**
    * `java.lang.NullPointerException` – if `startingTime` is `null`

  * `java.lang.ClassCastException` – if `startingTime` is not compatible with this chronology
  * `java.lang.IllegalArgumentException` – if `startingTime` is not *consistent with interval*
  * `java.lang.IndexOutOfBoundsException` – if the returned List would have more than `Integer.MAX_VALUE` elements.

- **asList**
  `java.util.List` **asList**`( TimeMask` **mask** `)`
  - **Description copied from Chronology (in 2.1, page 4)**
    Return a immutable *view* of all `Time` objects of this chronology with a specified `TimeMask`. Such a *view* provides a bridge with *Java Collections Framework*.
    For example, assuming that `myTime` is a `Time` object, calling:

    ```
    asList(myTime.getTimeMask()).contains(myTime)
    ```

    returns `true` if `myTime` is *consistent for interval*.
    To have the 100th day following `day1`, call:

    ```
    int idx = asList(dayMask).indexOf(day1);
    asList(dayMask).get(idx+100);
    ```

    Please note than unlike `Set`, a `List` cannot have more than `Integer.MAX_VALUE`. So some `mask` that would have define a too big list cannot be used with this method.
  - **Parameters**
    * `mask` – the specified `TimeMask`
  - **Returns** – a immutable *view* of all `Time` objects of this chronology with a specified `TimeMask`.
  - **Throws**
    * `java.lang.NullPointerException` – if `mask` is `null`
    * `java.lang.ClassCastException` – if `mask` is not compatible with this chronology
    * `java.lang.IllegalArgumentException` – if `mask` is not *consistent with interval*
    * `java.lang.IndexOutOfBoundsException` – if the returned List would have more than `Integer.MAX_VALUE` elements.

- **asSortedSet**
  `java.util.SortedSet` **asSortedSet**`( TimeMask` **mask** `)`
  - **Description copied from Chronology (in 2.1, page 4)**
    Return a immutable *view* of all `Time` objects of this chronology with a specified `TimeMask`. Such a *view* provides a bridge with *Java Collections Framework*.
    For example, assuming that `myTime` is a `Time` object, calling:

    ```
    asSortedSet(myTime.getTimeMask()).contains(myTime)
    ```

    returns `true` if `myTime` is *consistent for interval*.
    Another example, assuming that `dayMask` is a `TimeMask` object like [YEAR, MONTH, DAY_OF_MONTH], and that `day1` and `day2` are two differents days, calling:

    ```
    asSortedSet(dayMask).subSet(day1, day2).size()
    ```

returns the number of days between `day1` and `day2` according to this chronology.
To have the day following `day1`, call:

```
Iterator it = asSortedSet(day1.getTimeMask()).tailSet(day1).iterator();
it.next(); // To skip day1
return it.next()
```

To have the day before `day1`, call:

```
asSortedSet(dayMask).headSet(day1).last()
```

- **Parameters**
  - `mask` – the specified `TimeMask`
- **Returns** – a immutable *view* of all `Time` objects of this chronology with a specified `TimeMask`.
- **Throws**
  - `java.lang.NullPointerException` – if `mask` is `null`
  - `java.lang.ClassCastException` – if `mask` is not compatible with this chronology
  - `java.lang.IllegalArgumentException` – if `mask` is not *consistent with interval*

- **getInstant**
  `Instant` **getInstant**`( Time` **time** `) throws java.lang.ClassCastException`
  - **Description copied from Chronology (in 2.1, page 4)**
    Convert a `Time` object into `Instant` according to this chronology object.
  - **Parameters**
    - `time` – the `Time` object to be converted.
  - **Returns** – `Instant` corresponding to the `t` object.
  - **Throws**
    - `java.lang.ClassCastException` – if `time` does not have a `TimeMask` compatible with this `Chronology` object.
    - `java.lang.IllegalArgumentException` – if `time` is not consistent for instant.
    - `java.lang.NullPointerException` – if `time` is `null`

- **getInterval**
  `Interval` **getInterval**`( Time` **time** `) throws java.lang.ClassCastException`
  - **Description copied from Chronology (in 2.1, page 4)**
    Convert a `Time` object into an `Interval` object, according to this chronology.
  - **Parameters**
    - `time` – the `Time` object to be converted
  - **Returns** – the `Interval` corresponding to `t`
  - **Throws**
    - `java.lang.ClassCastException` – if `time` is not compatible with this chronology.
    - `java.lang.IllegalArgumentException` – if `time` is not consistent for interval.
    - `java.lang.NullPointerException` – if `time` is `null`

- **getTime**
  `Time` **getTime**`( Instant` **instant** `)`

- **Description copied from Chronology** (**in 2.1, page 4**)
  Convert an `Instant` object into a `Time` object with all known `TimeField` of this chronology.
- **Parameters**
  - `instant` – the `Instant` object to be converted.
- **Returns** – a `Time` object with all known `TimeField` of this chronology.

- **getTime**
  `Time` **getTime(** `Instant` **instant,** `TimeMask` **mask)** `throws java.lang.ClassCastException`

  - **Description copied from Chronology** (**in 2.1, page 4**)
    Convert an `Instant` object into a `Time` object with a precise `TimeMask` according to this chronology.
    Calling this method should be equivalent as the following:

    ```
    getTime(instant).subMap(mask)
    ```

  - **Parameters**
    - `instant` – the `Instant` object to be converted.
    - `mask` – the `TimeMask` to use.
  - **Returns** – a `Time` object with `mask` as TimeMask.
  - **Throws**
    - `java.lang.ClassCastException` – if `mask` is not compatible with this chronology.
    - `java.lang.NullPointerException` – if `instant` or `mask` is `null`

- **getTime**
  `Time` **getTime(** `Interval` **interval)**

  - **Description copied from Chronology** (**in 2.1, page 4**)
    Convert an `Interval` object into a `Time` object with all known `TimeField` of this chronology.
  - **Parameters**
    - `interval` – the `Interval` object to be converted.
  - **Returns** – a `Time` object with all known `TimeField` of this chronology.
  - **Throws**
    - `java.lang.IllegalArgumentException` – if `interval` cannot be converted into a `Time` object according to this chronology.
    - `java.lang.NullPointerException` – if `interval` is `null`

- **getTime**
  `Time` **getTime(** `Interval` **interval,** `TimeMask` **mask)** `throws java.lang.ClassCastException`

  - **Description copied from Chronology** (**in 2.1, page 4**)
    Convert an `Interval` object into a `Time` object with a precise `TimeMask` according to this chronology.
    Calling this method should be equivalent as the following:

    ```
    getTime(interval).subMap(mask)
    ```

- **Parameters**
  - `interval` – the `Interval` object to be converted.
  - `mask` – the `TimeMask` to use.
- **Returns** – a `Time` object with `mask` as TimeMask..
- **Throws**
  - `java.lang.IllegalArgumentException` – if `interval` cannot be converted into a `Time` object according to this chronology.
  - `java.lang.ClassCastException` – if `mask` is not compatible with this chronology.
  - `java.lang.NullPointerException` – if `interval` or `mask` is `null`

- **isConsistentForInstant**
  `boolean` **isConsistentForInstant**`( Time` **time** `)`
  - **Description copied from Chronology (in 2.1, page 4)**
    Test if a `Time` object is *consistent for instant*.
    A `Time` object is *consistent for instant* if it provides enough information to point a unique `Instant` in the whole Time.
  - **Parameters**
    - `time` – the `Time` object to be tested.
  - **Returns** – `true` if `time` is *consistant for instant*.
  - **Throws**
    - `java.lang.NullPointerException` – if `time` is `null`

- **isConsistentForInterval**
  `boolean` **isConsistentForInterval**`( Time` **time** `)`
  - **Description copied from Chronology (in 2.1, page 4)**
    Test if a `Time` object is *consistent for interval*.
    A `Time` object is *consistent for interval*, if it provides enough information to point a unique `Interval` in the whole Time (for example, a day, a week, a month...).
  - **Parameters**
    - `time` – the `Time` object to be tested.
  - **Returns** – `true` if `time` is *consistant for interval*.
  - **Throws**
    - `java.lang.NullPointerException` – if `time` is `null`

- **knownFields**
  `java.util.SortedSet` **knownFields**`( )`
  - **Description copied from Chronology (in 2.1, page 4)**
    Return all `TimeField` of this chronology.
    All `TimeMask` and `Time` object used with this chronology should only use one or several of thoses `TimeField` objects, otherwise a `ClassCastException` will be thrown by the other methods of this chronology when such an unappropriate `TimeField` will be used.
  - **Returns** – all `TimeField` of this chronology

- **next**
  `Time` **next**`( Time` **time** `) throws java.lang.IllegalArgumentException, java.lang.ClassCastException`

- **Description copied from Chronology** (**in 2.1, page 4**)
  Return the next `Time` object that directly follows a specified `Time` object according to this chronology.
  More formally, this method returns a `Time` object that have the same `TimeMask` than `time`, with the following property: the ending instant (according to this chronoloy) of `time` is the starting instant (according to this chronology) of the resulted `Time` object.
  If no exception is thrown, the following test is always `true`:

  ```
  getInterval(time).getEnd().equals(next(time).getStart())
  ```

  This is equivalent of the following call:

  ```
  Iterator it = asSortedSet(time.getTimeMask()).tailSet(time).iterator();
  it.next(); // To skip time
  return it.next()
  ```

- **Parameters**
  - `time` – the time whose next time is return
- **Returns** – the `Time` object that directly follows `time`.
- **Throws**
  - `java.lang.ClassCastException` – if `time` does not have a `TimeMask` compatible with this `Chronology` object.
  - `java.lang.IllegalArgumentException` – if `time` does not have a following Time.

- **previous**
  `Time` **previous**`( Time time ) throws java.lang.IllegalArgumentException, java.lang.ClassCastException`

  - **Description copied from Chronology** (**in 2.1, page 4**)
    Return the previous `Time` object that is just before a specified `Time` object according to this chronology.
    More formally, this method returns a `Time` object that have the same `TimeMask` than `time`, with the following property: the starting instant (according to this chronoloy) of `time` is the ending instant (according to this chronology) of the resulted `Time` object.
    If no exception is thrown, the following test is always `true`:

    ```
    getInterval(time).getStart().equals(previous(time).getEnd())
    ```

    This is equivalent of the following call:

    ```
    asSortedSet(time.getTimeMask()).headSet(time).last()
    ```

  - **Parameters**
    - `time` – the time whose previous time is return
  - **Returns** – the `Time` object that is just before `time`.
  - **Throws**
    - `java.lang.ClassCastException` – if `time` does not have a `TimeMask` compatible with this `Chronology` object.
    - `java.lang.IllegalArgumentException` – if `time` does not have a previous Time.

- **split**
  `java.util.SortedSet` **split(** `Time` **timeToSplit,** `TimeField[]` **fieldsToAdd** **)**

  – **Description copied from Chronology (in 2.1, page 4)**
  Return a immutable *view* with `Time` objects by adding some `TimeField` to an existing `Time` object.
  The provided `timeToSplit` must be *consistent for interval*. The returned set is a view of `timeToSplit` as set of its component. For example:
  * a year as a *set of its months*
  * a year as a *set of its weeks*
  * a month as a *set of days*
  * a day is a *set of hours*
  * ...

  The returned set contains `Time` objects whose `TimeMask` is the time mask of `timeToExplode` merged with all `fieldsToAdd`. All possibles values are contained in the returned set.
  For example, if one have:

```
timeToSplit = new Time(DAY_MASK, 2005, Month.NOVEMBER, 11);
```

  calling the following method:

```
split(timeToSplit, HOUR, MINUTE);
```

  return a set that contains all hours/minutes from 00:00 to 23:59.
  [2005, NOVEMBER, 11, 0 , 0] ... [2005, NOVEMBER, 11, 0 , 59],
  [2005, NOVEMBER, 11, 1 , 0] ... [2005, NOVEMBER, 11, 1 , 59],
  [2005, NOVEMBER, 11, 2 , 0] ... [2005, NOVEMBER, 11, 2 , 59],
  ...
  [2005, NOVEMBER, 11, 22 , 0] ... [2005, NOVEMBER, 11, 22 , 59],
  [2005, NOVEMBER, 11, 23 , 0] ... [2005, NOVEMBER, 11, 23 , 59]

  Another example, if you want to know the number of days in a month:

```
Time myMonth = new Time(MONTH_MASK, 2005, Month.DECEMBER);
int nbDays = myChronology.split(myMonth, DAY_OF_MONTH).size();
```

  – **Parameters**
    * `timeToSplit` – the time to split.
    * `fieldsToAdd` – all TimeField to add.
  – **Returns** – an immutable *view* of `Time` objects.

2.4 Class Instant

`Instant` is the implementation of a fully immutable instant in time. An instant in time represents an event that have theorically no duration.

An Instant represents a point in the universal line of time, which is supposed to be the same everywhere in the whole Universe. (This is a Newtonian vision of the Time which is known as approximated since Einstein's Theory of Relativity, which shows that space and time are linked, but this approximation is fine for us).

Here some examples of famous instants:

- the revolt of Spartacus (at http://en.wikipedia.org/wiki/Spartacus),
- the supernovae of the Crab Nebula (at http://www.seds.org/messier/more/m001_sn.html), observed by Chinese astronomers almost 1000 years ago,
- the birth of Beethoven,
- the first man on Moon.

Instant are conceptualy independant from Human representations of Time (calendar...). They simply exists by themself.

To ensure compatibility with existing classes in Java, `Instant` are implemented as milliseconds since the Java Epoch of 1970-01-01 00:00:00 GMT, but please note that this is only an implementation detail. The current implementation is then milliseconds accurate, but future implementation may offer finer accuracy (nanoseconds).

Instants are `Comparable`, and their natural order is the chronological order.

Two Instants are equals if they represent the same instant in time (they occur at exactly the same moment).

As `Instant` are immutable, they are fully thread-safe.

Note: Technically, an `Instant` looks like a immutable `java.util.Date`. But you should be aware that conceptually, they are very different, since a `java.util.Date` is supposed to be a date.

2.4.1 Declaration

public final class Instant
extends java.lang.Object
implements java.lang.Comparable

2.4.2 Fields

- public static final Instant **EPOCH**
  - The famous Epoch. (1970-01-01 00:00:00)
- public static final Instant **BIG_BANG**
  - The minimum value of an Instant, about 292 millions of years before the Epoch. We cannot have instant before this BIG_BANG. (This is not the *real* astrophysical BigBang, which have occured between 10 and 20 billions of years before the Epoch).
- public static final Instant **APOCALYPSE**
  - The maximum value of an Instant, which will happen about 292 millions of years after the Epoch. This is not the *real* Apocalypse.

2.4.3 Constructors

- **Instant**
  `public` **Instant**`( )`
  - **Description**
    Create a new `Instant`, which represents the current time (now).
- **Instant**
  `public` **Instant**`( java.util.Date` **date** `)`

- **Description**
  Create a new `Instant`. This constructor is provided for compatibility with existing `java.util.Date`.
- **Parameters**
  * `date` – Date of the Instant to create.
- **Throws**
  * `java.lang.NullPointerException` – if `date` is null.

- **Instant**
  `public` **Instant**`( long` **millis** `)`
  - **Description**
    Create a new `Instant`. This constructor is provided for compatibility with existing code.
  - **Parameters**
    * `millis` – number of milliseconds since the Epoch of the Instant to create

2.4.4 Methods

- **asImmutableDate**
  `public java.util.Date` **asImmutableDate**`( )`
  - **Description**
    Return a `java.util.Date` view of this Instant.
    The returned date is immutable, as the returned class is a subclass of `java.util.Date` which all mutating methods have been overidden to throw a `UnsupportedOperationException`.
  - **Returns** – the immutable date representing the same instant as this object.

- **compareTo**
  `public int` **compareTo**`( Instant` **otherInstant** `)`
  - **Description**
    Compare two instants chronogically.
  - **Parameters**
    * `otherInstant` – the instant to be compared.
  - **Returns** – the value 0 if the argument instant is equal to this instant; a value less than 0 if this instant occurs before the instant argument; and a value greater than 0 if this instant occurs after than the instant argument.
  - **Throws**
    * `java.lang.NullPointerException` – if `otherInstant` is `null`.

- **equals**
  `public boolean` **equals**`( java.lang.Object` **obj** `)`
  - **Description**
    Compare two instants for equality. The result is `true` if and only if the argument is not `null` and is a `Instant` object that represents the same point in time, (with a millisecond accuracy in this implementation) , than this object.
  - **Parameters**
    * `obj` – the object to compare with.

  - **Returns** – `true` if the objects are the same; `false` otherwise.

- **getMillis**
  `public long` **getMillis**`( )`
  - **Description**
    Returns the number of milliseconds since the Epoch represented by this Instant object.
  - **Returns** – the number of milliseconds since the Epoch represented by this Instant object.

- **getNanos**
  `public long` **getNanos**`( )`
  - **Description**
    Returns `0L` in this current implementation..
  - **Returns** – 0L.

- **hashCode**
  `public int` **hashCode**`( )`
  - **Description**
    Returns a hash code value for this object.

- **millisBetween**
  `public long` **millisBetween**`( Instant` **otherInstant** `)`
  - **Description**
    Compute the number of milliseconds between this instant and another instant.
    If this instant occurs after the other instant, the result is positive. If this instant occurs before the other instant, the result is negative.
  - **Returns** – the number of milliseconds between this instant and another instant.
  - **Throws**
    * `java.lang.NullPointerException` – if `otherInstant` is `null`.

- **toString**
  `public java.lang.String` **toString**`( )`
  - **Description**
    Convert this `Instant` to a `String`.
  - **Returns** – a string representation of this instant

2.5 Class Interval

Interval is the implementation of an immutable time interval. A time interval represents a period of time between two instants.

An `Interval` contains all `Instant` that are greater than the start instant (inclusive) and lower than the end instant (exclusive), using the natural order of `Instant` (that is the chronological order). So the end instant is always greater than or equal to the start instant.

As an `Interval` can be seen as a sorted set of instant, it implements the `java.util.SortedSet<Instant>` interface. And as an `Interval` is immutable, the `SortedSet` is also immutable: every mutative method of the `SortedSet` interface throws an `UnsupportedOperationException` as the `Collection` interface allows it.

The `Interval` has not been designed to be iterated through `Iterator`, because it can contain a lot of `Instant` objects. For example, if an interval has a duration of one simple day, since the current

implementation of `Instant` is millisecond accurate, it means that it contains 86 400 000 instants. Iterating such an interval will be likely unappropriate (but possible if really needed). And as `Instant` may be in the future event more accurate (nanoseconds), `Interval` may contain event more `Instant`, so you should also not use the `size()` method. The `getDuration()` is the appropriate method to know the duration of an Interval.

Interval have an natural order, which is consistent with `equals`: they are sorted by their starting intant first. If two intervals have the same starting instant, they are sorted by their ending instant. Other `Comparator` are also provided:

- `COMPARATOR_BY_STARTING_ONLY`
- `COMPARATOR_BY_ENDING_ONLY`
- `COMPARATOR_BY_ENDING_THEN_STARTING`

2.5.1 See also

– `Instant` (in 2.4, page 20)

2.5.2 Declaration

public final class Interval
extends java.lang.Object
implements java.lang.Comparable, java.util.SortedSet

2.5.3 Fields

- public static final java.util.Comparator **COMPARATOR_BY_STARTING_ONLY**
  - Comparator that compares Intervals by their starting instant.
    Please not that this comparator is not consistent with `equals`, since all Intervals that have the same starting instant are equals according to this comparator, even if they have different ending instants.
- public static final java.util.Comparator **COMPARATOR_BY_ENDING_ONLY**
  - Comparator that compares Intervals by their ending instant.
    Please not that this comparator is not consistent with `equals`, since all Intervals that have the same ending instant are equals according to this comparator, even if they have different starting instants.
- public static final java.util.Comparator **COMPARATOR_BY_STARTING_THEN_ENDING**
  - Comparator that compares Intervals by their starting instant first, then their ending instant (natural order of `Interval`).
    This comparator is consistent with `equals`.
- public static final java.util.Comparator **COMPARATOR_BY_ENDING_THEN_STARTING**
  - Comparator that compares Intervals by their ending instant first, then their starting instant.
    This comparator is consistent with `equals`.
- public static final Interval **WHOLE_TIME**
  - An interval that contains the whole representation of time. Its beginning is the BigBang and its ending is the Apocalypse. So all interval are contained in this one.
    This interval contains all known `Instant` objects except the Apocalypse, as the end instant is not included in an interval. (The Apocalypse cannot belong to any interval).

It can be used with `headSet()` or `tailSet()`. For example:
`Interval allBefore = WHOLE_TIME.headSet(limit);` Represents an Interval which contains all Instants strictly before `limit`.

And another example:
`Interval allAfter = WHOLE_TIME.tailSet(limit);` Represents an Interval which contains all Instants after `limit`, including limit.

- See also
  - `Instant.BIG_BANG` (in 2.4.2, page 21)
  - `Instant.APOCALYPSE` (in 2.4.2, page 21)

2.5.4 Constructors

- **Interval**
  `public` **Interval(** `Instant` **start,** `Instant` **end)**
  - **Description**
    Create a new Interval.
  - **Parameters**
    - `start` – the starting instant of the new interval
    - `end` – the ending instant of the new interval
  - **Throws**
    - `java.lang.NullPointerException` – if end or start are `null`
    - `java.lang.IllegalArgumentException` – if the ending instant is before the starting instant.

2.5.5 Methods

- **add**
  `public boolean` **add(** `Instant` **o)**
  - **Description**
    As `Interval` are immutable, throws a `UnsupportedOperationException`.
- **addAll**
  `public boolean` **addAll(** `java.util.Collection` **c)**
  - **Description**
    As `Interval` are immutable, throws a `UnsupportedOperationException`.
- **clear**
  `public void` **clear(** **)**
  - **Description**
    As `Interval` are immutable, throws a `UnsupportedOperationException`.
- **comparator**
  `public java.util.Comparator` **comparator(** **)**
  - **Description**
    Returns `null`, as Interval uses the natural order of Instant.
- **compareTo**
  `public int` **compareTo(** `Interval` **otherInterval)**

- **Description**
  Compare two interval chronogically.
  The order is the same as the one used by COMPARATOR_BY_STARTING_THEN_ENDING. The interval are sorted by their starting instant first, then by their ending instant.
- **Parameters**
  * `otherInterval` – the instant to be compared.
- **Returns** – the value 0 if the argument interval is equal to this interval; a value less than 0 if this interval is before the interval argument; and a value greater than 0 if this interval is after than the interval argument.
- **Throws**
  * `java.lang.NullPointerException` – if `otherInterval` is null.
- **See also**
  * `Interval.COMPARATOR_BY_STARTING_THEN_ENDING` (in 2.5.3, page 24)

• **contains**
`public boolean` **contains**`( java.lang.Object` **obj** `)`

- **Description**
  Returns true if this set contains the specified Instant.
  * If `obj` is `null`, a `NullPointerException` is thrown.
  * If `obj` is not an `Instant`, a `ClassCastException` is thrown.
  * If `obj` is an `Instant` which occurs after the starting instant of the interval (not strictly) and before the ending instant (strictly), return `true`. Return `false` otherwise.
- **Throws**
  * `java.lang.NullPointerException` – if `obj` is `null`.
  * `java.lang.ClassCastException` – if `obj` is not an `Instant`.

• **containsAll**
`public boolean` **containsAll**`( java.util.Collection` **c** `)`

- **Description**
  Returns `true` if this set contains all of the elements of the specified collection.
  * If the specified collection is an `Interval` object, this method returns `true` if it is a subintervall of this interval. The method is in this case executed in constant time
  * If the specified collection is a set, this method returns `true` if it is a subset of this interval.
- **Parameters**
  * `c` – collection to be checked for containment in this interval.
- **Returns** – `true` if this interval contains all of the elements of the specified collection.
- **Throws**
  * `java.lang.NullPointerException` – if `c` is `null`.
  * `java.lang.NullPointerException` – if `c` contains one or more null elements.
  * `java.lang.ClassCastException` – if `c` contains one or more elements that are not `Instant`.

• **equals**
`public boolean` **equals**`( java.lang.Object` **obj** `)`

- **Description**
  Compare two interval for equality. The result is `true` if and only if the argument is not `null` and is a `Interval` object that represents the same Interval, as this object.

- **Parameters**
  - `obj` – the object to compare with.
- **Returns** – `true` if the objects are the same; `false` otherwise.

- **first**
  `public Instant first( )`
  - **Description**
    Returns the first (lowest) Instant currently in this Interval.
    If the interval is not empty, it is the same instant as the one returned by the `getStart()` method.
  - **Throws**
    - `java.util.NoSuchElementException` – if the interval is empty.
  - **See also**
    - `Interval.getStart()` (in 2.5.5, page 28)

- **getDuration**
  `public long getDuration( )`
  - **Description**
    Compute the duration in milliseconds of the Interval. If the duration is bigger than `Long.MAX_VALUE`, return `Long.MAX_VALUE` (which happens for `WHOLE_TIME` for example).
  - **Returns** – the duration in milliseconds of the Interval
  - **See also**
    - `Interval.WHOLE_TIME` (in 2.5.3, page 24)

- **getEnd**
  `public Instant getEnd( )`
  - **Description**
    Return the ending instant of this interval. This method is almost the same as the `last()` method, with two exception:
    - if the interval is not empty, the following test :
      `getEnding().millisBetween(last())==1L` is always `true`,
    - if the interval is empty, (which means that the starting and ending instants are the same), this method will return the ending instant.
  - **See also**
    - `Interval.last()` (in 2.5.5, page 28)

- **getIntersection**
  `public Interval getIntersection( Interval otherInterval )`
  - **Description**
    Return the intersection between this interval and another interval. More formally, return the larger interval that are contained by both this interval and the other interval.
    Return `null` if there is no intersection, ie the two intervals do not intersect at all.
    If the ending of an interval is the starting of the other, return an interval that have a duration of 0.
  - **Returns** – the intersection or `null` is there is no intersection.
  - **Throws**

* `java.lang.NullPointerException` – if `otherInterval` is null.

- **getStart**
  `public Instant` **getStart**`( )`
  - **Description**
    Return the starting instant of this interval. This method is almost the same as the `first()` method, except that even if the interval is empty (which means that the starting and ending instants are the same), this method will return the starting instant.
  - **See also**
    * `Interval.first()` (in 2.5.5, page 27)

- **hashCode**
  `public int` **hashCode**`( )`
  - **Description**
    Returns a hash code value for this object.

- **headSet**
  `public Interval` **headSet**`( Instant` **toInstant** `)`
  - **Description**
    Return a new interval that contains all instants of this interval strictly less than `toInstant`.
  - **Parameters**
    * `toInstant` – ending instant (exclusive) of the new interval.
  - **Returns** – new interval that contains all instants of this interval strictly less than `toInstant`.
  - **Throws**
    * `java.lang.NullPointerException` – if `toInstant` is `null`.
    * `java.lang.IllegalArgumentException` – if `toInstant` is not in the interval.

- **isEmpty**
  `public boolean` **isEmpty**`( )`
  - **Description**
    Test if the Interval is empty (has a duration of `0L`).
  - **Returns** – `true` if the duration is 0L.

- **iterator**
  `public java.util.Iterator` **iterator**`( )`
  - **Description**
    Returns an iterator over the instants in this interval. The instants are returned in chronological order.
  - **Returns** – an iterator over the instants in this interval.

- **last**
  `public Instant` **last**`( )`
  - **Description**
    Returns the last (highest) Instant currently in this Interval.
    If the interval is not empty, it differs from 1 milliseconds from the instant returned by `getEnd()`, as the Instant returned by `last()` belongs to this interval and the Instant returned by `getEnd()` does not.

- **Throws**
  - `java.util.NoSuchElementException` – if the interval is empty.
- **See also**
  - `Interval.getEnd()` (in 2.5.5, page 27)

- **remove**
  `public boolean` **remove**`( java.lang.Object` **o** `)`
  - **Description**
    As `Interval` are immutable, throws a `UnsupportedOperationException`.

- **removeAll**
  `public boolean` **removeAll**`( java.util.Collection` **c** `)`
  - **Description**
    As `Interval` are immutable, throws a `UnsupportedOperationException`.

- **retainAll**
  `public boolean` **retainAll**`( java.util.Collection` **c** `)`
  - **Description**
    As `Interval` are immutable, throws a `UnsupportedOperationException`.

- **size**
  `public int` **size**`( )`
  - **Description**
    Return the number of `Instant` in this Interval.
    As `Instant` are millisecond accurate, an `Interval` can contain a lot of `Instant`. If the `Interval` is larger than about 24 days, it will contains more than `Integer.MAX_VALUE` elements, so this method will return `Integer.MAX_VALUE`, as specified in `Collection` interface. Since future implementations of `Instant` may be more accurate than millisecond, the value returned by this method may be even larger in the future. Please considere the usage of `getDuration()` instead.
  - **Returns** – the number of Instants in this interval, or `Integer.MAX_VALUE`.
  - **See also**
    - `Interval.getDuration()` (in 2.5.5, page 27)

- **subSet**
  `public Interval` **subSet**`( Instant` **fromInstant**`, Instant` **toInstant** `)`
  - **Description**
    Return a new interval that contains all instants of this interval greater than or equals to `fromInstant` and strictly less than `toInstant`.
  - **Parameters**
    - `fromInstant` – starting instant (inclusive) of the new interval.
    - `toInstant` – ending instant (exclusive) of the new interval.
  - **Returns** – a new subinterval of this interval
  - **Throws**
    - `java.lang.NullPointerException` – if `fromInstant` or `toInstant` are `null`.
    - `java.lang.IllegalArgumentException` – if `fromInstant` or `toInstant` are not in the interval.

* `java.lang.IllegalArgumentException` – if `toInstant` is greater than `fromInstant`.

- **tailSet**
  `public Interval` **tailSet**`( Instant` **fromInstant** `)`
  - **Description**
    Return a new interval that contains all instants of this interval greater than or equals to `fromInstant`.
  - **Parameters**
    * `fromInstant` – starting instant (inclusive) of the new interval.
  - **Returns** – new interval that contains all instants of this interval greater than or equals to `fromInstant`.
  - **Throws**
    * `java.lang.NullPointerException` – if `fromInstant` is `null`.
    * `java.lang.IllegalArgumentException` – if `fromInstant` is not in the interval.
- **toArray**
  `public java.lang.Object[]` **toArray**`( )`
  - **Description**
    Returns an array containing all of the instants in this interval. Obeys the general contract of the `Collection.toArray` method.
    As the interval may contain a lot of instants, the returned array may be very big, so this method should be used very carefully.
  - **Returns** – an array containing the instants of this interval.
  - **Throws**
    * `java.lang.OutOfMemoryError` – if the interval is too big.
- **toArray**
  `public java.lang.Object[]` **toArray**`( java.lang.Object[] a )`
  - **Description**
    Returns an array containing all of the instants in this interval.
    As the interval may contain a lot of instants, the returned array may be very big, so this method should be used very carefully.
  - **Returns** – an array containing the instants of this interval.
  - **Throws**
    * `java.lang.OutOfMemoryError` – if the interval is too big.
- **toString**
  `public java.lang.String` **toString**`( )`
  - **Description**
    Convert the interval in a `String`.
  - **Returns** – the `String` describing the interval.

2.6 Class Month

The 12 months of the year.

Although it is specific to a GregorianChronology, this enum is provided to promote its usage for other chronologies that may need it (exemple: Julian Chronology).

2.6.1 Declaration

public final class Month
extends java.lang.Enum

2.6.2 Fields

- public static final Month **JANUARY**
- public static final Month **FEBRUARY**
- public static final Month **MARCH**
- public static final Month **APRIL**
- public static final Month **MAY**
- public static final Month **JUNE**
- public static final Month **JULY**
- public static final Month **AUGUST**
- public static final Month **SEPTEMBER**
- public static final Month **OCTOBER**
- public static final Month **NOVEMBER**
- public static final Month **DECEMBER**

2.6.3 Methods

- **fromValue**
  `public static Month` **fromValue**`( int` **i** `)`
- **getValue**
  `public int` **getValue**`( )`
- **valueOf**
  `public static Month` **valueOf**`( java.lang.String` **name** `)`
- **values**
  `public static final Month[]` **values**`( )`

2.6.4 Members inherited from class `java.lang.Enum`

- `protected final Object` **clone**`( ) throws CloneNotSupportedException`
- `public final int` **compareTo**`( Enum` **arg0** `)`
- `public final boolean` **equals**`( Object` **arg0** `)`
- `public final Class` **getDeclaringClass**`( )`
- `public final int` **hashCode**`( )`
- `public final String` **name**`( )`
- `public final int` **ordinal**`( )`
- `public String` **toString**`( )`
- `public static Enum` **valueOf**`( Class` **arg0**`, String` **arg0** `)`

2.7 Class Time

`Time` is a representation of a piece of time. This is the main class of this API. This class provides a good replacement for `java.util.Date` . But its usage is larger than simply a date, as it can represent thinks like:

- a normal date. Example: the 20th of October 2005
- a Year. Example: 2005
- a Month of Year . Example: December 2005
- a Week. Example: Week 47 of year 2005
- a Hour/Minutes: Example: 13:10

Indeed, `Time` are defined using a `TimeMask` which is the format of the time. The `Time` is just a map whose keys are the `TimeField`, so it implements `SortedMap<TimeField, Object>`.

It is possible to merge several `Time` objects in one. For example, you can have in one side `Time` objects that represents a day (like the 20th of October 2005). In one other side, `Time` objects that represents hour and minutes (for example 9:00). If you want to merge those two `Time` objects to speak about the 20th of October 2005 9:00, simply use the appropriate constructor .

This class is immutable, so it can itself be used as key in `java.util.Map`. It is also *thread-safe*.

2.7.1 Declaration

public final class Time
extends java.lang.Object
implements java.lang.Comparable, java.util.SortedMap

2.7.2 Constructors

- **Time**
  `public` **Time(** `java.util.Map` **map)**
  - **Description**
    Create new `Time` from a map of TimeField.
  - **Parameters**
    - `map` – the map whose keys are `TimeField` and values the values of the TimeFields.
- **Time**
  `public` **Time(** `Time[]` **otherTimes)**
  - **Description**
    Create a new `Time` object by merging existing other `Time` objects. The new `Time` objects contains all keys and values of all other `Time` objects.
  - **Throws**
    - `java.lang.IllegalArgumentException` – if two or more `Time` objects has the same `TimeField` with different values.
- **Time**
  `public` **Time(** `TimeMask` **mask,** `java.lang.Object[]` **values)**
  - **Description**
    Create a new `Time` from a `TimeMask` and some values. This is the main constructor for `Time`. The order of `values` should be the same as the order in the `mask`.

- **Parameters**
  - `mask` – the `TimeMask` which defines the format
  - `values` – all values of the TimeFields contained in the `TimeMask`.
- **Throws**
  - `java.lang.IllegalArgumentException` – if `mask` and `values` do not have the same size.
  - `java.lang.IllegalArgumentException` – if a value is incompatible with the `mask`.

2.7.3 Methods

- **clear**
  `public void` **clear**`( )`
  - **Description**
    As `TimeField` are immutable, throws a `UnsupportedOperationException`.
- **comparator**
  `public java.util.Comparator` **comparator**`( )`
  - **Description**
    As the *naturel order* if `TimeField` is used, returns `null`. `TimeField` are indeed sorted in this `Time` object.
  - **Returns** – `null`.
- **compareTo**
  `public int` **compareTo**`( Time` **other** `)`
  - **Description**
    Compares this `Time` with the specified `Time` for order. The comparaison is made with the *most significant* TimeFields of each `Time`. If values of both the *most significant* TimeField of each `Time` are the same, the comparaison is made on the next *most significant* TimeField, until a difference is founded. If all TimeFields of each `Time` object have the same values, a zero is return (and this is consistent with equals).
  - **Returns** – a negative integer, zero, or a positive integer as this `Time` is less than, equal to, or greater than the specified `Time`.
  - **Throws**
    - `java.lang.IllegalArgumentException` – if both `Time` objects do not have the same `TimeMask`.
- **containsKey**
  `public boolean` **containsKey**`( java.lang.Object` **key** `)`
  - **Description**
    Returns `true` if this `Time` object contains a mapping for the specified key.
  - **Returns** – `true` if this `Time` object contains a mapping for the specified key.
  - **Throws**
    - `java.lang.NullPointerException` – if `value` is `null`.
    - `java.lang.ClassCastException` – if `value` is not a `TimeField`.
- **containsValue**
  `public boolean` **containsValue**`( java.lang.Object` **value** `)`

- **Description**
  Returns `true` if this `Time` object maps one or more keys to the specified value.
- **Returns** – `true` if this `Time` object maps one or more keys to the specified value.
- **Throws**
  * `java.lang.NullPointerException` – if `value` is `null`.

- **entrySet**
  `public java.util.Set` **entrySet**`( )`
  - **Description**
    Returns a set view of the mappings contained in this `Time` object. The set's iterator returns the mappings in ascending key order. Each element in the returned set is a `Map.Entry<TimeField, Object>`.
    This method is can be useful to test in an instant is part of another instant. Example:

```
Time myMonth = new Time(MONTH_MASK, 2005, Month.DECEMBER);
Time other = new Time(...);
boolean test = other.entrySet().containsAll(myMonth.entrySet());
```

  - **Returns** – a set view of the mappings contained in this `Time` object.

- **equals**
  `public boolean` **equals**`( java.lang.Object` **o** `)`
  - **Description**
    Compare two `Time` for equality. The result is `true` if and only if the argument is not `null` and is a `Time` object that represents the same time as this object.
  - **Returns** – `true` if the objects are the same; false otherwise.

- **firstKey**
  `public TimeField` **firstKey**`( )`
  - **Description**
    Return the most significant `TimeField` in this `Time` object.
  - **Returns** – the most significant `TimeField` in this `Time` object.

- **get**
  `public java.lang.Object` **get**`( java.lang.Object` **field** `)`
  - **Description**
    Return the value of a `TimeField` of this `Time`.
    You should consider the `get(TimeField<V>)` methods instead, as it avoid casting.
  - **Returns** – the value of the `TimeField` provided.
  - **Throws**
    * `java.lang.NullPointerException` – if `field` is `null`.
    * `java.lang.ClassCastException` – if `field` is not a `TimeField`.
    * `java.lang.IllegalArgumentException` – if the `field` is not in this `Time` object keys'.
  - **See also**
    * `Time.get(TimeField)` (in 2.7.3, page 35)

- **get**
  `public java.lang.Object` **get**`( TimeField` **field** `)`
  - **Description**
    Return the value of a `TimeField` of this `Time`.
  - **Returns** – the value of the `TimeField` provided.
  - **Throws**
    * `java.lang.NullPointerException` – if `field` is `null`.
    * `java.lang.IllegalArgumentException` – if the `field` is not in this `Time` object keys'.
- **getTimeMask**
  `public TimeMask` **getTimeMask**`( )`
  - **Description**
    Return the format of this `Time` object.
  - **Returns** – the `TimeMask` of this `Time` object.
- **headMap**
  `public Time` **headMap**`( TimeField` **toKey** `)`
  - **Description**
    Return a new `Time` object that contains all `TimeField` more significant than `toKey` (excluding).
  - **Returns** – a new `Time` object that contains all `TimeField` more significant than `toKey` (excluding).
- **isEmpty**
  `public boolean` **isEmpty**`( )`
  - **Description**
    Returns `true` if this `Time` object contains no key-value mappings.
  - **Returns** – `true` if this `Time` object contains no key-value mappings.
- **keySet**
  `public java.util.Set` **keySet**`( )`
  - **Description**
    Returns a Set view of the keys contained in this `Time` object. The set's iterator will return the keys in ascending order.
  - **Returns** – a set view of the keys contained in this `Time` object.
- **lastKey**
  `public TimeField` **lastKey**`( )`
  - **Description**
    Return the less significant `TimeField` in this `Time` object.
  - **Returns** – the less significant `TimeField` in this `Time` object.
- **put**
  `public java.lang.Comparable` **put**`( TimeField` **key**`, java.lang.Object` **value** `)`
  - **Description**
    As `TimeField` are immutable, throws a `UnsupportedOperationException`.

- **putAll**
  `public void` **putAll**`( java.util.Map` **t** `)`
  - **Description**
    As `TimeField` are immutable, throws a `UnsupportedOperationException`.
- **remove**
  `public java.lang.Comparable` **remove**`( java.lang.Object` **key** `)`
  - **Description**
    As `TimeField` are immutable, throws a `UnsupportedOperationException`.
- **sameTimeMask**
  `public boolean` **sameTimeMask**`( Time` **other** `)`
  - **Description**
    Test if two `Time` objects use the same `TimeMask`.
  - **Parameters**
    * `other` – the `Time` to test with
  - **Returns** – `true` if this `Time` object has the same `TimeMask` than `other`.
- **sameTimeMask**
  `public boolean` **sameTimeMask**`( TimeMask` **mask** `)`
  - **Description**
    Test if this `Time` object uses a certain `TimeMask`.
  - **Parameters**
    * `mask` – the `TimeMask` to be tested
  - **Returns** – `true` if `mask` is the `TimeMask` of this `Time` object
- **size**
  `public int` **size**`( )`
  - **Description**
    Return the number of `TimeField` in this `Time` object.
  - **Returns** – the number of `TimeField` in this `Time` object.
- **subMap**
  `public Time` **subMap**`( java.util.Collection` **fields** `)`
  - **Description**
    Return a new `Time` object that contains only some `TimeField` of this `Time` object.
  - **Parameters**
    * `fields` – the `TimeField` to be kept.
  - **Returns** – a new `Time` object that contains only some `TimeField`
- **subMap**
  `public Time` **subMap**`( TimeField` **fromKey**`, TimeField` **toKey** `)`
  - **Description**
    Return a new `Time` object that contains all `TimeField` less significant than `fromKey` (including) and more significant than `toKey` (excluding).

  - **Returns** – a new `Time` object that contains all `TimeField` less significant than `fromKey` (including) and more significant than `toKey` (excluding).

- **tailMap**
  `public Time` **tailMap**`( TimeField` **fromKey** `)`
  - **Description**
    Return a new `Time` object that contains all `TimeField` less significant than `fromKey` (including).
  - **Returns** – a new `Time` object that contains all `TimeField` less significant than `fromKey` (including).

- **toString**
  `public java.lang.String` **toString**`( )`
  - **Description**
    Convert this `Time` object in a string.
  - **Returns** – a `String` describing this object.

- **update**
  `public Time` **update**`( TimeField` **field**`, java.lang.Object` **value** `)`
  - **Description**
    Create a new `Time` by updating or adding a new `TimeField` in this `Time` object.
  - **Returns** – a new `Time` that contains all `TimeField` of this `Time` object and the `field` provided.
  - **Throws**
    * `java.lang.IllegalArgumentException` – if `value` is not a valid value for `field`

- **values**
  `public java.util.Collection` **values**`( )`
  - **Description**
    Returns a collection view of the values contained in this `Time` object.
    The collection's iterator will return the values in the order that their corresponding keys appear in this `Time` object.
  - **Returns** – a collection view of the values contained in this `Time` object.

2.8 Class TimeField

Implements a field used in time representation.

A TimeField can be used by one or several different Chronologies. For example, Julian and Gregorian chronologies will likely share the same TimeFields. But Chinese chronology will have its own TimeField, although it may use the same TimeField for hour, minutes, second and millisecond.

`TimeField` objects are provided by Chronologies, and the end user should not create new ones.

A `TimeField` can have several values. Thoses value are defined with the `TimeField` itself, at TimeField creation. For example,

- for Month : JANUARY, FEBRUARY, MARCH [...], DECEMBER
- for Hour : 0, 1, [...], 23

The `TimeField` object defines also an order for its possible values, and that's why it implements `Comparable<V>`. There are two options:

- Either a `java.util.Comparator` is provided at `TimeField` creation, and this Comparator is used to achieve comparaisons between values.
- Or no specific `java.util.Comparator` is used, and the *natural order* of values are used to achieve comparaisons. (so values must implement `Comparable` in this case).

TimeField have a *typical duration* which is exprimed in milliseconds. The *typical duration* of a TimeField is the mean, or the most common duration of this TimeField. More formaly, this *typical duration* is the variation of time implied by the minimal change of this TimeField values.

Here some examples of *typical duration*:

- for Month: 30 days [2 592 000 000 milliseconds] (some month last 31 days, some 30 days, some 28 or even 29), so 30 days is a good *typical duration*.
- for Hour : 3 600 000 milliseconds, as an hour lasts 3600 seconds.
- for Days : 24 hours [86 400 000 milliseconds], as almost everyday lasts 24 hours, except with Day Light Saving, where some days last 23 hours, and some 25 hours.

The *typical duration* is only used for sorting the TimeFields together. TimeField have indeed a natural order.

TimeField are first compared by their *typical duration*. The greater the *typical duration*, the smaller the TimeField. So a TimeField like Month is before a TimeField like Days, which is before Hour. This allows to sort TimeFields with most significant TimeFields in first position. Two `TimeFields` objects with the same *typical duration* are sorted lexicographically using their name. It is suggested that a Chronology does *not* provide such TimeFields with the same *typical duration*.

As `equals()` method is not redefined, the natural order is *not* consistent with `equals()`: two `TimeField` defined by two different chronologies with the same name and duration will be equals according to `compareTo()` but not according to `equals()`. This should not be a problem as such `TimeField` should not be used together in the same `TimeMask`.

2.8.1 See also

- `TimeMask` (in 2.9, page 40)
- `Chronology` (in 2.1, page 4)

2.8.2 Declaration

public final class TimeField
extends java.lang.Object
implements java.lang.Comparable, java.util.Comparator

2.8.3 Constructors

- **TimeField**
  `public` **TimeField**`( java.lang.String` **name**`, long` **typicalDuration**`,`
  `java.util.Collection` **possibleValues**`, java.util.Comparator` **comparator** `)`

  - **Description**

    Create a new `TimeField`.

    End user should not create new `TimeField`. This constructor is provided for new `Chronology` implementations.

    If values of this `TimeField` object are not `Comparable`, the order of the `possibleValues` collection is used to achieve comparaisons between values. (this collections is assumed ordered by ascending order).

The provided `possibleValues` set should not be changed after `TimeField` creation (with `add` or `remove` methods for example), as this implementation keeps a reference to this set for testing future acceptable values.

- **Parameters**
  - `name` – the name of the new `TimeField`
  - `typicalDuration` – the typical duration in millisecond of the new `TimeField`
  - `possibleValues` – all possible values of the new `TimeField`
  - `comparator` – `null` if the *natural order* of `possibleValues` is used, otherwise a comparator that defines the order for `possibleValues`.
- **Throws**
  - `java.lang.NullPointerException` – if `name` is null
  - `java.lang.IllegalArgumentException` – if `possibleValues` is empty.
- **See also**
  - `TimeField.isValueOk(Object)` (in 2.8.4, page 40)

2.8.4 Methods

- **compare**
  `public int` **compare**`( java.lang.Object` **o1**`, java.lang.Object` **o2** `)`
  - **Description**
    Compare two possibles values of a TimeField according to this TimeField rule.
    If the values are `Comparable` if no `Comparator` have been provided at `TimeField` creation, the natural order of the values is used.
    Otherwise, the `Comparator` provided at `TimeField` creation is used.
  - **Returns** – the result
  - **Throws**
    - `java.lang.IllegalArgumentException` – if `o1` or `o2` is not a legal value for this `TimeField` object.
    - `java.lang.ClassCastException` – if values are not `Comparable` and no `Comparator` have been provided

- **compareTo**
  `public int` **compareTo**`( TimeField` **other** `)`
  - **Description**
    Compare this TimeField with another one. The most significant TimeField is sorted first.
  - **Returns** – 0 if this TimeField has the same *typical duration* and name than the `other` TimeField; a value less than 0 if this TimeField is more significant than the `other` TimeField (ie has a higher *typical duration*), or if it has the same *typical duration*, but a name lexicographically smaller; a value greater than 0 if this TimeField is less significant than the `other` TimeField (ie has a lower *typical duration*), or if it has the same *typical duration*, but a name lexicographically greater;
  - **Throws**
    - `java.lang.NullPointerException` – if `other` is null.

- **getName**
  `public java.lang.String` **getName**`( )`
  - **Description**
    Return the name of this `TimeField`. The name was specified at `TimeField` creation.

  - **Returns** – the name of this `TimeField`

- **isValueOk**
  `public boolean` **isValueOk**`( java.lang.Object` **value** `)`

  - **Description**
    Test if a value is acceptable for this TimeField.
    The `value` is acceptable if is contained in the possible value set provided at `TimeField` creation.
  - **Returns** – `true` if `value` is acceptable for this TimeField; `false` otherwise.
  - **Throws**
    * `java.lang.NullPointerException` – if `value` is null.

- **toString**
  `public java.lang.String` **toString**`( )`

  - **Description**
    Convert this `TimeField` into a `String`.
  - **Returns** – a `String` describing the `TimeField`

- **values**
  `public java.util.SortedSet` **values**`( )`

  - **Description**
    Return all possible values of this TimeField.
  - **Returns** – all possible values of this TimeField.

2.9 Class TimeMask

A TimeMask is a sorted set of `TimeField` which is used to defined `Time`. It defines a format which will be used to manipulate `Time`.

2.9.1 Declaration

public final class TimeMask
extends java.lang.Object
implements java.util.SortedSet

2.9.2 Constructors

- **TimeMask**
  `public` **TimeMask**`( java.util.Collection s )`

  - **Description**
    Create a new TimeMask from several TimeField. The `TimeFields` will be sorted. It is recommanded that all `TimeField` belongs to the same chronology, although this is not mandatory. A `TimeMask` that mixes several `TimeField` from different chronologies is likely to be useless.
  - **Parameters**
    * `s` – all TimeField used by the TimeMask.
  - **Throws**
    * `java.lang.NullPointerException` – if one or more of the TimeField is `null`.

- **TimeMask**
  `public` **TimeMask**`( TimeField[] f )`
  - **Description**
    Create a new TimeMask from several TimeField. The `TimeFields` will be sorted. It is recommanded that all `TimeField` belongs to the same chronology, although this is not mandatory. A `TimeMask` that mixes several `TimeField` from different chronologies is likely to be useless.
  - **Parameters**
    * `f` – all TimeField used by the TimeMask.
  - **Throws**
    * `java.lang.NullPointerException` – if one or more of the TimeField is `null`.
- **TimeMask**
  `public` **TimeMask**`( TimeMask[]` **masks** `)`
  - **Description**
    Create a new TimeMask from several other TimeMask. All `TimeField` from every `TimeMask` are collected in the new `TimeMask`.
  - **Parameters**
    * `masks` – all TimeMask that will be merged in the new TimeMask.
  - **Throws**
    * `java.lang.NullPointerException` – if one or more of the TimeMask is `null`.
- **TimeMask**
  `public` **TimeMask**`( TimeMask` **mask**`, TimeField[] f )`
  - **Description**
    Create a new TimeMask by adding TimeField to an existing TimeMask.
  - **Parameters**
    * `mask` – the original TimeMask
    * `f` – all TimeField to add in the new TimeMask
  - **Throws**
    * `java.lang.NullPointerException` – if `mask` is `null`.
    * `java.lang.NullPointerException` – if one or more of the TimeField is `null`.

2.9.3 Methods

- **add**
  `public boolean` **add**`( TimeField` **o** `)`
  - **Description**
    As `TimeMask` are immutable, throws a `UnsupportedOperationException`.
  - **See also**
    * `TimeMask(TimeMask,TimeField[])` (in 2.9.2, page 41)
- **addAll**
  `public boolean` **addAll**`( java.util.Collection` **c** `)`
  - **Description**
    As `TimeMask` are immutable, throws a `UnsupportedOperationException`.

- **clear**
  `public void` **clear**`( )`
  - **Description**
    As `TimeMask` are immutable, throws a `UnsupportedOperationException`.
- **comparator**
  `public java.util.Comparator` **comparator**`( )`
  - **Description**
    Return `null`, as natural order of TimeField are used in TimeMask.
  - **Returns** – `null`
- **contains**
  `public boolean` **contains**`( java.lang.Object` **obj** `)`
  - **Description**
    Returns `true` if this TimeMask contains the specified element.
  - **Parameters**
    * `obj` – object to be checked for containment in this TimeMask.
- **containsAll**
  `public boolean` **containsAll**`( java.util.Collection` **c** `)`
  - **Description**
    Returns `true` if this TimeMask contains all of the elements in the specified collection.
  - **Parameters**
    * `c` – collection to be checked for containment in this TimeMask.
  - **Returns** – `true` if this TimeMask contains all of the elements in the specified collection.
- **equals**
  `public boolean` **equals**`( java.lang.Object` **anObject** `)`
  - **Description**
    Compares this TimeMask to the specified object. The result is `true` if and only if the argument is not `null` and is a TimeMask that have the same sequence of TimeField as this object.
  - **Parameters**
    * `anObject` – - the object to compare this TimeMask against.
- **first**
  `public TimeField` **first**`( )`
  - **Description**
    Return the *most significant* TimeField of this TimeMask.
  - **Returns** – the *most significant* TimeField of this TimeMask.
- **getFields**
  `public java.util.SortedSet` **getFields**`( )`
  - **Description**
    Return all TimeField contained in this TimeMask.

- **Returns** – all TimeField contained in this TimeMask.

- **hashCode**
  `public int` **hashCode**`( )`
  - **Description**
    Returns a hash code value for this object.

- **headSet**
  `public TimeMask` **headSet**`( TimeField` **toField** `)`
  - **Description**
    Build a new TimeMask with TimeField of this TimeMask which are *more significant* (exclusive) than a TimeField.
  - **Parameters**
    * `toField` – TimeField to compare with.
  - **Returns** – new TimeMask with some of the *more significant* TimeField of this TimeMask.
  - **Throws**
    * `java.lang.NullPointerException` – if `toField` is `null`.
    * `java.lang.IllegalArgumentException` – if `toField` is not in the range of this TimeMask.

- **isEmpty**
  `public boolean` **isEmpty**`( )`
  - **Description**
    Return `true` is this TimeMask contains no TimeField.
  - **Returns** – `true` is this TimeMask contains no TimeField.

- **iterator**
  `public java.util.Iterator` **iterator**`( )`
  - **Description**
    Return an iterator over TimeField contained in this TimeMask.

- **last**
  `public TimeField` **last**`( )`
  - **Description**
    Return the *less significant* TimeField of this TimeMask.
  - **Returns** – the *less significant* TimeField of this TimeMask.

- **remove**
  `public boolean` **remove**`( java.lang.Object` **o** `)`
  - **Description**
    As `TimeMask` are immutable, throws a `UnsupportedOperationException`.
  - **See also**
    * `TimeMask.remove(TimeField[])` (in 2.9.3, page 43)

- **remove**
  `public TimeMask` **remove**`( TimeField[]` **f** `)`

  - **Description**
    Build a new TimeMask by removing some TimeField of this TimeMask.
  - **Parameters**
    * `f` – all TimeField to remove from this TimeField.
  - **Returns** – a new TimeMask that contains less TimeField than this TimeMask.
  - **Throws**
    * `java.lang.IllegalArgumentException` – if one or more of the TimeField are not contained in the TimeMask.

- **removeAll**
  `public boolean` **removeAll**`( java.util.Collection` **c** `)`
  - **Description**
    As `TimeMask` are immutable, throws a `UnsupportedOperationException`.

- **retainAll**
  `public boolean` **retainAll**`( java.util.Collection` **c** `)`
  - **Description**
    As `TimeMask` are immutable, throws a `UnsupportedOperationException`.

- **size**
  `public int` **size**`( )`
  - **Description**
    Return the number of TimeFields in the TimeMask.
  - **Returns** – number of TimeFields in the TimeMask.

- **subSet**
  `public TimeMask` **subSet**`( TimeField` **fromField**`, TimeField` **toField** `)`
  - **Description**
    Build a new TimeMask with some TimeField of this TimeMask. Only TimeField that are:
    * *less significant* than `fromField` (inclusive)
    * *more significant* than `toField` (exclusive)

    will be in the returned TimeMask.
  - **Parameters**
    * `fromField` – lower limit of the TimeField.
    * `toField` – uppder limit TimeField.
  - **Returns** – new TimeMask with some of the TimeField of this TimeMask.
  - **Throws**
    * `java.lang.NullPointerException` – if `fromField` or `toField` is `null`.
    * `java.lang.IllegalArgumentException` – if `fromField` or `toField` is not in the range of this TimeMask.
    * `java.lang.IllegalArgumentException` – if `fromField` is greater than `toField`.

- **tailSet**
  `public TimeMask` **tailSet**`( TimeField` **fromField** `)`
  - **Description**
    Build a new TimeMask with TimeField of this TimeMask which are *less significant* (inclusive) than a TimeField.

- **Parameters**
  - `fromField` – TimeField to compare with.
- **Returns** – new TimeMask with some of the *less significant* TimeField of this TimeMask.
- **Throws**
  - `java.lang.NullPointerException` – if `fromField` is `null`.
  - `java.lang.IllegalArgumentException` – if `fromField` is not in the range of this TimeMask.

- **toArray**
  `public java.lang.Object[] toArray( )`
- **toArray**
  `public java.lang.Object[] toArray( java.lang.Object[] a )`
- **toString**
  `public java.lang.String toString( )`
  - **Description**
    Convert this TimeMask to a String.
  - **Returns** – a String describing this TimeMask

2.10 Example of code

```
1  package time;
2
3  import java.util.*;
4
5  /**
6   * This is only a simple example of API time usage.
7   */
8  class Sample1 {
9
10     public static void main(String arg[]) {
11
12         // Get the default GregorianChronology
13         GregorianChronology gc = new GregorianChronology();
14
15         // For each Time, we have a comment.
16         // Our book is just a Map, where the key is the Time, and the value the comment.
17         Map<Time, String> book = new TreeMap<Time, String>();
18
19
20         //-------------------------------------------------------------
21
22         // Build some common TimeMask, taking advantage of varargs
23
24         // This mask define format like YYYY/MM/DD
25         TimeMask yearMonthDay = new TimeMask(gc.YEAR, gc.MONTH, gc.DAY_OF_MONTH);
26
27         // This mask defines format like YYYY WW, where WW is the week number
28         TimeMask yearWeek = new TimeMask(gc.YEAR, gc.WEEK_OF_YEAR);
29
```

```
30          // This mask defines format like YYYY WW MM/DD, where WW is the week number
31          TimeMask yearMonthWeekDay =
32              new TimeMask(gc.YEAR, gc.MONTH, gc.WEEK_OF_YEAR, gc.DAY_OF_MONTH);
33
34          // We could also write:
35          yearMonthWeekDay = new TimeMask(yearMonthDay, yearWeek);
36
37          // This mask defines only hour & minute
38          TimeMask hourMinute = new TimeMask(gc.HOUR_OF_DAY, gc.MINUTE);
39
40          //-------------------------------------------------------------
41
42          // Let's work on the Week 49 of Year 2005
43          // (You may notice that we use varargs and autoboxing)
44          Time week = new Time(yearWeek, 2005, 49);
45
46
47          // Imagine we want to do something each day of this week
48          // So we iterate on the weekday of this week:
49          for (Time day : gc.split(week, gc.DAY_OF_WEEK)) {
50
51              DayOfWeek dayWeek = day.get(gc.DAY_OF_WEEK); //Monday, Tuesday...
52
53              // Let's retrieve all information (month, day_of_month...)
54              Time full = gc.getTime(gc.getInterval(day), yearMonthWeekDay);
55
56              // No casting!
57              Month m = full.get(gc.MONTH);
58
59              // Auto-unboxing!
60              int dayOfMonth = full.get(gc.DAY_OF_MONTH);
61
62              // The 9:15 hour
63              Time h = new Time(hourMinute, 9, 15);
64
65              // We complete the day information with the hour & minute
66              Time dayAndHour = new Time(full, h);
67
68              // We update our timebook
69              book.put(dayAndHour, "It is 9:15.");
70          }
71
72
73          // Maybe we can print out timebook, in chronological order
74          for (Map.Entry<Time, String> entry : book.entrySet()) {
75              Time t = entry.getKey();
76              System.out.println("At " + t +
77                  " the current entry is "+entry.getValue());
78          }
79
80          //-------------------------------------------------------------
```

```
81
82          // Let's work on the 12/12/2005
83          Time day1 = new Time(yearMonthDay, 2005, Month.DECEMBER, 12);
84
85          // Imagine we have something to do each hour of this day
86          for (Time hourly : gc.split(day1, gc.HOUR_OF_DAY)) {
87              System.out.println("It is "+hourly);
88
89              // We want to use legacy code: we convert hourly in a
90              // java.util.Date.
91              // The usual convention is to use the beginning of the hour
92              // (Example: 12/12/2005 12:00:00.000)
93              Date date = gc.getInterval(hourly).getStart().asImmutableDate();
94          }
95
96          // By the way, how many days in the month?
97          // We get the month
98          Time monthOfDay1 = day1.subMap(new TimeMask(gc.YEAR, gc.MONTH));
99
100         // Or we can use SortepMap methods:
101         monthOfDay1 = day1.headMap(gc.DAY_OF_MONTH);
102
103         // The computation is simple:
104         int numberOfDayInMonth = gc.split(monthOfDay1, gc.DAY_OF_MONTH).size();
105
106         //-------------------------------------------------------------
107
108         // Last example: compute the number of day between the 02/02/1970 and
109         // the 03/03/2010.
110         Time dayStart = new Time(yearMonthDay, 1970, Month.FEBRUARY, 2);
111         Time dayEnd = new Time(yearMonthDay, 2010, Month.MARCH, 3);
112
113         SortedSet<Time> allDaysSet = gc.asSortedSet(yearMonthDay);
114
115         // And the result is:
116         int nbDays = allDaysSet.subSet(dayStart, dayEnd).size();
117
118         // And what's the 1000st day after the 02/02/1970?
119         List<Time> allDaysList = gc.asList(yearMonthDay);
120         int idx = allDaysList.indexOf(dayStart);
121
122         // Here is the searched day:
123         Time dayStart1000 = allDaysList.get(idx+1000);
124     }
125 }
```