**Java Core Fundamentals: Math, Randomization, BigDecimal, and Data/Time**

**Java.uti.Math**

It’s like a calculator. It has functions you’d find on a basic calculator, as well as a few scientific methods.

**Use Math.random or Random’s methods**

Int random = (int) (Math.random() \* (26)) + 65;

The Math.random method uses an instance of the Random class and invokes the nextDouble method on that class.

The first time you call Math.random, a single new instance of java.util.Random is created and use for all subsequent calls.

**Some floating point number basics**

Precision defines the number of digits in a decimal number.

Precision includes digits to both the left and the right of the decimal point.

Scale is the number of digits to the right of the decimal point in a number

|  |  |  |
| --- | --- | --- |
| Examples | Precision | Scale |
| 15.456 | 5 | 3 |
| 8 | 1 | 0 |
| 100000.000001 | 12 | 6 |
| .123 | 3 | 3 |

**BigDecimal**

* The BigDecimal class stores a floating-point number in two integer fields.
* The first field holds an unscaled value, with a type of BigInteger, another class in the java.math package, that can store numbers, bigger than even long values.
* The second field is the scale, which can be positive or negative.
* A positive or 0 scale defines how many digits in the unscaled value, are after the decimal point.
* You can use a negative scale as well, which means the unscaled value is multiplied by 10 to the power of the negation of the scale.

Examples

|  |  |  |  |
| --- | --- | --- | --- |
| Examples | Unscaled Value | Precision | Scale |
| 15.456 | 15456 | 5 | 3 |
| 8 | 8 | 1 | 0 |
| 100000.000001 | 100000000001 | 12 | 6 |
| .123 | 123 | 3 | 3 |

**Java.time**

A screenshot of a computer program

Description automatically generated

The OffsetDateTime and ZonedDateTime classes are used when you need to measure or store dates and time, against the universal standard.

**java.time related packages**

* Java has other packages under the java.time umbrella, as shown on the image below.
* These are java.time.temporal, and java.time.format.
* In addition, there are the java.time.zone and java.time.chrono packages.

A screenshot of a computer program

Description automatically generated

**Java.time.temporal**

* The java.time.temporal package contains important interfaces, that the java.time classes implement.
* These include the Temporal, and TemporalAccessor interfaces, that describe a uniform way to read from, or to write to, a date time object
* The TemporalAdjuster, TemporalAmount, TemporalField and TemporalUnit interfaces are used often as method parameters; to select the specific information you want from a date time object.
* There are two enums in this package, ChronoField and ChronoUnit.
* The TemporalAdjusters class is a helper class to return specific implementations of TemporalAdjuster, which can give you helpful dates such as first day of the month, or last day of year etc.

A screenshot of a computer

Description automatically generated

**Java.time.format**

* Using the “%t” specifier, in a formatted String we can format date time.
* The java.time.format package gives us a lot more options, as well as support for localization.
* There are enums to support dates and time in defined styles, called Full, Long, Medium and Short.

A screenshot of a computer program

Description automatically generated

**LocalDate, LocalTime and LocalDateTime**

The most common classes for Date and Time, when we don’t need to include time zone data, these are LocalDate, LocalTime and LocalDateTIme.

Each implement both Temporal and the TemporalAccessor interface, and the methods on those interfaces

These include the get and range methods from TemporalAccessor.

From Temporal, there are methods to add or subtract units of time from the objects. These are the Plus and Minus methods.

Field values can be directly set on a returned copy, using the with methods

We also have methods on LocalDate, that in general, are methods available in some form, on any of the temporal objects.

These have prefixes such as: at, get, and is

A screenshot of a computer program

Description automatically generated

**Creating instances of Temporal Implementations**

There are several static methods, which return a new instance of a temporal class.

There is the **now** method, that gives you a representation of the current moment, for the class you specify.

You can create instances, using any of the several overloaded versions that exist for the **of** factory method.

Alternately, you can use the **parse** method with a character sequence, usually a String, that's formatted in a predetermined way, or you can pass in a defined format.

The methods shown on this slide are purposely shown without too much detail.

You can use similarly named methods on any of the temporal implementations, with varying parameters and return types.

A blue and white box with text

Description automatically generated

**Temporal Instances Are Immutable**

All temporal instances of classes, in the java.time package, are immutable and thread safe.

A new instance is returned from methods that write to a temporal instance, such as the plus or minus methods, for example.

You'll need to assign the result of these methods to a variable.

The original instance won't be modified.

**Methods of Temporal Implementations**

In addition to the methods on the Temporal and TemporalAccessor interfaces, most implementations have methods prefixed with at, get, and is, as well as a format method.

The **at** methods allow you to combine temporal instances. As an example, a time is combined with the LocalDate for both the atStartOfDay and atTime methods and return LocalDateTime instances.

The Date and Time instances implement Comparable, so each has a compareTo method.

A screenshot of a computer program

Description automatically generated

**Methods of Temporal Implementations**

The **get** methods are specific to the class, so for LocalDate, you'd have date related getters, like **getYear**, **getMonth**, **getDayofWeek**, and so on.

For LocalTime, these would be **getHour**, **getMinute**, etc.

Each class supports **isAfter**, **isBefore** and **isEqual**, so you can compare units of date or time.

And there's a format method, to output a formatted date.

A screenshot of a computer program

Description automatically generated

**LocalDate, LocalTime, and LocalDateTime classes**

Java is internally storing the date fields, and the time fields, as separate numeric fields.

These fields can be retrieved, with or without context, of the other fields.

For example, you can get day of the month, or day of the year.

LocalDate uses an int for year, and a short for month and day.

LocalTime uses bytes for the hour, minute, and second, and an int for nanoseconds.

LocalDateTime has two fields, a date, with a type of LocalDate, and a time, with a type of LocalTime

A screenshot of a computer program

Description automatically generated

**The LocalDate Class**

LocalDate is a class for storing and managing a date, with a year, month, and day, without reference to a specific time zone.

This kind of date might be used for an anniversary date, a birth date, or a special holiday like Cinco de Mayo (the 5th of May), or Thanksgiving Day.

This class doesn't have a clock time component.

A blue and white box with text

Description automatically generated

**The LocalTime Class**

The LocalTime class provides a description of the local time as seen on a wall clock.

It contains neither a date or a time zone.

Time is represented as hour-minute-second, with nanosecond precision, if it's available.

A screenshot of a computer

Description automatically generated

***ChronoField*** implements temporalField - TemporalField is an interface

for a class, that implements a date time field.

**TemporalField vs. Temporal Unit**

A TemporalFiled represents a specific field within a date-time, such as MONTH\_OF\_YEAR, DAY\_OF\_WEEK, or HOUR\_OF\_DAY. It defines fields we commonly think about in terms of date-time components (e.g., the month part of a date or the hour part of a time).

In contrast, a TemporalUnit represents a unit or duration of time, such as YEARS, MONTHS, DAYS, or MINUTES. Rather than representing a part of a date or time, it represents the amount of time you might use to measure intervals between date-times.

A screenshot of a computer program

Description automatically generated

The get and with methods take a TemporalField as a parameter. The plus, minus and until methods, take a TemporalUnit.

**A Date-Based Timeline**

The following is the timeline from January 1, 2020, through the end of April of the same year.

A point on a timeline, shown here as March 20, at 4:55pm is called an **Instant** in time.

An interval (or span of time) on a date time line is called a **Period**

It is often represented by elapsed time in date units such as years, months or days

A blue line with orange dots

Description automatically generated

**An Hourly Timeline**

A blue and orange rectangular object with text

Description automatically generated

The mentioned above is a timeline for a single 24- hour day.

This timeline can have an Instant as well, as shown at 16:30 which represents 4:30pm, on this timeline

This timeline is also showing a span of time, from 6:40 am until 1:20pm

When the interval is time based (in terms of the units being in hours, minutes or seconds), this is called a Duration

In the above-mentioned timeline, the Duration lasted 6 hours and 40 minutes.

Instant

You can represent an event on a timeline as an Instant.

Java provides the Instant class for this.

It has two fields, one for seconds, a long, and one for nano seconds, an int.

A blue and white rectangular object with white text

Description automatically generated

**Instant**

What does seconds mean?

How can we store a moment in time as seconds?

Does not it need some context, like a starting point int time, from which to measure these seconds, or a defined timeline?

These seconds are called epoch seconds, and the epoch is the starting point for many such points in time.

Instead of having to specify this point in time, every time you want a time stamp or instant of time, many software languages used a specific date and time called **Epoch time.**

**Whats the significance of the EPOCH Time?**

00:00:00 UTC, Thursday 1st January 1970 is called alternately Unix Time, POSIX Time, Epoch Time or Unix Epoch Time.

Computer systems have different epoch times, but many software languages use this particular date and time, including Java, as the predetermined start of time.

It is used to create time stamps whose meaning, as seconds, can be understood.

If we ever want to see it, or use it, it’s a constant named EPOCH, on the LocalDate and Instant classes.

UTC is a symbol that stands for Coordinated Universal Time, which coordinates with Greenwich Mean Time (GMT)

**Duration and Period**

There are significant differences in Instant class and Duration class.

First, Duration and Period do not implement Temporal or TemproalAccessor.

They instead implement TemporalAmount as shown.

This means these classes are something quite different from a unit of date or time.

They are amounts of time

Both represent elapsed time between units of time (or dates)

A diagram of a period

Description automatically generated

**Greenwich Mean Time**

Maybe you'll remember from an old geography class, that the earth is drawn with imaginary lines on it, to help uniformly describe locations on earth.

The lines drawn from the north pole to the south pole are called meridians.

These could be drawn anywhere, but a **Prime Meridian** was historically agreed upon, as a starting point or the zero measurement.

This is a reference point for all other measurements.

The time in Greenwich, was solar time, based on the position of the sun in the sky.

When atomic clocks were introduced, they were able to provide more precise time than Greenwich Mean Time (GMT).

In 1972, GMT was superseded by UTC, which stands for Coordinated Universal Time, and is based on atomic time.

The differences between GMT and UTC can differ by up to 0.9 seconds in a day.

For this reason, GMT and UTC are often used interchangeably, when you don't need great precision.

**Time Zone**

A Time Zone consists of two parts, a UUTC offset, and optionally, information about Daylight Savings Time.

Java derives it’s time zone data from three sources:

* The Internet Assigned Numbers Authority (IANA)’s Time Zone Database (TZDB) is the default, and takes precedence over the others
* IATA (the airline industry body)
* Microsoft

Two helpful links are displayed on this slide.

The first site gives you a list of the time zone identifiers.

<https://twiki.org/cgi-bin/xtra/tzdatepick.html>

The second gives you a list of day light savings rules.

<https://en.wikipedia.org/wiki/Daylight_saving_time_by_country>

Both of these sites use the IANA Time Zone database as their source.

**Before JDK 8 and the introduction of the java.time.package**

There are classes in java.util that may look attractive to use, and were used before JDk 8. These are Date, TimeZone, GregorianCalendar

Additional classes for formatting are in the java.text package:

* DateFormat
* SimpleDateFormat

The use of these classes in new code in discouraged. The immutable thread-safe classes, provide by the java.time packages, should be used, instead.

**System.currentTimeMillis**

System.currentTimeMillis returns the milliseconds since epoch time, so midnight, January 1, 1970 UTC

This time is based on the operating system

This can be used to measure elapsed times, or provide timestamps

**System.nanoTime**

System.nanoTime uses the JVM’s high resolution time source, to return nanoseconds, from an arbitrary origin time, which is not Epoch time, and may even be a time in the future.

This origin time may differ across different JVM instances

For this reason, this time can’t be used to represent real time, or wall clock time.

It should not be used a timestamp for data.

Instead, its purpose is to measure elapsed time for invocations in a single JVM instance.

**Instant**

The Instant class is designed to represent only a point in time, or a timestamp. Internally, it stores a value in seconds and nanoseconds, from the fixed epoch time of 1970-01-01Z.

As such, an Instant can’t be formatted as a date or time, without providing time-zone information. When printed, Z is used as a suffix to indicate this timestamp is in reference to UTC.

This allows for comparisons, regardless of time zones or operating systems, which is why it’s an effective timestamp.

**Locale**

Locale is an English word for a place where something happens.

It's also the name of a class in the java.util package that underpins support for both localization and internationalization.

Java has some built-in support for localization, with methods that let you pass a Locale instance to them.

A locale has five fields, a language, a country (or region), a variant, and less apparent, are script and extensions.

A blue and white squares with white text

Description automatically generated

**Java’s localization support**

Java provides localization support for dates, numbers, and currencies with no additional effort, other than defining a Locale and passing it to certain methods.

For language constructs, other than a month or a weekday specified in a date, youll need to do some additional work.

**Internationalization**

Internationalization, or I18n for short, is a method of designing your application to allow language and regional elements, with the help of locales to be plug and play.

Strings used in messages or user interface elements and images are stored externally to the application.

These can be retrieved using a locale, and then displayed for a specific user.

Java uses the ResourceBundle class to support this feature.

**The ResouceBundle class**

The RecourceBuundle class is an abstract class in the java.uutil.package

You can implement classes that extend this abstract class. You can get an instance of a ResouceBuuundle, by calling one of several static getBundle methods, on the ResourceBundle class

The latter approach is dependent on resource data, either stored in a series of files, or provided by a service.

**ResourceBundle data in a .properties file**

Data that's customized is often textual, in the form of user messages, button labels or menu items, but may contain other elements such as images or audio components.

To date, the most common method of supplying data for a ResourceBundle, is using the properties file.

This is a simple text file, containing key value pairs.

The key is a string, a name to be used when the data is requested, and the value is also a single text literal.

The properties file name includes a base name, called the bundle name and some part of the Locale identifier, and ends with the extension .properties.

Here, I'm showing an example of a base TextMessages properties file.



You'll create additional properties files, to support other languages.

You can think of a bundle as a series of files that have the same base bundle name but are differentiated by Locale specifics.

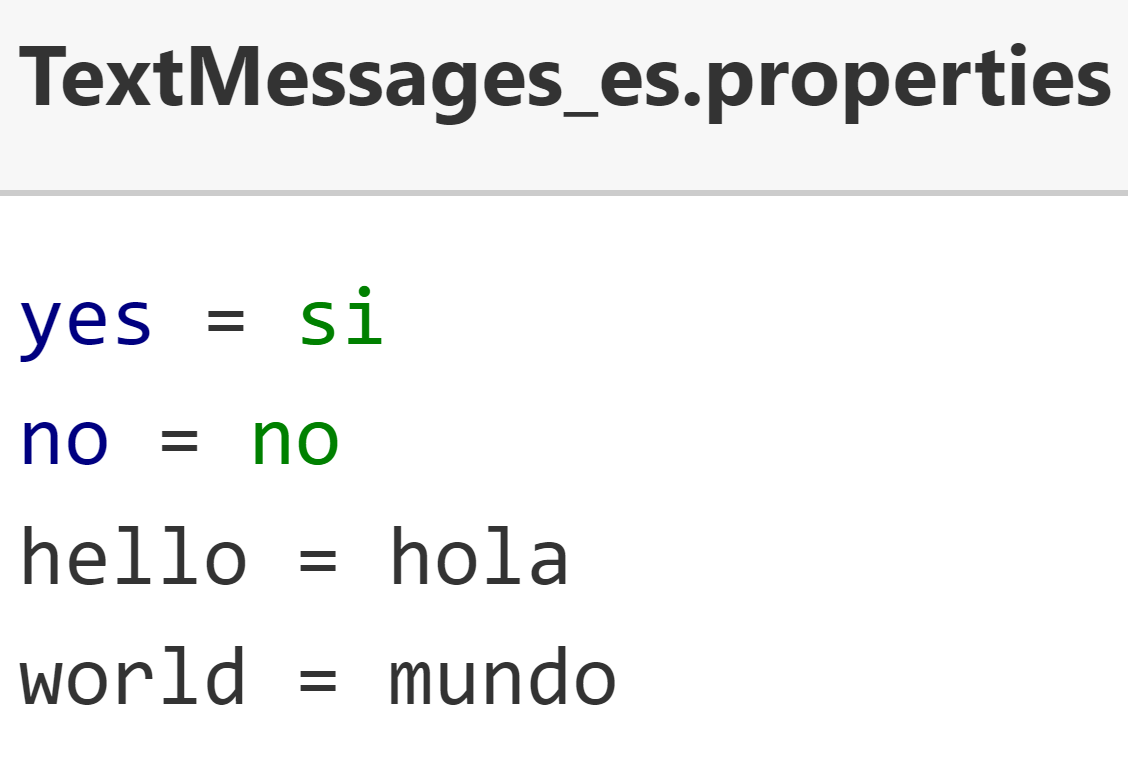
If you need to support the spanish language for example, you would create a TextMessages\_es.properties file, with spanish language text literals for yes, no, hello and so on.

I show an example of this file here.

Notice that the keys for both these properties files are in English, and are the same for both files.

Whatever language you choose for your keys is up to you, but they do need to be consistent across files, so that the key can be used to look up the value.

Of course, there are many language dialects, so you may need to also provide additional language variations for these, which can be done by including country, script and or a variant.



**Java's matching process to locate the best bundle**

Java has specific rules for searching and matching one of these properties files, to a Locale.

These rules can be found in the ResourceBundle class documentation.

Here, I include the link to this information.

[https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ResourceBundle.html#default\_behavior](https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ResourceBundle.html)

**ResourceBundle - alternatives to .properties files**

You're not limited to using properties files for your internationalization support.

You can extend the ResourceBundle class, or another related abstract class called the ListResourceBundle class. You'd have the subclasses house your data in code, or source it from another place.

You can use other file formats, such as xml, with a little extra configuration, by extending both the ResourceBundle and ResourceBundle.Control classes.

You can find instructions for this in the ResourceBundle.Control API documentation, the link is displayed here.

<https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/ResourceBundle.Control.html>

Alternately, you can make a call to a service provider.

You can also mix and match any of these methods.