## Java 8 Date/Time API - introduction

**Core Ideas**

The new API is driven by three core ideas:

* **Immutable-value classes.** One of the serious weaknesses of the existing formatters in Java is that they aren’t thread-safe. This puts the burden on developers to use them in a thread-safe manner and to think about concurrency problems in their day-to-day development of date-handling code. The new API avoids this issue by ensuring that all its core classes are immutable and represent well-defined values.
* **Domain-driven design.** The new API models its domain very precisely with classes that represent different use cases for Date and Time closely. This differs from previous Java libraries that were quite poor in that regard. For example, java.util.Date represents an instant on the timeline—a wrapper around the number of milli-seconds since the UNIX epoch—but if you call toString(), the result suggests that it has a time zone, causing confusion among developers.

This emphasis on domain-driven design offers long-term benefits around clarity and understandability, but you might need to think through your application’s domain model of dates when porting from previous APIs to Java SE 8.

* **Separation of chronologies.** The new API allows people to work with different calendaring systems in order to support the needs of users in some areas of the world, such as Japan or Thailand, that don’t necessarily follow ISO-8601. It does so without imposing additional burden on the majority of developers, who need to work only with the standard chronology.

**LocalDate and LocalTime**

The first classes you will probably encounter when using the new API are LocalDate and LocalTime. They are local in the sense that they represent date and time from the context of the observer, such as a calendar on a desk or a clock on your wall. There is also a composite class called LocalDateTime, which is a pairing of LocalDate and LocalTime.

The existing classes aren’t thread-safe, leading to potential concurrency issues for users—**not something the average developer would expect.**

Time zones, which disambiguate the contexts of different observers, are put to one side here; you should use these local classes when you don’t need that context. A desktop JavaFX application might be one of those times. These classes can even be used for representing time on a distributed system that has consistent time zones