

Intermediate Java

Daniel Hinojosa

Java Date Time API

ISO 8601 Standard

- Standard and Collaborative means of managing date and time
- Based on the cesium-133 atom atomic clock

ISO 8601 Formats

| Format | Example |
|-------------------------------|---|
| Date | 2014-01-01 |
| Combined Date and Time in UTC | 2014-07-07T07:01Z |
| Combined Date and Time in MDT | 2014-07-07T07:38:51.716-06:00 |
| Date With Week Number | 2014-W27-3 |
| Ordinal Date | 2014-188 |
| Duration | P3Y6M4DT12H30M5S |
| Finite Interval | 2014-03-01T13:00:00Z/2015-05-11T15:30:00Z |
| Finite Start with Duration | 2014-03-01T13:00:00Z/P1Y2M10DT2H30M |
| Duration with with Finite End | P1Y2M10DT2H30M/2015-05-11T15:30:00Z |

Life and Times Java

java.util.Date

- Introduced millisecond resolution
- java.util.Date
- What was wrong with it?
 - Constructors that accept year arguments require offsets from 1900, which has been a source of bugs.
 - January is represented by 0 instead of 1, also a source of bugs.
 - Date doesn't describe a date but describes a date-time combination.
 - Date's mutability makes it unsafe to use in multithreaded scenarios without external synchronization.
 - Date isn't amenable to internationalization.

Source: <http://www.javaworld.com/article/2078757/java-se/java-101-the-next-generation-it-s-time-for-a-change.html>

java.util.Calendar

- Introduced in Java 1.1
- What is wrong with it?
 - It isn't possible to format a calendar.
 - January is represented by 0 instead of 1, a source of bugs.
 - Calendar isn't type-safe; for example, you must pass an int-based constant to the `get(int field)` method. (In fairness, enums weren't available when Calendar was released.)
 - Calendar's mutability makes it unsafe to use in multithreaded scenarios without external synchronization. (The companion `java.util.TimeZone` and `java.text.DateFormat` classes share this problem.)
 - Calendar stores its state internally in two different ways — as a millisecond offset from the epoch and as a set of fields — resulting in many bugs and performance issues.

Source: <http://www.javaworld.com/article/2078757/java-se/java-101-the-next-generation-it-s-time-for-a-change.html>

Of course then there is this:

```
> new java.util.GregorianCalendar
```

```
java.util.GregorianCalendar = java.util.GregorianCalendar[time=1393764079082,areFieldsSet=true,areAllFieldsSet=true,lenient=true,zone=sun.util.calendar.ZoneInfo[id="America/New_York",offset=-18000000,dstSavings=3600000,useDaylight=true,transitions=235,lastRule=java.util.SimpleTimeZone[id=America/New_York,offset=-18000000,dstSavings=3600000,useDaylight=true,startYear=0,startMode=3,startMonth=2,startDay=8,startDayOfWeek=1,startTime=7200000,startTimeMode=0,endMode=3,endMonth=10,endDay=1,endDayOfWeek=1,endTime=7200000,endTimeMode=0]],firstDayOfWeek=1,minimalDaysInFirstWeek=1,ERA=1,YEAR=2014,MONTH=2,WEEK_OF_YEAR=10,WEEK_OF_MONTH=2,DAY_OF_MONTH=2,DAY_OF_YEAR=61,DAY_OF_WEEK=1,DAY_OF_WEEK_IN_MONTH=1,AM_PM=0,HOUR=7,HOUR_OF_DAY=7,MINUTE=41,SECOND=19,MILLISECOND=82,ZONE_OFFSET=-18000000,DST...
```

What was cool about Joda Time

- Straight-forward instantiation and methods
- UTC/ISO 8601 Based, not Gregorian Calendar
- Has support for other calendar systems if you need it (Julian, Gregorian-Julian, Coptic, Buddhist)
- Includes classes for date times, dates without times, times without dates, intervals and time periods.
- Advanced formatting
- Well documented and well tested

- Immutable!
- Months are 1 based

About the Java 8 Date Time API

- Authored by the same team as Joda Time
- Immutable & Threadsafe
- Learned from previous mistakes made in Joda Time
- There are no *constructors* (Dude what?)
- Nanosecond Resolution

The Java Date Time Packaging

- `java.time` - Base package for managing date time
- `java.time.chrono` - Package that handles alternative calendaring and chronology systems
- `java.time.format` - Package that handles formatting of dates and times
- `java.time.temporal` - Package that allows us to query dates and times

Date Time Conventions

- `of` - static factory usually validating input parameters not converting them
- `from` - static factory that converts to an instance of a target class
- `parse` - static factory that parses an input string
- `format` - uses a specified formatter to format the date
- `get` - Returns part of the state of the target object
- `is` - Queries the state of the object
- `with` - Returns a copy of the object with one element changed, this is the immutable equivalent
- `plus` - Returns a copy of the target object with the amount of time added
- `minus` - Returns a copy of the target object with the amount of time subtracted
- `to` - Converts this object to another object type
- `at` - Combines the object with another

Instant

- Single point in time
- Time since the Unix/Java Epoch `1970-01-01T00:00:00Z`
- Differs from the `java.util.Date` and `long` representation

- Contains two states:
 - `long` of seconds since the Unix Epoch
 - `int` of nano seconds within one second

That a lot of resolution!

An `Instant` can be resolved as $1.844674407 \times 10^{19}$ seconds or 584542046090 years!

Some of the basic features of **Instant**

```
Instant now = Instant.now();
System.out.println(now.getEpochSecond());
System.out.println(now.getNano());
System.out.println(Instant.parse("2014-02-20T20:21:20.432Z"));
```

Enums

Month and DayOfWeek

- The Java Date/Time API contains **enum** classes to describe our months and days
 - **Month**
 - **DayOfWeek**

Month and DayOfWeek Exemplified

```
DayOfWeek.SUNDAY
DayOfWeek.FRIDAY
```

```
Month.JANUARY
Month.JULY
Month.DECEMBER
```

ChronoUnit

- **enum** to represent a unit of time for a scalar
- implements **TemporalUnit**
- **ChronoUnit** is meant to be general enough for various calendars

ChronoUnit Exemplified

```
ChronoUnit.DAYS
ChronoUnit.CENTURIES
ChronoUnit.ERAS
ChronoUnit.MINUTES
ChronoUnit.MONTHS
ChronoUnit.SECONDS
ChronoUnit.FOREVER
```

```
Instant.now().plus(19, ChronoUnit.DAYS)
```

ChronoField

- Represents a field in a date
- Given: `2010-10-22T12:00:13` has six fields
 - The year: `2010`
 - The month: `10`
 - The day of the month: `22`
 - The hour of the day: `12`
 - The minute: `0`
 - The seconds: `13`
- implements `TemporalField`
- `ChronoField` is also meant to be general enough for various calendars

ChronoField Exemplified

```
ChronoField.MONTH_OF_YEAR  
ChronoField.DAY_OF_MONTH  
ChronoField.HOUR_OF_DAY  
ChronoField.SECOND_OF_MINUTE  
ChronoField.SECOND_OF_DAY  
ChronoField.MINUTE_OF_DAY  
ChronoField.MINUTE_OF_HOUR
```

```
Instant.now.get(ChronoField.HOUR_OF_DAY);
```

Local Dates and Times

- `LocalDate` - An ISO 8601 date representation without timezone and time
- `LocalTime` - An ISO 8601 time representation without timezone and date
- `LocalDateTime` - An ISO 8601 date and time representation without time zone

Lab: Create a `LocalDate`

Step 1: Create a new test file in the `src/test/java` folder and inside the `com.xyzcorp` package called `DatesTest`

Step 2: In `DatesTest` create a test called using `testCreateLocalDate` with the following content.

```

@Test
public void testCreateDate() {
    LocalDate february20th = LocalDate.of(2014, Month.FEBRUARY, 20);
    System.out.println(february20th);
    System.out.println(LocalDate
        .from(february20th
            .plus(15, ChronoUnit.YEARS)));
    System.out.println(LocalDate.parse("2014-11-22"));
}

```

Step 3: Run the test

LocalTime exemplified

```

LocalTime.MIDNIGHT;
LocalTime.NOON;
LocalTime.of(23, 12, 30, 500);
LocalTime.now();
LocalTime.ofSecondOfDay(11 * 60 * 60);
LocalTime.from(LocalTime.MIDNIGHT.plusHours(4));

```

LocalDateTime exemplified

```

LocalDateTime.of(2014, 2, 15, 12, 30, 50, 200);
LocalDateTime.now();
LocalDateTime.from(
    LocalDateTime.of(
        2014, 2, 15, 12, 30, 40, 500)
        .plusHours(19)));
LocalDateTime.MIN;
LocalDateTime.MAX;

```

ZonedDateTime

- Specifies a complete date and time in a particular time zone
- Contains methods that can convert from `LocalDate`, `LocalTime`, and `LocalDateTime` to `ZonedDateTime`

But first, ZoneId

- `ZoneId` represents the IANA Time Zone Entry
- <http://www.iana.org/time-zones>

- Download tar.gz file, locate the region file (e.g. northamerica)
- TimeZone names are divided by region

```
# Monaco
# Shanks & Pottenger give 0:09:20 for Paris Mean Time; go with Howse's
# more precise 0:09:21.
# Zone  NAME          GMTOFF  RULES   FORMAT  [UNTIL]
Zone    Europe/Monaco  0:29:32 -   LMT 1891 Mar 15
        0:09:21 -   PMT 1911 Mar 11   # Paris Mean Time
        0:00      France WE%sT   1945 Sep 16 3:00
        1:00      France CE%sT   1977
        1:00      EU   CE%sT
```

Creating the `ZoneId`

```
ZoneId.of("America/Denver");
ZoneId.of("Asia/Jakarta");
ZoneId.of("America/Los_Angeles");
ZoneId.ofOffset("UTC", ZoneOffset.ofHours(-6));
```

`ZonedDateTime` exemplified

```
ZonedDateTime.now(); //Current Date Time with Zone

ZonedDateTime myZonedDateTime = ZonedDateTime.of(2014, 1, 31, 11, 20, 30, 93020122,
ZoneId.systemDefault());

ZonedDateTime nowInAthens = ZonedDateTime.now(ZoneId.of("Europe/Athens"));

LocalDate localDate = LocalDate.of(2013, 11, 12);
LocalTime localTime = LocalTime.of(23, 10, 44, 12882);
ZoneId chicago = ZoneId.of("America/Chicago");
ZonedDateTime chicagoTime = ZonedDateTime.of(localDate, localTime, chicago);

LocalDateTime localDateTime = LocalDateTime.of(1982, Month.APRIL, 17, 14, 11);
ZonedDateTime jakartaTime = ZonedDateTime.of(localDateTime, ZoneId.of(
"Asia/Jakarta"));
```

Daylight Saving Time Begins

- In the summer
 - In the case of a gap, when clocks jump forward, there is no valid offset.

- Local date-time is adjusted to be later by the length of the gap
- For a typical one hour daylight savings change, the local date-time will be moved one hour later into the offset typically corresponding to "summer"

Daylight Saving Time Exemplified

```
LocalDateTime date = LocalDateTime.of(2012, 11, 12, 13, 11, 12);
date.atZone(ZoneId.of("America/Los_Angeles")) //2012-11-12T13:11:12-
08:00[America/Los_Angeles]

LocalDateTime daylightSavingTime = LocalDateTime.of(2014, 3, 9, 2, 0, 0, 0);
daylightSavingTime.atZone(ZoneId.of("America/Denver")); //2014-03-09T03:00-
06:00[America/Denver]

LocalDateTime daylightSavingTime2 = LocalDateTime.of(2014, 3, 9, 2, 30, 0, 0);
daylightSavingTime2.atZone(ZoneId.of("America/New_York")); //2014-03-09T03:30-
04:00[America/New_York]

LocalDateTime daylightSavingTime3 = LocalDateTime.of(2014, 3, 9, 2, 0, 0, 0);
daylightSavingTime3.atZone(ZoneId.of("America/Phoenix")); //2014-03-09T02:00-
07:00[America/Phoenix]

LocalDateTime daylightSavingTime4 = LocalDateTime.of(2014, 3, 9, 2, 59, 59,
999999999);
daylightSavingTime4.atZone(ZoneId.of("America/Chicago")); //2014-03-
09T03:59:59.999999999-05:00[America/Chicago]
```

Daylight Saving Time Ends

- In the winter
 - In the case of an overlap, when clocks are set back, there are two valid offsets.
 - This method uses the earlier offset typically corresponding to "summer".

Standard Time Exemplified

```

LocalDateTime date2 = LocalDateTime.of(2012, 11, 12, 13, 11, 12);
date2.atZone(ZoneId.of("America/Los_Angeles")); //2012-11-12T13:11:12-
08:00[America/Los_Angeles]

LocalDateTime standardTime = LocalDateTime.of(2014, 11, 2, 2, 0, 0, 0);
standardTime.atZone(ZoneId.of("America/Denver")); //2014-11-02T02:00-
07:00[America/Denver]

LocalDateTime standardTime2 = LocalDateTime.of(2014, 11, 2, 2, 30, 0, 0);
standardTime2.atZone(ZoneId.of("America/New_York")); //2014-11-02T02:30-
05:00[America/New_York]

LocalDateTime standardTime3 = LocalDateTime.of(2014, 11, 2, 2, 0, 0, 0);
standardTime3.atZone(ZoneId.of("America/Phoenix")); //2014-11-02T02:00-
07:00[America/Phoenix]

LocalDateTime standardTime4 = LocalDateTime.of(2014, 11, 2, 2, 59, 59, 999999999);
standardTime4.atZone(ZoneId.of("America/Chicago")); //2014-11-02T02:59:59.999999999-
06:00[America/Chicago]

```

Which 1:30 AM?

```

LocalDateTime standardTime6 = LocalDateTime.of(2014, 11, 2, 1, 30, 0, 0);
standardTime6.atZone(ZoneId.of("America/New_York"));
standardTime6.atZone(ZoneId.of("America/New_York"))
    .withEarlierOffsetAtOverlap().toInstant().getEpochSecond();
standardTime6.atZone(ZoneId.of("America/New_York"))
    .withLaterOffsetAtOverlap().toInstant().getEpochSecond();

```

Shifting Time

Durations and Periods

- To model a span of time (e.g. 10 days) you have two choices
 - **Duration** - a span of time in seconds and nanoseconds
 - **Period** - a span of time in years, months and days
- Both implement **TemporalAmount**

More about Duration

- Spans only seconds and nanoseconds
- Meant to adjust **LocalTime** (assumes no dates are involved)

- **static** method calls include construction for:
 - days
 - hours
 - milliseconds
 - nanoseconds
- Can have a side effect depending on which API calls you make

Duration Exemplified

```
Duration duration = Duration.ofDays(33); //seconds or nanos
Duration duration1 = Duration.ofHours(33); //seconds or nanos
Duration duration2 = Duration.ofMillis(33); //seconds or nanos
Duration duration3 = Duration.ofMinutes(33); //seconds or nanos
Duration duration4 = Duration.ofNanos(33); //seconds or nanos
Duration duration5 = Duration.ofSeconds(33); //seconds or nanos
Duration duration6 = Duration.between(LocalDate.of(2012, 11, 11), LocalDate.of(2013, 1, 1));
```

More about Period

- Spans years, months, weeks and days
- Meant to adjust **LocalDate** (assumes no times are involved)
- **static** method calls include construction for:
 - days
 - months
 - weeks
 - years
- Can also have a side effect depending on which API call you make

Period Exemplified

```
Period p = Period.ofDays(30);
Period p1 = Period.ofMonths(12);
Period p2 = Period.ofWeeks(11);
Period p3 = Period.ofYears(50);
```

Shifting Dates and Time

- Any class that derives from `Temporal` has the ability to add or remove any time using methods:
 - `plus`
 - `minus`
- Changing any one implementation of a `Temporal` will provide a copy!

Shifting `LocalDate`

- A shift of `LocalDate` can be done with:
 - a `TemporalAmount` (`Period`)
 - a `long` with `TemporalUnit` (`ChronoUnit`)

```
LocalDate localDate = LocalDate.of(2012, 11, 23);
localDate.plus(3, ChronoUnit.DAYS); //2012-11-26
localDate.plus(Period.ofDays(3)); //2012-11-26
try {
    localDate.plus(Duration.ofDays(3)); //2012-11-26
} catch (UnsupportedTemporalTypeException e) {
    e.printStackTrace();
}
```

Shifting `LocalTime`

- A shift of `LocalTime` can be done with:
 - a `TemporalAmount` (`Duration`)
 - a `long` with `TemporalUnit` (`ChronoUnit`)

```
LocalTime localTime = LocalTime.of(11, 20, 50);
localTime.plus(3, ChronoUnit.HOURS); //14:20:50
localTime.plus(Duration.ofDays(3)); //11:20:50
try {
    localTime.plus(Period.ofDays(3));
} catch (UnsupportedTemporalTypeException e) {
    e.printStackTrace();
}
```

Temporal Adjusters

- New construct
- `interface` that can be implemented to specialize a time shift

- Use Case - An object that shifts time based on external factors

```
@FunctionalInterface
public interface TemporalAdjuster {
    Temporal adjustInto(Temporal temporal);
}
```

Overly Simplified Temporal Adjuster

```
TemporalAdjuster fourMinutesFromNow = new TemporalAdjuster() {
    @Override
    public Temporal adjustInto(Temporal temporal) {
        return temporal.plus(4, ChronoUnit.MINUTES);
    }
};

LocalTime localTime = LocalTime.of(12, 0, 0);
localTime.with(fourMinutesFromNow); //12:04
```

But, wait there's more!

Remember this?

```
@FunctionalInterface
public interface TemporalAdjuster {
    Temporal adjustInto(Temporal temporal);
}
```

That's a Java 8 Lambda! Therefore `fourMinutesFromNow` can now be:

```
TemporalAdjuster fourMinutesFromNow = temporal -> temporal.plus(4, ChronoUnit.
MINUTES);
LocalTime localTime = LocalTime.of(12, 0, 0);
localTime.with(fourMinutesFromNow); //12:04
```

Refactoring and inlining

```
LocalTime.of(12, 0, 0).with(temporal -> temporal.plus(4, ChronoUnit.MINUTES));
```

Parsing and Formatting

- Converting dates and times from a String is always important
- `java.time.format.DateFormatter`
- Immutable and Threadsafe

Formatting `LocalDate`

```
DateTimeFormatter dateFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM);  
  
dateFormatter.format(LocalDate.now()); // Jan. 19, 2014
```

Formatting `LocalTime`

```
DateTimeFormatter timeFormatter =  
    DateTimeFormatter.ofLocalizedTime(FormatStyle.MEDIUM);  
  
timeFormatter.format(LocalTime.now()); //3:01:48 PM
```

Formatting `LocalDateTime`

```
DateTimeFormatter dateTimeFormatter =  
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM, FormatStyle.SHORT);  
  
dateTimeFormatter.format(LocalDateTime.now()); // Jan. 19, 2014 3:01 PM
```

Formatting Customized Patterns

```
DateTimeFormatter obscurePattern =  
    DateTimeFormatter.ofPattern("MMMM dd, yyyy '(In Time Zone: 'VV')'");  
ZonedDateTime zonedNow = ZonedDateTime.now();  
  
obscurePattern.format(zonedNow); //January 19, 2014 (In Time Zone: America/Denver)
```

Formatting with Localization

- Localization using `java.util.Locale` is available for:

- `ofLocalizedDate`
- `ofLocalizedTime`
- `ofLocalizedDateTime`

```
ZonedDateTime zonedDateTime = ZonedDateTime.now(ZoneId.of("Europe/Paris"));

DateTimeFormatter longDateTimeFormatter =
    DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL, FormatStyle.FULL).withLocale
        (Locale.FRENCH);
longDateTimeFormatter.getLocale(); //fr
longDateTimeFormatter.format(zonedDateTime); //samedi 19 janvier 2014 00 h 00 CET
```

Shifting Time Zones

```
LocalDateTime localDateTime = LocalDateTime.of(1982, Month.APRIL, 17, 14, 11);
ZonedDateTime jakartaTime = ZonedDateTime.of(localDateTime, ZoneId.of(
    "Asia/Jakarta"));
jakartaTime.withZoneSameInstant(ZoneId.of("America/Los_Angeles")); //1982-04-
16T23:11-08:00[America/Los_Angeles]
jakartaTime.withZoneSameLocal(ZoneId.of("America/New_York")); //1982-04-17T14:11-
05:00[America/New_York]
```

Temporal Querying

- Process of asking information about a `TemporalAccessor`
 - `LocalDate`
 - `LocalTime`
 - `LocalDateTime`
 - `ZonedDateTime`

```
@FunctionalInterface
public interface TemporalQuery<R> {
    R queryFrom(TemporalAccessor temporal);
}
```


Lab: A Festive Example

Step 1: Create a test called `testDaysUntilChristmas` in `DatesTest` with the following content:

```
@Test
public void testDaysBeforeChristmas() {
    TemporalQuery<Long> daysBeforeChristmas = temporal -> {
        LocalDate localDate = LocalDate.from(temporal);
        long d = ChronoUnit.DAYS.between(localDate,
            LocalDate.of(localDate.getYear(), 12, 25));
        if (d >= 0) return d;
        return ChronoUnit.DAYS.between
            (localDate, LocalDate.of(localDate.getYear() + 1, 12, 25));
    };

    System.out.println(LocalDate.of(2013, 12, 26).query(daysBeforeChristmas)); //364
}
```

Step 2: Run the test

Simple Parsing

```
DateTimeFormatter dateFormatter = DateTimeFormatter.ofLocalizedDate(FormatStyle.
MEDIUM);
dateFormatter.parse("Jan 19, 2014")); // {}, ISO resolved to 2014-01-19
```

- Parses to `java.time.format.Parsed` which is rather useless
- The more effective call is to `parse(CharSequence, TemporalQuery)`

First Attempt

```
TemporalQuery<LocalDate> localDateTemporalQuery = new TemporalQuery<LocalDate>() {
    @Override
    public LocalDate queryFrom(TemporalAccessor temporal) {
        return LocalDate.from(temporal);
    }
};

dateFormatter.parse("Jan 19, 2014", localDateTemporalQuery); //2014-01-19
```

Second Attempt

```
dateFormatter.parse("Jan 19, 2014", temporal -> LocalDate.from(temporal)); //2014-01-19
```

Last attempt

```
dateFormatter.parse("Jan 19, 2014", LocalDate::from); // Jan 19, 2014
```

Interoperability with Legacy Code

- `calendar.toInstant()` - converts the Calendar object to an Instant.
- `gregorianCalendar.toZonedDateTime()` - converts a GregorianCalendar instance to a ZonedDateTime.
- `gregorianCalendar.from(ZonedDateTime)` - creates a GregorianCalendar object using the default locale from a ZonedDateTime instance.
- `date.from(Instant)` - creates a Date object from an Instant.
- `date.toInstant()` - converts a Date object to an Instant.
- `timeZone.toZoneId()` - converts a TimeZone object to a ZoneId.

```
GregorianCalendar gregorianCalendar = new GregorianCalendar();  
gregorianCalendar.toZonedDateTime();
```

Generics

- Generics
- Get Put Principles
- Wildcards

Generics

- Add stability to your code by making more of your bugs detectable at *compile time* * Enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.
- Provide a way for you to re-use the same code with different inputs, requiring less code
- Eliminates Casting
- One of the harder concepts in Java Programming since JDK 5.

Eliminating Casting

Before:

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

After:

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0);    // no cast
```

Diamond Operator

In Java SE 7, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
List<String> list = new ArrayList<>();  
list.add("hello");  
String s = list.get(0);    // no cast
```

Generics with **for**

```
List<Integer> ints = Arrays.asList(1,2,3);
int s = 0;
for (int n : ints) { s += n; }
```

Unreadability without Generics

```
List ints = Arrays.asList( new Integer[] {
    new Integer(1), new Integer(2), new Integer(3)
} );
int s = 0;
for (Iterator it = ints.iterator(); it.hasNext(); ) {
    int n = ((Integer)it.next()).intValue();
    s += n;
}
```

Unreadability with Arrays

```
int[] ints = new int[] { 1,2,3 };
int s = 0;
for (int i = 0; i < ints.length; i++) { s += ints[i]; }
```

Notes:

- Less flexible
- Less readable

Erasure

Comparing these two sets of code:

The following uses generics...

```
List<String> words = new ArrayList<String>();
words.add("Hello ");
words.add("world!");
String s = words.get(0)+words.get(1);
```

The following doesn't and uses a *raw type*.

```
List words = new ArrayList();
words.add("Hello ");
words.add("world!");
String s = ((String)words.get(0))+((String)words.get(1))
```

But at runtime, *they are the same* due to *erasure*.

`List<Integer>`, `List<String>`, and `List<List<String>>` are all represented at run-time by the same type, `List`

Generics vs. Templates

- Generics in Java resemble templates in C++.
- Keep in mind: Syntax and semantics.
 - Syntax is deliberately similar
 - Semantics are deliberately different.

Template Declarations in C++ vs. Generic Declarations in Java

In C++, nested parameters require extra spaces, so you see things like this:

```
List< List<String> >
```

In Java, no spaces are required, and it's fine to write this:

```
List<List<String>>
```

C++ Expansion vs. Java Erasure

- C++ Templates Expansion:
 - Each instance of a template at a new type is compiled separately.
 - e.g If you use a list of integers, a list of strings, and a list of lists of string, there will be three versions of the code. (*code bloat*)
 - Efficient, possible to be optimized
- Java Generics Erasure
 - Erasure doesn't track the generic type at runtime
 - This offers flexibility and less *code bloat* than expansion

- Maintains safety and ease of use to understand, to a point

Reification

Reification is making something real, bringing something into being, or making something concrete.

Java's Generics are *not reified* at runtime.

Reification Rationale

Generics are implemented using erasure as a response to the design requirement that they support migration compatibility: it should be possible to add generic type parameters to existing classes without breaking source or binary compatibility with existing clients.

Reification Rationale Continued

Without migration compatibility, the collection APIs could not be retrofitted use generics; we would probably have added a separate, new set of collection APIs that use generics. That was the approach used by C# when generics were introduced, but Java did not take this approach because of the huge amount of pre-existing Java code using collections.

Automatic Boxing of Primitives

```
List<Integer> ints = new ArrayList<>();
ints.add(1); //adding a primitive 1
int n = ints.get(0);
```

This is equivalent to:

```
List<Integer> ints = new ArrayList<>();
ints.add(Integer.valueOf(1));
int n = ints.get(0).intValue();
```

Lab: Discussing Parameterized Classes:

Step 1: In the `src/main/java` folder, and inside the `com.xyzcorp` package.

Step 2: Open `Box.java`

Step 3: Discuss creating `Box` parameterized types.

Lab: Basic Generics Test

Step 1: Create the `src/test/java` directory if necessary.

Step 2: In the `src/test/java` directory, create a package called `com.xyzcorp` if it is not there.

Step 3: Inside the package `com.xyzcorp`, create a java file called `GenericsTest.java` with the following contents:

```
package com.xyzcorp;

public class GenericsTest {

}
```

Lab: Test Using Box

Step 1: In the `GenericsTest.java` file create a test called `testUsingBox` with the following contents.

```
@Test
public void testUsingBox() {
    Box<Integer> box = new Box<>(4);
    assertEquals(new Integer(4), box.getContents());
}
```

Step 2: Run the test.

Lab: Substituting a Type Parameter with a Parameterized Type

You can also substitute a type parameter (i.e., `K`, `V`, `E`, `T`) with a parameterized type

Step 1: In the `GenericsTest.java` file create a test called `testUsingBoxOfBoxInteger` with the following contents.

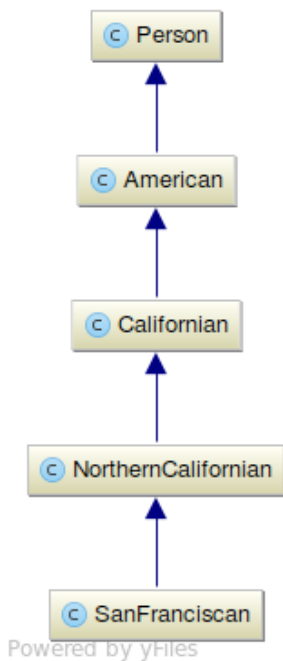
```
@Test
public void testUsingBox() {
    Box<Box<Integer>> box = new Box<>(new Box<>(10));
    assertEquals(new Integer(10), box.getContents().getContents());
}
```

Step 2: Run the test.

Wildcards

Class Diagram of People

For the wildcard generics, we will use the following classes located in `com.xyzcorp.people` package in `src/main/java`



Lab: Invariance

Step 1: Create a test called `testInvariance()` test located in the `GenericsTest` with the following:


```

@Test
public void testInvariance() {
    //Call by site
    Box<Californian> boxOfCalifornians = new Box<>();

    //Setters OK
    boxOfCalifornians.setContents(new Californian());

    //Getters OK
    Californian californian = boxOfCalifornians.getContents();

    System.out.println("boxOfCalifornians = " + boxOfCalifornians);
}

```

Step 2: Run the test to ensure that it all works.

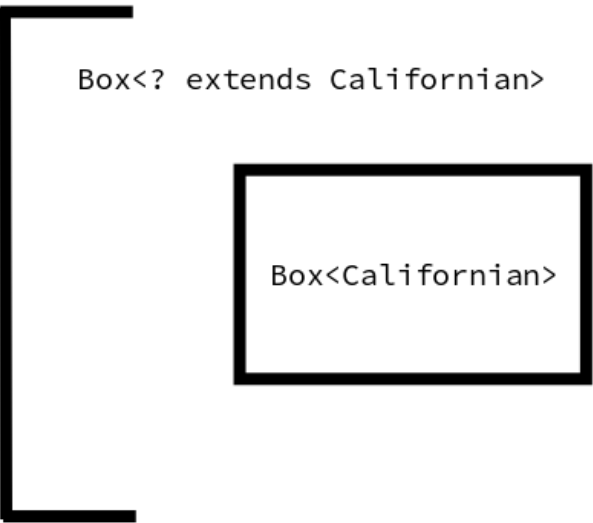
By default generics are invariant. Meaning that the given **Box** cannot vary in the types used. The Box is *always* going to be a box of **Californians** both on the assignment and instantiation.

Covariance

S is a subtype of **T** iff **List<S>** is a subtype of **List<T>**

Box<? extends Californian>

Box<Californian>





```
Box<? extends Californian>
```

```
Box<SanFranciscan>
```

Lab: An Attempt at Covariance

Step 1: In the `GenericsTest` add the following test `testCovarianceAssignments`

Step 2: Add the following content:

```
@Test
public void testCovarianceAssignments() {
    Box<NorthernCalifornian> boxOfNorthernCalifornians = new Box<>();

    //This is an attempt at covariance, this will not work
    Box<Californian> boxOfCalifornians = boxOfNorthernCalifornians;
}
```

Step 3: Describe why this did not work.

Lab: Try Other Variance Assignments

Step 1: In the `testCovarianceAssignments` that we have been using up to this point try the following assignments, one by one.

```

@Test
public void testCovarianceAssignments() {
    Box<NorthernCalifornian> boxOfNorthernCalifornians = new Box<>();

    //This is an attempt at covariance, this will not work
    Box<? extends Californian> boxOfCalifornians = boxOfNorthernCalifornians;
    Box<? extends Object> boxOfObjects = boxOfNorthernCalifornians;
    Box<? extends Person> boxOfPeople = boxOfNorthernCalifornians;
    Box<? extends American> boxOfAmericans = boxOfNorthernCalifornians;
    Box<? extends NorthernCalifornian> boxOfNorthernCalifornians2 =
boxOfNorthernCalifornians;
    Box<? extends SanFranciscan> boxOfSanFranciscans = boxOfNorthernCalifornians;
}

```

Step 2: Explain why the last one fails, after reviewing, please comment the last line out.

Lab: <? extends Object>

- <? extends Object> is nearly equivalent to <?>

Step 1: In the test that we are working on change `Box<? extends Object> boxOfObjects = boxOfNorthernCalifornians` to `<?>`

```

@Test
public void testCovarianceAssignments() {
    Box<NorthernCalifornian> boxOfNorthernCalifornians = new Box<>();

    //This is an attempt at covariance, this will not work
    Box<? extends Californian> boxOfCalifornians = boxOfNorthernCalifornians;
    Box<?> boxOfObjects = boxOfNorthernCalifornians;
    Box<? extends Person> boxOfPeople = boxOfNorthernCalifornians;
    Box<? extends American> boxOfAmericans = boxOfNorthernCalifornians;
    Box<? extends NorthernCalifornian> boxOfNorthernCalifornians2 =
boxOfNorthernCalifornians;
    Box<? extends SanFranciscan> boxOfSanFranciscans = boxOfNorthernCalifornians;
}

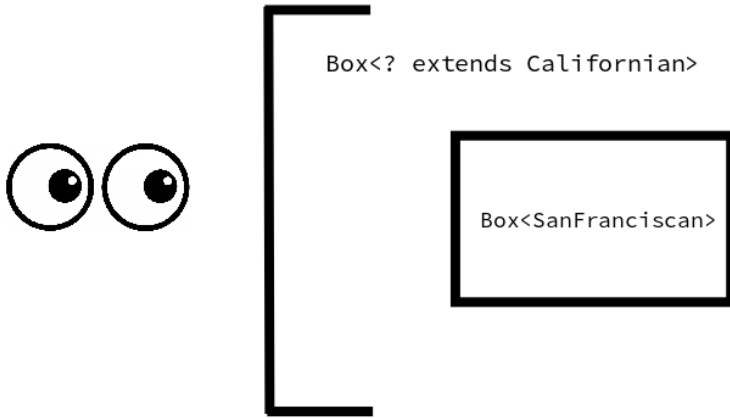
```



For an interesting discussion of <?> edge cases see [this StackOverflow article](#).

Get Principle

Use an `extends` wildcard when you only get values out of a structure



Lab: Covariance Get Principle

Step 1: In the `GenericsTest` create a test called `testCovarianceGetPrinciple`

Step 2: In the test itself add the following lines.

```
@Test
public void testCovarianceGetPrinciple() {
    Box<SanFranciscan> boxOfSanFranciscans = new Box<>();
    Box<? extends Californian> californians = boxOfSanFranciscans;

    Object object = californians.getContents();
}
```

Step 3: Object has been provided for you, add lines to see if you can assign to a `Person`, `American`, `Californian`, `Northern Californian`, and `San Franciscan`

Step 4: Comment out what didn't compile.

Step 5: Explain why certain retrievals didn't work

Lab: The Covariance Put Principle

Step 1: In the `GenericsTest` create a new test called `testCovariancePutPrinciple`

Step 2: Start with the following content and work your way down to `San Franciscan` from `Object`

```
@Test
public void testCovariancePutPrinciple() {
    Box<SanFranciscan> boxOfSanFranciscans = new Box<>();
    Box<? extends Californian> californians = boxOfSanFranciscans;

    californians.setContents(new Object());
}
```

Step 3: Discuss why it is *not* safe to "set" information from `californians`

Step 4: Try one more line, `californians.setContents(null)` and explain why that one makes sense.

Step 5: Comment out the lines that did not work.

Contravariance

`S` is a supertype of `T` iff `List<S>` is a supertype of `List<T>`

`Box<? super Californian>`

`Box<Californian>`

`Box<? super Californian>`

`Box<Object>`

Lab: An Attempt at Contravariance

Step 1: In the `GenericsTest` add the following test `testContravarianceAssignments`

Step 2: Add the following content:

```

@Test
public void testContravarianceAssignments() {
    Box<Californian> boxOfCalifornians = new Box<>();

    //This is an attempt at contravariance, this will not work
    Box<? super Object> boxOfObjects = boxOfCalifornians;
}

```

Step 3: Describe why this did not work

Lab: Try Other Variance Assignments

Step 1: In the `testContravarianceAssignments` that we have been using up to this point try the following assignments, one by one.

```

@Test
public void testCovarianceAssignments() {
    Box<Californian> boxOfCalifornians = new Box<>();

    //This is an attempt at contravariance, this will not work
    Box<? super Object> boxOfObjects = boxOfCalifornians;
    Box<? super Person> boxOfPeople = boxOfCalifornians;
    Box<? super American> boxOfAmericans = boxOfCalifornians;
    Box<? super NorthernCalifornian> boxOfNorthernCalifornians =
        boxOfCalifornians;
    Box<? super SanFranciscan> boxOfSanFranciscans =
        boxOfNorthernCalifornians;
}

```

Step 2: Explain why the first three failed.

Lab: Contravariance Get Principle

Step 1: In the `GenericsTest` create a test called `testContravarianceGetPrinciple`

Step 2: In the test itself add the following lines.

```

@Test
public void testContravarianceGetPrinciple() {
    Box<Object> boxOfObjects = new Box<>();
    Box<? super SanFranciscan> boxOfSanFranciscansAndSuperclasses = boxOfObjects;

    Object object = boxOfSanFranciscansAndSuperclasses.getContents();
}

```

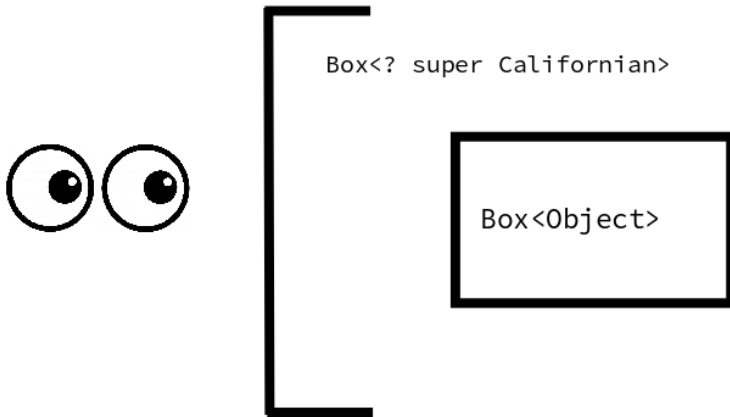
Step 3: Object has been provided for you, add lines to see if you can assign to a `Person`, `American`, `Californian`, `Northern Californian`, and `San Franciscan`

Step 4: Comment out what didn't compile.

Step 5: Explain why most of retrievals minus `Object` didn't work.

Put Principle

Use a `super` wildcard when you set values into a structure



Lab: The Contravariance Put Principle

Step 1: In the `GenericsTest` create a new test called `testContravariancePutPrinciple`

Step 2: Start with the following content and work your way down to `San Franciscan` from `Object`

```
@Test
public void testContravariancePutPrinciple() {
    Box<Object> boxOfCalifornians = new Box<>();
    Box<? super Californian> boxOfSanFranciscansAndSuperclasses = boxOfCalifornians;

    boxOfSanFranciscansAndSuperclasses.setContents(new Object());
}
```

Step 3: Discuss why it is **not** safe to

Step 4: Try one more line, `californians.setContents(null)` and explain why that one makes sense.

Step 5: Comment out the lines that did not work.

Using Wildcards in Methods

Step 1: In `GenericsTest` create a test method called `testVariancesInMethod()` with the following content:

```
@Test
public void testVariancesInMethod() {
    List<Integer> items = Arrays.asList(5, 10, 12, 10, 19, 44);
    assertEquals(Optional.of(5), findFirst(items));
}
```

Step 2: In the same test file, create a non-test method called `findFirst` that would pass the `testVarianceInMethod`

Step 3: Discuss solution

Generic Method in a Generic Class returning a different type

Often times when a class is parameterized, a method can use another parameterized type either to use in conjunction with the types with the class:

```
class A<T> {
    public <U> U foo(T t) {
        //return a type U
    }
}
```

`U` may or not be different than `T` at runtime, but the potential should be present.

This is incorrect, and is referred to as *type hiding*.

```
class A<T> {
    public <T> T foo(T t) {
        //return a type U
    }
}
```

Generic Static Method in a Generic Class returning a different type

Also when a class is parameterized, a `static` method can use another parameterized type either to use in conjunction with the types with the class.

The type system is different from the object graph. There all types established are applicable whether is is `static` or non-`static`


```
class A<T> {
    public static <U> U foo(T t) {
        //return a type U
    }
}
```

U may or not be different than **T** at runtime, but the potential should be present.

This is incorrect, but is not *type hiding*, but is bad and unreadable form.

```
class A<T> {
    public static <T> T foo(T t) {
        //return a type U
    }
}
```

Lab: Creating a generic method in a generic class.

Step 1: In `GenericsTest` create `testMap` with the following content:

```
@Test
public void testMap() throws Exception {
    Box<Integer> box = new Box<>(4);
    Box<String> newBox = box.map(integer ->
        Stream.generate(() -> "Wow")
            .limit(integer)
            .collect(Collectors.joining()));
    System.out.println("newBox = " + newBox);
}
```

Step 2: In `Box.java` add a method called `map` that takes a `java.util.function.Function` (see Java API)

Step 3: The implementation of `map` should take the function and return a *new* `Box` with the previous state transformed by the function.



Nerd Alert

Multiple Bounds

A type parameter can have multiple bounds

```
<T extends B1 & B2 & B3>
```

If one of the bounds is a class, it must be specified first. For example..

```
class A { /* ... */ }  
interface B { /* ... */ }  
interface C { /* ... */ }  
  
class D <T extends A & B & C> { /* ... */ }
```

If not you will receive a compile time exception.

```
class D <T extends B & A & C> { /* ... */ } // compile-time error
```

From [The Java Documentation Online](#)

Lab: Multiple Bounds

Step 1: In `src/test/java` and in the package `com.xyzcorp` create another class, `MultipleBoundsTest.java`

Step 2: Create a test in the `MultipleBoundsTest` called `testMultipleInheritance` and a non-test method `foo` with the following content.

```

package com.xyzcorp;

import org.junit.Test;

import java.io.CharArrayWriter;

public class MultipleBoundsTest {

    public <FILL_HERE> void foo(T t) throws IOException {
        t.append('c');
        t.append('d');
        t.flush();
        t.close();
    }

    @Test
    public void testMultipleInheritance() throws IOException {
        CharArrayWriter writer = new CharArrayWriter(40);
        foo(writer);
        System.out.println(writer.toCharArray());
    }
}

```

Step 3: For method `foo` replace what is in `FILL_HERE` with what you suspect the parameterized type `T` should look like if `T` is also `Appendable`, `Closeable`, and `Flushable` (all are interfaces)

<T extends Comparable<T>>

Why does `<T extends Comparable<T>>` look the way it does?

This should make sense.

```

public class Foo implements Comparable<Foo>{
    private int i = 0;
    @Override
    public int compareTo(Foo o) {
        return Integer.valueOf(i).compareTo(o.i);
    }
}

```

But if it was just `Comparable` it would have to look like this:

```
public class Foo implements Comparable {
    private int i = 0;
    @Override
    public int compareTo(Object o) {
        return Integer.valueOf(i).compareTo(o.i);
    }
}
```

Therefore, <T extends Comparable<T>>

Therefore this code should make sense.

```
public class MyCollection<T extends Comparable<T>> {
    private final T[] items;

    public MyCollection(T... items) { //varargs
        this.items = items;
    }

    public Optional<T> max() {
        if (items.length == 0) return Optional.empty();
        T result = items[0];
        for (T item : items) {
            if (item.compareTo(result) > 0) result = item;
        }
        return Optional.of(result);
    }
}
```

The Problem with <T extends Comparable>

```

public class MyCollection<T extends Comparable> {
    private final T[] items;

    public MyCollection(T... items) { //varargs
        this.items = items;
    }

    public Optional<T> max() {
        if (items.length == 0) return Optional.empty();
        T result = items[0];
        for (T item : items) {
            if (item.compareTo(result) > 0) result = item;
        }
        return Optional.of(result);
    }
}

```



`item.compareTo(result)` is unchecked because `compareTo` is only expecting `Object` not `T`

Without `<Class<T>>`

We see this everywhere, but what is it? And why does it exist?

Consider:

```

public void listMethodsFromRawClass(Class clazz) {
    clazz.getMethods();
}

```

We can call it with anything.

```
listMethodsFromRawClass(Person.class); //Not constrained
```

With `<Class<T>>`

With `<Class<T>>` we can constrain the type of classes that are called.

Consider:

```

public void listMethodsFromRawClass(Class<Person> clazz) {
    clazz.getMethods();
}

```

We can call only call with **Person**

```
listMethodsFromRawClass(Person.class);
```

Review JDK Collections library for Generics