Session: 13

New Date and Time API

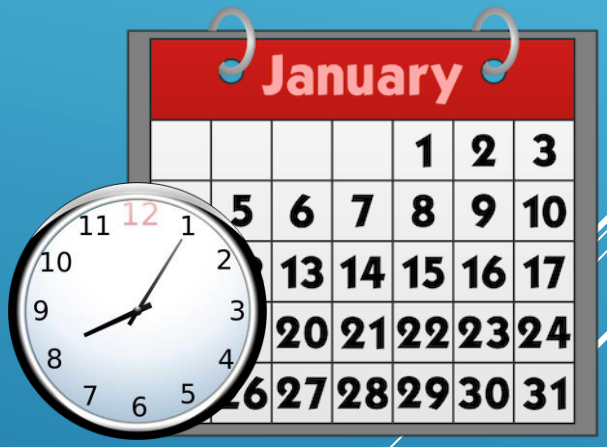
Objectives

- Explain new classes of the Date and Time API in Java 8
- Explain `Enum` and `Clock` types
- Describe the role of time-zones in Java 8
- Explain support for backward compatibility in the new API

Introduction

New Date-Time API overcomes issues faced by earlier version of date and time library such as:

- Thread-safe issue
- Poor design
- Time-zone handling issue



Classes in New Date and Time API 1/29

`java.time` package acts as a repository where all classes of new Date and Time API are located.
Complete list of classes in API is:

Classes in New Date and Time API 2/29

`Clock` class in Java 8 version is used to get the date and time using current time-zone.

`Clock` can be used in the place of `System.currentTimeInMillis()` and `TimeZone.getDefault()`.

Classes in New Date and Time API 3/29

Following Code Snippet displays the instance of using `Clock`. It represents how a clock can provide access to the current date and time using a time-zone.

```
import java.time.*; // import the package for Date-Time API classes
…
// Creates a new Clock instance based on UTC.
…
Clock defaultClock = Clock.systemUTC();
System.out.println("Clock : " + defaultClock);
…
// Creates a clock instance based on system clock zone
Clock defaultClock2 = Clock.systemDefaultZone();
System.out.println("Clock : " + defaultClock2);
```

Classes in New Date and Time API 4/29

Following Code Snippet displays how the given date can be verified against the `Clock` object:

```
public class MyClass {
private Clock clock;
…
public void process(LocalDate eventDate) {
if (eventDate.isBefore(LocalDate.now(clock))) {
// logic
}
}
}
```

Classes in New Date and Time API 5/29

Duration

`Duration` class consists of a group of methods to perform calculations based on a `Duration` object.

`plusNanos()`	`plusSeconds()`	`plusHours()`
`minusNanos()`	`minusSeconds()`	`minusHours()`
`plusMillis()`	`plusMinutes()`	`plusDays()`
`minusMillis()`	`minusMinutes()`	`minusDays()`

Classes in New Date and Time API 6/29

Duration

Following Code Snippet shows the usage of `plusDays()` and `minusDays()` methods:

```
Duration present = ... // assume code is written to
// get a present duration
Duration samplePlusA = present.plusDays(3);
Duration sampleMinusA = present.minusDays(3);
```

Here, first line of code produces a `Duration` variable, `present` that will be used as the base of calculations. It is assumed that code to create the `Duration` object is added.

Code Snippet then produces two new `Duration` objects based on the `present` object. The second line generates a `Duration`, which is equivalent to `present` plus three days. The third line builds `Duration` that is equivalent to `present` minus three days.

Classes in New Date and Time API 7/29

`Instant`
`(java.time.instant)`

`Instant` class helps in time stamp creation.

Generating an Instant:
An instance of an `Instant` can be generated using one of the `Instant` class factory methods.

Code Snippet shows an `Instant` object representing the exact moment of now, using method `Instant.now()`.

```
Instant sampleNow = Instant.now();
```

Instant Calculations:
Code Snippet displays the use of `Instant` in nanoseconds and milliseconds.

```
Instant sampleFuture = sampleNow.plusNanos(4);
// four nanoseconds in the future
Instant samplePast = sampleNow.minusNanos(4);
//four nanoseconds in the past
```

Classes in New Date and Time API 8/29

`LocalDate` class is bundled with the `java.time` package.

Creating a `LocalDate`:

`LocalDate` objects can be created using several approaches. The first approach is to get a `LocalDate` equivalent to the local date of today.

Code Snippet shows creating a `LocalDate` object using `now()`.

```
LocalDate sampleLocDaA =
LocalDate.now();
```

Obtain a `LocalDate`:

To obtain a `LocalDate`, you can also create it from a specific year, month, and day information.

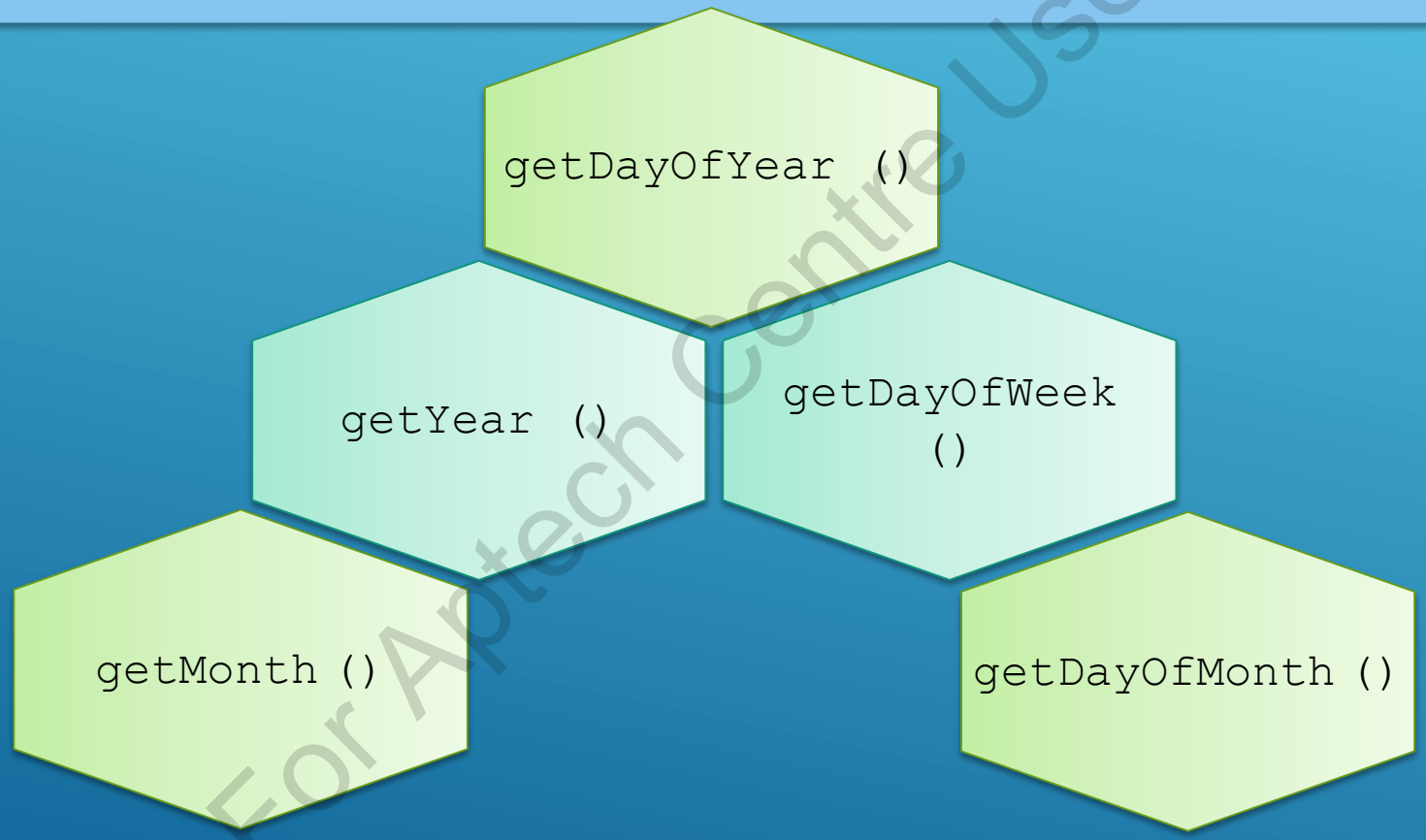
Code Snippet shows creating `LocalDate` using `of()`.

```
LocalDate sampleLocDaB =
LocalDate.of(2016, 07, 04);
```

Classes in New Date and Time API 9/29

LocalDate

Date information of a `LocalDate` object can be accessed using following methods:



Classes in New Date and Time API 10/29

Following Code Snippet illustrates date information of a `LocalDate`.

```
int year = localDate.getYear();
int dayOfMonth = localDate.getDayOfMonth();
Month month = localDate.getMonth();
int dayOfYear = localDate.getDayOfYear();
DayOfWeek dayOfWeek = localDate.getDayOfWeek();
int monthvalue = month.getValue();
```

Notice how `getMonth()` and `getDayOfWeek()` methods return an enum instead of an `int`. These `enums` can provide their data as `int` values by calling their `getValue()` methods.

Classes in New Date and Time API 11/29

LocalDate Calculations:
A set of date calculations can be achieved with the `LocalDate` class using one or more of following methods:

- `plusDays ()`
- `minusDays ()`
- `plusWeeks ()`
- `minusWeeks ()`
- `plusMonths ()`
- `minusMonths ()`
- `plusYears ()`
- `minusYears ()`

Classes in New Date and Time API 12/29

LocalDate Calculation Method:

Code Snippet displays how a `LocalDate` calculation methods works.

```
LocalDate sampleLocDa = LocalDate.of(2016,
04, 30);
LocalDate sampleLocDaA =
sampleLocDa.plusYears(4);
LocalDate sampleLocDaB =
sampleLocDa.minusYears(4);
```

In the code, `sampleLocDa`, a new instance of `LocalDate`, is created using `of()` method. Then, the code builds a new `LocalDate` instance that represents the date four years later from the specified date. Finally, the code generates a new `LocalDate` instance that denotes the date four years earlier from the specified date.

Classes in New Date and Time API 13/29

`LocalDateTime`

Represents a local date and time without any time-zone data.

Date-Time information of a `LocalDateTime` object can be accessed using `getValue()` method.

Various date and time calculations can be performed on `LocalDateTime` object with plus or minus methods.

Creating a LocalDateTime object based on a specific year, month, and day:

```
LocalDateTime sampleLocDaTiB = LocalDateTime.of (2016, 05, 07, 12, 06, 16, 054);
```

Parameters passed to `of()` are `year`, `month`, `day (of month)`, `hours`, `minutes`, `seconds`, and `nanoseconds` respectively.

Classes in New Date and Time API 14/29

Code Snippet illustrates how `LocalDateTime` calculation methods work.

```
LocalDateTime sampleLocDaTi = LocalDateTime.now();
LocalDateTime sampleLocDaTiA = sampleLocDaTi.plusYears(4);
LocalDateTime sampleLocDaTiB = sampleLocDaTi.minusYears(4);
```

The code first creates a `LocalDateTime` instance `sampleLocDaTi` signifying the current moment.

Then, the code creates a `LocalDateTime` object that denotes a date and time four years later.

Finally, the code builds a `LocalDateTime` object that denotes a date and time four years prior.

Classes in New Date and Time API 15/29

LocalTime Class

`LocalTime` class in Date-Time API signifies exact time of day without any time-zone data.

Creating a `LocalTime` Class:

A `LocalTime` instance can be generated using several approaches. The foremost approach is to create a `LocalTime` instance that denotes the exact time of now. Code Snippet shows the `now()` method.

Code Snippet:

```
LocalTime sampleLocTiA =
LocalTime.now();
```

Another approach to produce a `LocalTime` object is to create it from specific hours, minutes, seconds, and nanoseconds. Code Snippet displays the `of()` method.

Code Snippet:

```
LocalTime sampleLocTiB =
LocalTime.of(12, 24, 33, 00135);
```

Classes in New Date and Time API 16/29

LocalTime Calculations

- `LocalTime` class consists of a set of methods that can perform local time calculations.
- For example, `plusMinutes()` method adds minutes and `minusMinutes()` subtracts minutes from a given value in a calculation.
- Plus or minus methods are in `LocalDateTime` object.
- Code Snippet explains `LocalTime` calculations.

```
LocalTime sampleLocTi = LocalTime.of(12, 24, 33, 00135);
// current local time
LocalTime sampleLocTiFuture = sampleLocTi.plusHours(4);// future
LocalTime sampleLocTiPast = sampleLocTi.minusHours(4);// past
```

Classes in New Date and Time API 17/29

MonthDay Class

- `MonthDay` is an immutable Date-Time object that represents month as well as day-of-month.
- Code Snippet depicts how `MonthDay` class can be used for checking recurring date-time events.

```
…
// Code to display Birthday wishes
LocalDate dateOfBirth = LocalDate.of(2006, 02, 24);
MonthDay bday = MonthDay.of(dateOfBirth.getMonth(), dateOfBirth.
getDayOfMonth());
MonthDay currentMonthDay = MonthDay.from(today); //assume today is defined
if(currentMonthDay.equals(bday)){
System.out.println("**Colorful Joyful Birthday Buddy**");
}
else{
System.out.println("Nope, today is not your B'day");
}
```

Classes in New Date and Time API 18/29

OffsetDateTime Class

- `OffsetDateTime` is an immutable illustration of date and time with an offset.
- Code Snippet displays an example stating California is GMT or UTC–07:00 and to get a similar time-zone, static method `ZoneOffset.of()` can be used.
- After fetching the offset value, `OffSetDateTime` can be shaped by passing a `LocalDateTime` and an offset to it.

```
LocalDateTime datetime = LocalDateTime.of(2016, Month.FEBRUARY, 15,
18, 20);
// to display the result using Offset
ZoneOffset sampleoffset = ZoneOffset.of("-07:00");
OffsetDateTime date = OffsetDateTime.of(datetime, sampleoffset);
System.out.println("Sample display of Date and Time using time-zone
offset : " + date);
```

Classes in New Date and Time API 19/29

OffsetTime Class

- `OffsetTime` class is an immutable Date-Time object that denotes a time, frequently observed as hour-minute-second-offset.
- Following Code Snippet shows the complete program to fetch the seconds using the `OffsetTime` class:

```
import java.time.OffsetTime;// Class to show the result by using
// OffsetTime class method
public class MinuteOffset {
public static void main(String[] args) {
OffsetTime d = OffsetTime.now();
int e = d.getMinute();
System.out.println("Minutes: " + e);
}
}
```

Output:

Minutes: 49

Classes in New Date and Time API 20/29

Period
Class

- `Period (java.time.Period)` represents an amount of time in terms of days, months, and years.
- Duration and Period are somewhat similar; however, the difference between the two can be seen in their approach towards Daylight Savings Time (DST) when they are added to ZonedDateTime.

Classes in New Date and Time API 21/29

Period Class

Following Code Snippet displays an example to calculate the span of time from today until a birthday, assuming the birthday is on May 22nd:

```
import java.time.LocalDate;// Class to get the present day
import java.time.Month; // Class to get month related calculations
import java.time.Period;//Class to calculate the time period between two
//time instances
import java.time.temporal.ChronoUnit;
public class NextBday {
public static void main(String[] args) {
LocalDate presentday = LocalDate.now();
LocalDate bday = LocalDate.of(1983, Month.MAY, 22);
LocalDate comingBDay = bday.withYear(presentday.getYear());
// To address the belated b'day celebration.
if (comingBDay.isBefore(presentday) || comingBDay.isEqual(presentday))
{
comingBDay = comingBDay.plusYears(1);
}
Period waitA = Period.between(presentday, comingBDay);
long waitB = ChronoUnit.DAYS.between(presentday, comingBDay);
```

Classes in New Date and Time API 22/29

```
System.out.println("Totally, I need to wait for " + waitA.getMonths() + "
months, and " +
waitA.getDays() + " days to celebrate my next B'day. (" +
waitB + " days in total)");// to display the waiting time for B'day Bash
}
}
```

Output:

Totally, I need to wait for 0 months and 22 days to celebrate my next B'day. (22 days in total)

Classes in New Date and Time API 23/29

`Year` Class

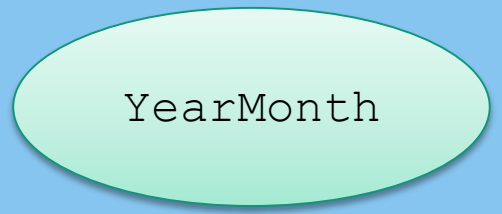
- A `Year` (`java.time.Year`) object is an immutable Date-Time object that denotes a year.
- Following Code Snippet displays the calculations using `Year` class:

```
import java.time.Year;// Class to use Year values in calculations
public class SampleYear {
public static void main(String[] args) {
System.out.println(" The Present Year():"+Year.now());
System.out.println("The year 2002 is a Leap year :"+
Year.isLeap(2002));// to display whether the year 2002 is a leap
// year or not
System.out.println("The year 2012 is a Leap year :"+
Year.isLeap(2012));
// to display whether the year 2012 is a leap year or not
}
}
```

Output:

The Present Year (): 2016
The year 2002 is a Leap year: false
The year 2012 is a Leap year: true

Classes in New Date and Time API 24/29



`YearMonth` (`java.time.YearMonth`) is a stable Date-Time object that denotes the combination of year and month. This class does not store or denote a day, time, or time-zone. For example, the value 'November 2011' can be stored in a `YearMonth`.

`YearMonth` can be used to denote things such as credit card expiry, Fixed Deposit maturity date, Stock Futures, Stock options expiry dates, or determining if the year is a leap year or not.

Classes in New Date and Time API 25/29

YearMonth

Following Code Snippet shows the `YearMonth` calculations:

```
import java.time.YearMonth;// to use the Year and Month info
public class YearMonth {
public static void main(String[] args) {
System.out.println("The Present Year Month:"+YearMonth.now());
// To display present year and month
System.out.println("Month alone:"+YearMonth.parse("2016-
04").getMonthValue()); // To display only the month value
System.out.println("Year alone:"+YearMonth.parse("2016-04"
).getYear());// to display the year value alone
System.out.println("This year is a Leap year:"
+YearMonth.parse("2016-04").isLeapYear());// leap year check
}
}
```

Output:

The Present Year Month: 2016-05

Month alone: 4

Year alone: 2016

This year is a Leap year: true

Classes in New Date and Time API 26/29

ZonedDateTime

`ZonedDateTime(java.time.ZonedDateTime)` is an immutable that represents date and time in addition to a time-zone.

The `ZonedDateTime` class is immutable. This means that all methods executing calculations on a `ZonedDateTime` object yields a new `ZonedDateTime` instance.

Classes in New Date and Time API 27/29

ZonedDateTime

Following example depicts the usage of methods to get year, month, day, hour, minute, seconds, and zone offset:

```
import java.time.ZonedDateTime;// to access Zoned Date Time
public class ZoneDT {//Class ZoneDT refers to ZonedDateTime
public static void main(String[] args) {

System.out.println(ZonedDateTime.now());
ZonedDateTime sampleZoDT = ZonedDateTime.parse("2016-04-
03T10:15:30+08:00[Asia/Singapore]");

System.out.println("Present day of the year:"+sampleZoDT.
getDayOfYear());
System.out.println("Present year:"+sampleZoDT.getYear());
}
}
```

Output:

2016-05-06T06:03:51.787+08:00[Etc/UTC]
Present day of the year: 94
Present year: 2016

Classes in New Date and Time API 28/29

ZoneID

`ZoneId` is used to recognize rules used to convert between an `Instant` and a `LocalDateTime`.

The two different ID types are as follows:

- Fixed offsets
- Geographical regions

Classes in New Date and Time API 29/29

`ZoneOffset`

A time-zone offset is the quantity of time that a time-zone differs from Greenwich/UTC.

For example, Berlin is two hours ahead of Greenwich/UTC in Spring and four hours ahead during Autumn.

The `ZoneId` instance for Berlin will reference two `ZoneOffset` instances - a +02:00 instance for Spring and a +04:00 instance for Autumn.

Code Snippet illustrates the usage of this class.

```
ZoneOffset sampleOffset = ZoneOffset.of("+05:00");
```

Enums In Java 8 1/3

Benefits of using Enums in Java:

- Enum: Type Safe
- Enum: Has its own name space
- Enum: Used inside switch statements
- Enum: New constants can be added without breaking the existing code

Enums In Java 8 2/3

`ChronoUnit` enumeration is used in following Code Snippet:

```
import java.time.LocalDate;
import java.time.temporal.ChronoUnit;
public class EnumDateCalculation{
    public static void main(String args[]){
    EnumDateCalculation java8enum = new EnumDateCalculation ();
    java8enum.enumChromoUnits();
    }
    public void enumChromoUnits(){
      // To display the current date
      LocalDate today = LocalDate.now();
      System.out.println("Current date: " + today);
      // To display the result 2 weeks addition to the current
date
      LocalDate nextWeek = today.plus(2, ChronoUnit.WEEKS);
      System.out.println("After 2 weeks: " + nextWeek);
      // To display the result 2 months addition to the current
date
      LocalDate nextMonth = today.plus(2, ChronoUnit.MONTHS);
      System.out.println("After 2 months: " + nextMonth);
      // To display the result 2 years addition to the current
```

Enums In Java 8 3/3

```
date
      LocalDate nextYear = today.plus(2, ChronoUnit.YEARS);
      System.out.println("After 2 years: " + nextYear);
      // To display the result 20 years addition to the current
date
      LocalDate nextDecade = today.plus(2, ChronoUnit.DECADES);
      System.out.println("Date after twenty year: " + nextDecade);
    }
}
```

Output:

Current date: 2016-04-07
After 2 weeks: 2016-04-21
After 2 months: 2016-06-07
After 2 years: 2018-04-07
Date after twenty year: 2036-04-07

Temporal Adjusters 1/3

Temporal Adjuster acts as a key tool in modifying the Temporal Object.

Temporal Adjuster is a functional interface that uses `adjustInto(Temporal)` method to return a copy of Temporal object with unchanged field value.

`java.time.temporal`

A Temporal Adjuster can be used to perform complicated date 'math' that is popular in business applications.

`Date-Time` objects

For example, it can be used to find 'first Thursday of the month' or 'next Tuesday'.

`FirstDayOfMonth ( )`

Temporal Adjusters 2/3

Following Code Snippet shows how to find the first day of a month using a specified date:

```
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
import java.time.DayOfWeek;
public class TemporalAdj {
    public static void main(String args[]){
       TemporalAdj TemporalAdj = new TemporalAdj();
       TemporalAdj.sampleAdj();
       }
       public void sampleAdj(){
       // To display the current date
       LocalDate sampledateA = LocalDate.now();
       System.out.println("Current date: " + sampledateA);
       // To display the next Wednesday from current date
       LocalDate nextWednesday =
       sampledateA.with(TemporalAdjusters.next(DayOfWeek.
       WEDNESDAY));
```

Temporal Adjusters 3/3

```
        System.out.println("Next Wednesday on : " + nextWednesday);
        // To display the second Sunday of next month
        LocalDate firstInYear =
        LocalDate.of(sampledateA.getYear(),sampledateA.
        getMonth(), 1);
        LocalDate secondSunday = firstInYear.with(TemporalAdjusters.
        nextOrSame(DayOfWeek.SUNDAY)).with(TemporalAdjusters.next(DayOfWeek.
        SUNDAY));
        System.out.println("Second Sunday on : " + secondSunday);
}
}
```

Output:
Current date: 2016-04-07
Next Wednesday on: 2016-04-13
Second Sunday on: 2016-04-10

Backward Compatibility with Older Versions 1/3

`toInstant()`

```
Date sampleDate = new Date();
Instant sampleNow = sampleDate.toInstant();
LocalDateTime dateTime =
LocalDateTime.ofInstant(sampleNow, myZone);
ZonedDateTime zdt =
ZonedDateTime.ofInstant(sampleNow, myZone);
```

`Instant`, `ZoneId`

In the given code, `toInstant()` method is being added to the original `Date` and `Calendar` objects to convert them into new Date-Time API.

Backward Compatibility with Older Versions 2/3

`ofInstant(Instant, ZoneId)` method is used to get a `LocalDateTime` or `ZonedDateTime` object.

```
import java.time.LocalDateTime;// to initiate local date and time
import java.time.ZonedDateTime; // to initiate zoned time
import java.util.Date;
import java.time.Instant;
import java.time.ZoneId;

public class BWCompatibility {
    public static void main(String args[]){
            BWCompatibility bwcompatibility = new BWCompatibility();
            bwcompatibility.sampleBW();
        }
        public void sampleBW(){
          // To display the current date
          Date sampleCurDay = new Date();
          System.out.println(" Desired Current date= " + sampleCurDay);
          // to display result
          // To display the instant of current date
          Instant samplenow = sampleCurDay.toInstant();
          ZoneId samplecurZone = ZoneId.systemDefault();
```

Backward Compatibility with Older Versions 3/3

```
        // To display the current local date
        LocalDateTime sampleLoDaTi = LocalDateTime.ofInstant(samplenow,
        samplecurZone);
        System.out.println(" Desired Current Local date= " + sampleLoDaTi);

          // To display result
          // To display the desired current zoned date
          ZonedDateTime sampleZoDaTi = ZonedDateTime.ofInstant(samplenow,
          samplecurZone);
          System.out.println(" Desired Current Zoned date= " + sampleZoDaTi);
          // To display result
    }
}
```

Output:

Desired Current date= Fri May 06 07:32:58 EDT 2016
Desired Current Local date= 2016-05-06T07:32:58.769
Desired Current Zoned date= 2016-05-06T07:32:58.769-04:00[America/New York]

Parsing and Formatting Dates

Parsing dates from strings and formatting dates to strings is possible with the `java.text.SimpleDateFormat` class.

Code Snippet shows an example of how the `SimpleDateFormat` class works on `java.util.Date` instances.

```
SimpleDateFormat format = new SimpleDateFormat("yyyy-MM-dd");
String dateString = format.format( new Date() );
Date samplDate = format.parse ("2011-03-25");
```

TimeZone (java.util.TimeZone) 1/2

`TimeZone (java.util.TimeZone)` is used in time-zone bound calculations.

Code Snippet displays a simple example of how to get the time-zone from a Calendar.

```
Calendar cal = new GregorianCalendar();
TimeZone tiZo = cal.getTimeZone();
```

Code Snippet displays a simple example of how to set time-zone.

```
cal.setTimeZone(tiZo);
```

Code Snippet shows two ways to obtain a `TimeZone` instance.

```
TimeZone tiZo = TimeZone.getDefault();
OR
TimeZone tiZo =
TimeZone.getTimeZone("Europe/Paris");
```

TimeZone (java.util.TimeZone) 2/2

Following Code Snippet shows a sample of time-zone:

```
import java.time.ZonedDateTime;
import java.time.ZoneId;

public class Java8CurTZone {
    public static void main(String args[]){
       Java8CurTZone = new Java8CurTZone();
       java8curtzone.sampleZDTime();
    }
    public void sampleZDTime(){
       // To display the current date and time
       ZonedDateTime dateA = ZonedDateTime.parse("2016-04-
       03T10:15:30+08:00[Asia/Singapore]");
       System.out.println("dateA: " + dateA);
       // To display the zoneId
       ZoneId sampleidA = ZoneId.of("Asia/Singapore");
       System.out.println("ZoneId: " + sampleidA);
       // To display the current Zone
       ZoneId samplecurrentZoneA = ZoneId.systemDefault();
       System.out.println("CurrentZone: " +
       samplecurrentZoneA);
    }
}
```

Output:
dateA: 2016-04-03
T10:15:30+08:00[Asia/Singapore]
ZoneId: Asia/Singapore
CurrentZone: Etc/UTC

SUMMARY

- The new Date-Time API introduced in Java 8 is a solution for many unaddressed drawbacks of the previous API.
- Date-Time API contains many classes to reduce coding complexity and provides various additional features to work on date and time.
- Enum in Java denotes fixed number of well-known values.
- TemporalAdjuster is a functional interface and a key tool for altering a temporal object.
- Java TimeZone class is a class that denotes time-zones and is helpful when doing calendar arithmetic across time-zones.
- A time-zone offset is the quantity of time that a time-zone differs from Greenwich/UTC.