# Java Learning Center

# Java 8

## New Features

*Put Into Knowledge & Practice*

**Doc Version - 1.0**

**Author**
**Srinivas Dande**

# Contents

**Java 8 New Features ..... 1**

1. Default Method in Interface ..... 3
2. Static Method in Interface ..... 7
3. New Date and Time API ..... 9
    - 3.1 DayOfWeek ..... 10
    - 3.2 Month ..... 10
    - 3.3 LocalDate ..... 11
    - 3.4 LocalTime ..... 13
    - 3.5 LocalDateTime ..... 14
    - 3.6 Instant ..... 15
    - 3.7 Duration ..... 17
    - 3.8 Period ..... 18
4. Using this as Method Parameter ..... 19
5. Accessing Parameter Name Using Reflection ..... 20
6. Funactional Programming ..... 22
    - 6.1 Moving from Value Parameterization to Behaviour Parameterization 22
7. Lambda ..... 27
    - 7.1. Lambda Introduction .....
    - 7.2. Lambda Uses .....
    - 7.3. Funactional Interface .....
    - 7.4. Funaction Descriptor .....
    - 7.5. Working with Funactional Interface .....
    - 7.6. Type checking in Lambdas .....
    - 7.7. Method and Consutuctor reference .....
    - 7.8. Lambdas and Method Reference .....
8. Streams .....
9. Optional .....

## 1. Default Method in Interface

- Interfaces in Java are the collection of abstract methods, which need to be implemented by concrete class.
- When you add new method in an existing interface than all the implementing classes need to be changed to implement new method. So till Java 7, adding a method to an existing interface was a maintainence problem.
- When you have one interface with some abstract methods and you have defined many subclasses then in all the subclass you need to override all the method in all the non abstract classes.

| Interface | First Sub Class | Second Sub Class |
|---|---|---|
| interface Inter1{<br>void show();<br>void display();<br>} | class Hello implements Inter1{<br>public void show(){}<br>public void display(){}<br>} | class Hai implements Inter1{<br>public void show(){}<br>public void display(){}<br>} |

- When you add a new method in this interface then it will be a maintainence problem because you need to override new method in all the existing sub classes.

| Interface | First Sub Class | Second Sub Class |
|---|---|---|
| interface Inter1{<br>void show();<br>void display();<br>void process();<br>} | class Hello implements Inter1{<br>public void show(){}<br>public void display(){}<br>}<br><br>Compilation Error | class Hai implements Inter1{<br>public void show(){}<br>public void display(){}<br>}<br><br>Compilation Error |

- Java 8 solves this problem by introducing default methods concept.
- Before Java 8 all methods in the interface were implicitly abstract. It means you can't include body for the method in the interface.
- From Java 8, it is possible to define a default implementation for the method of the interface. This new feature is called the *default method*.
- Now interface is no more pure abstract classes.
- Now interfaces have come closer to abstract classes which show similar attributes.

- **The default method solves this problem by providing an implementation that will be used if no other implementation is explicitly provided.**

| Interface | First Sub Class | Second Sub Class |
|---|---|---|
| interface Inter1{<br>void show();<br>void display();<br>default void process(){}<br>} | class Hello implements Inter1{<br>public void show(){}<br>public void display(){}<br>}<br><br>No Compilation Error | class Hai implements Inter1{<br>public void show(){}<br>public void display(){}<br>public void process(){}<br>}<br><br>You can override also |

- **When you extend an interface that contains a default method, you can do the following:**
  - **When you don't override default method in the subclass then default implementation will be inherited from implemented interface.**
  - **If you want you can override the default method in your class.**
  - **When you override the default method in the subclass and you want to invoke the implementation from interface also then you can use**

    **<InterfaceName>.super.<methodName>();**

*JAVA8LAB1.JAVA*

```
interface Inter1 {
default void process() {
System.out.println("\n** process() in Inter1 **");
}
default void show() {
System.out.println("\n** show() in Inter1 **");
}
}
class Hello implements Inter1 {
public void process() {
System.out.println("** process() in Hello **");
}
}
public class Java8Lab1 {
public static void main(String[] args) {
Hello h = new Hello();
h.process();
h.show();
}
}
```

*JAVA8LAB2.JAVA*

```
interface Inter1 {
default void show() {
System.out.println("\n** show() in Inter1 **");
}
}

class Hello implements Inter1 {
public void show() {
System.out.println("** show() in Hello **");
Inter1.super.show();
}
}

public class Java8Lab2 {
public static void main(String[] args) {
Hello h = new Hello();
h.show();
}
}
```

- There is a possibility that a class may end up having the same method with same signature from multiple sources.
- It may cause ambiguity problem. We can follow the below rules to resolve the ambiguity.

1. Super classes always wins. When the method is declared in the super class and also in the interface.

*JAVA8LAB3.JAVA*

```
interface Inter1 {
default Number getNumber() {
System.out.println("getNumber in Inter1");
return 42;
}
}

class A {
public Integer getNumber() {
System.out.println("getNumber in A");
return 10;
}
}

class Hello extends A implements Inter1 { }

public class Java8Lab3 {
public static void main(String[] args) {
Hello h = new Hello();
System.out.println(h.getNumber());
}
}
```

2. Otherwise, more specific interface default method always wins.

*JAVA8LAB4.JAVA*

```
interface Inter1 {
default Number getNumber() {
System.out.println("getNumber in Inter1");
return 42;
}
}

interface Inter2 extends Inter1{
default Integer getNumber() {
System.out.println("getNumber in Inter2");
return 10;
}
}

class Hello implements Inter1,Inter2 { }
public class Java8Lab4 {
public static void main(String[] args) {
Hello h = new Hello();
System.out.println(h.getNumber());
}
}
```

*JAVA8LAB5.JAVA*

```
interface Inter1 {
default Number getNumber() {
System.out.println("getNumber in Inter1");
return 42;
}
}
interface Inter2 extends Inter1 { }

interface Inter3 extends Inter1 { }

class Hello implements Inter2, Inter3 { }

public class Java8Lab5 {
public static void main(String[] args) {
Hello h = new Hello();
System.out.println(h.getNumber());
}
}
```

3. **If still ambiguity occurs, then calling class needs to override the method and explicitly select a default method.**

*JAVA8LAB6.JAVA*

```
interface Inter1 {
default Number getNumber() {
System.out.println("getNumber in Inter1");
return 42;
}
}

interface Inter2 {
default Integer getNumber() {
System.out.println("getNumber in Inter2");
return 10;
}
}
class A implements Inter1 { }

class B implements Inter2 { }

// class C implements Inter1, Inter2 { }

class Hello implements Inter1, Inter2 {
public Integer getNumber() {
System.out.println(Inter1.super.getNumber());
System.out.println(Inter2.super.getNumber());
return 23;
}
}

public class Java8Lab6 {
public static void main(String[] args) {
Hello h = new Hello();
System.out.println(h.getNumber());
}
}
```

- **From Java 8, an interface method is abstract only if it does not specify a default implementation.**
- **Default method can help us to avoid utility classes, such as all the Collections class method can be provided in the interfaces itself.**
- **Default method is also referred to as Defender Method or Virtual extension method.**
- **A default method cannot override a method from java.lang.Object.**

```
// INVALID
interface Inter1 {
default  public String toString() {
return "Hello";
}
}
```

## 2. Static Method in Interface

- Before Java 8, interface can't have static method.
- From Java 8, we can define static method inside the interface with the method implementation.
- Java 8 provides static methods be defined in interfaces to assist default methods.
- Java interface static method is visible to interface methods only.
- When you implement an interface that contains a static method, the static method is still part of the interface and not part of the implementing class.
- The static method can only be called through the interface name, not with the reference variable.
- Java interface static methods are good for providing utility methods, for example null check, collection sorting etc.
- To access the static method of interface always you need to use
  <interfaceName>.<staticMethodName>(<parameter>);
- Interface static method helps us to provide security by stopping implementation classes to override them.
- A method from java.lang.Object can't be defined as static in interface.

*JAVA8LAB7.JAVA*

```
interface Inter1 {
static void show() {
System.out.println("show in Inter1");
}
}
class Hello implements Inter1{ }
public class Java8Lab7 {
public static void main(String[] args) {
Inter1.show();
}
}
```

*JAVA8LAB8.JAVA*

```
interface Inter1 {
static void show() {
System.out.println("show in Inter1");
}
}
class Hello implements Inter1{ }
public class Java8Lab8 {
public static void main(String[] args) {
 Hello.show();      // INVALID
 Inter1 in = null;
 in.show();         // INVALID
 Inter1 in1 = new Hello();
 in1.show();        // INVALID
}
}
```

*JAVA8LAB.JAVA*

```
interface Inter1 {
static void show() {
System.out.println("show in Inter1");
}
default void process(){
System.out.println("process in Inter1");
show();
}
}
```

```
class Hello implements Inter1 {
static void  display() {
// show();
Inter1.show();
}
public void process(){
//show();
Inter1.show();
}
}
```

*JAVA8LAB9.JAVA*

```
interface Inter1 {
static void show() {
System.out.println("show in Inter1");
}
}
class Hello implements Inter1 {
public int show() {
System.out.println("show in Hello Class");
return 0;
}
}
class Hai implements Inter1 {
static public String show() {
System.out.println("show in Hai Class");
return null;
}
}
public class Java8Lab9{
public static void main(String[] args) {
Hello h = new Hello();
h.show();
Hai.show();
}
}
```

**Only one of abstract, default, or static permitted with the method defined in the interface.**

*JAVA8LAB.JAVA*

```
interface Inter1 {
default static void show() {
System.out.println("show in Inter1");
} }
```

*JAVA8LAB10.JAVA*

```
interface Inter1 {
static void show() {
System.out.println("show in Inter1");
}
}
class Hello implements Inter1 {
static int show() {
System.out.println("** show in Hello class **");
return 0;
}
}
public class Java8Lab10 {
public static void main(String[] args) {
Inter1.show();
Hello.show();
}
}
```

**Following are the valid definitation for the sub class**

**1)**

```
interface Inter1 {
static void show() { }
}
class Hello implements Inter1 {
int show() {
return 0;
}
}
```

**2)**

```
interface Inter1 {
static void show() { }
}
class Hello implements Inter1 {
void show() { }
}
```

## 3. New Date and Time API

- Initially Java had java.util.Date, java.util.Calendar class for managing the Data and Time in Java Program.
- They are mutable and not thread safe.
- It had many inconsistencies with mutability and deficient operations.
- There are various third party APIs like joda-time are used for date type.
- The new API introduces human readable and machine readable formats for Data and Time.
- The Date and Time object of new API are immutable.
- The Date and Time APIs can be mixed to create time instances easily.
- There are new classes/enums to support the complicated date operations.
- These classes are available in various packages.
- APIs from java.time package
    - DayOfWeek (enum)
    - Month (enum)
    - Year
    - YearMonth
    - MonthDay
    - LocalDate
    - LocalTime
    - LocalDateTime
    - Instant
    - Duration
    - Period

      etc
- APIs from java.time.format package
    - TextStyle (enum)
- APIs from java.time.temporal package
    - TemporalAdjuster (Functional interafce)
    - TemporalAdjusters (class)

## 1. DayOfWeek:

- The DayOfWeek enum contains seven constants for describing the days of the week.
- The integer values of the DayOfWeek constants range from 1 (Monday) to 7 (Sunday).
- This enum also provides some methods.

*JAVA8LAB11.JAVA*

```
import java.time.DayOfWeek;
import java.time.format.TextStyle;
import java.util.Locale;

public class Java8Lab11 {
public static void main(String[] args) {
DayOfWeek day  = DayOfWeek.MONDAY;
System.out.println("Day "+day);
System.out.println("Day "+day.getValue());
Locale loc = Locale.getDefault();
System.out.println("Full "+day.getDisplayName(TextStyle.FULL, loc));
System.out.println("Short "+day.getDisplayName(TextStyle.SHORT, loc));
System.out.println("Narrow "+day.getDisplayName(TextStyle.NARROW, loc));

DayOfWeek day2  = DayOfWeek.SUNDAY;
System.out.println("compareTo "+day.compareTo(day2));

System.out.println("Day "+day);
System.out.println("-2 "+day.minus(2));
System.out.println("+2 "+day.plus(2));
}
}
```

## 2. Month:

- The DayOfWeek enum contains seven constants for describing the days of the week.
- The Month enum contsina constants for the twelve months i.e JANUARY to DECEMBER.
- The integer value of each constant corresponds to the ISO range from 1 (January) to 12 (December).
- The Month enum also includes a number of methods.

*JAVA8LAB12.JAVA*

```
import java.time.Month;
import java.time.format.TextStyle;
import java.util.Locale;
public class Java8Lab12 {
public static void main(String[] args) {
Month mon = Month.FEBRUARY;
System.out.println("Month :"+mon);
System.out.println("Value :"+mon.getValue());
System.out.println("Max :"+mon.maxLength());
```

```
System.out.println("Min :"+mon.minLength());

Locale loc = Locale.getDefault();
System.out.println("Full :"+mon.getDisplayName(TextStyle.FULL, loc));
System.out.println("Short :"+mon.getDisplayName(TextStyle.SHORT, loc));
System.out.println("Narrow :"+mon.getDisplayName(TextStyle.NARROW, loc));
System.out.println("-2 :" +mon.minus(2));
System.out.println("+2 :" +mon.plus(2));
System.out.println("N of Days:" +mon.length(false));
System.out.println("N of Days (Leap Y):" +mon.length(true));
System.out.println("\nDay of Year :"+Month.JANUARY.firstDayOfYear(false));
System.out.println("Day of Year :"+Month.JANUARY.firstDayOfYear(true));
System.out.println("\nDay of Year :"+Month.MARCH.firstDayOfYear(false));
System.out.println("Day of Year :"+Month.MARCH.firstDayOfYear(true));
System.out.println("\nMon of Quater :"+Month.JANUARY.firstMonthOfQuarter());
System.out.println("Mon of Quater :"+Month.MAY.firstMonthOfQuarter());
System.out.println("Mon of Quater :"+Month.SEPTEMBER.firstMonthOfQuarter());
System.out.println("Mon of Quater :"+Month.DECEMBER.firstMonthOfQuarter());
}
}
```

## 3. LocalDate:

- **A LocalDate represents a year-month-day in the ISO calendar and is useful for representing a date without a time. You can use a LocalDate to track a significant event, such as a birth date or wedding date.**

*JAVA8LAB13.JAVA*

```
import java.time.LocalDate;

public class Java8Lab13 {
public static void main(String[] args) {
LocalDate date = LocalDate.of(2016, 2, 5);
System.out.println("YEAR :"+date.getYear());
System.out.println("MONTH :"+date.getMonth());
System.out.println("MONTH VALUE :"+date.getMonthValue());
System.out.println("DATE :"+date.getDayOfMonth());
System.out.println("DAYOF WEEK :"+date.getDayOfWeek());
System.out.println("DAYOF WEEK VALUE :"+date.getDayOfWeek().getValue());
System.out.println("DAY OF YEAR :"+date.getDayOfYear());
System.out.println("Month Len :"+date.lengthOfMonth());
System.out.println("Year Len :"+date.lengthOfYear());

LocalDate date1 = LocalDate.now();
System.out.println("\nYEAR :"+date1.getYear());
System.out.println("MONTH :"+date1.getMonth());
System.out.println("MONTH VALUE :"+date1.getMonthValue());
System.out.println("DATE :"+date1.getDayOfMonth());
```

```
System.out.println("DAYOF WEEK :"+date1.getDayOfWeek());
System.out.println("DAYOF WEEK VALUE :"+date1.getDayOfWeek().getValue());
System.out.println("DAY OF YEAR :"+date1.getDayOfYear());
System.out.println("Month Len :"+date1.lengthOfMonth());
System.out.println("Year Len :"+date1.lengthOfYear());
}
}
```

*JAVA8LAB14.JAVA*

```
import java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;

public class Java8Lab14 {
public static void main(String[] args) {
LocalDate date = LocalDate.now();
System.out.println(date);
LocalDate nextWedDate = date.with(TemporalAdjusters.next(DayOfWeek.WEDNESDAY));
System.out.println("nextWed Date :"+nextWedDate);
LocalDate lastWedDate = date.with(TemporalAdjusters.lastInMonth(DayOfWeek.WEDNESDAY));
System.out.println("lastWedDate:"+lastWedDate);
System.out.println("-2 Year :"+date.minusYears(2));
System.out.println("-2 Mon :"+date.minusMonths(4));
System.out.println("-2 Wk :"+date.minusWeeks(2));
System.out.println("-2 Day :"+date.minusDays(2));
System.out.println(date);
System.out.println("+2 Year :"+date.plusYears(2));
System.out.println("+2 Mon :"+date.plusMonths(2));
System.out.println("+2 Wk :"+date.plusWeeks(2));
System.out.println("+2 Day :"+date.plusDays(2));
System.out.println(date);
System.out.println(LocalDate.MIN);
System.out.println(LocalDate.MAX);
LocalDate currentDateInLosAngeles = LocalDate.now(ZoneId.of("America/Los_Angeles"));
LocalDate currentDateInLocalSystem = LocalDate.now(ZoneId.systemDefault());
System.out.println(currentDateInLosAngeles);
System.out.println(currentDateInLocalSystem);
System.out.println(date.withYear(2019));
System.out.println(date.withMonth(Month.DECEMBER.getValue()));
System.out.println(date.withYear(1999999999));     // Exception
}
}
```

## 4. LocalTime:

- **The LocalTime class is similar to the other classes whose names are prefixed with Local, but deals in time only. This class is useful for representing human-based time of day, such as movie times, or the opening and closing times of the local library.**

*JAVA8LAB15.JAVA*

```
import java.time.LocalTime;
import java.time.ZoneId;

public class Java8Lab15 {
public static void main(String[] args) {
LocalTime time = LocalTime.of(15, 23, 45);
System.out.println(time);
System.out.println("Hour :"+time.getHour());
System.out.println("Min :"+time.getMinute());
System.out.println("Sec :"+time.getSecond());
System.out.println("Nano Sec :"+time.getNano());

System.out.println("-2 Hour :"+time.minusHours(2));
System.out.println("-2 Min :"+time.minusMinutes(2));
System.out.println("-2 Sec :"+time.minusSeconds(2));

System.out.println("+2 Hour :"+time.plusHours(2));
System.out.println("+2 Min :"+time.plusMinutes(2));
System.out.println("+2 Sec :"+time.plusSeconds(2));
System.out.println(time);
System.out.println("with Hour-12 :"+time.withHour(12));
System.out.println("with Min-34 :"+time.withMinute(34));
System.out.println("with Sec-56 :"+time.withSecond(56));

System.out.println("MIN "+LocalTime.MIN);
System.out.println("MID "+LocalTime.MIDNIGHT);
System.out.println("NOON "+LocalTime.NOON);
System.out.println("MAX" +LocalTime.MAX);

System.out.println("now "+LocalTime.now());

LocalTime currentTimeInLosAngeles = LocalTime.now(ZoneId.of("America/Los_Angeles"));
LocalTime currentTimeInLocalSystem = LocalTime.now(ZoneId.systemDefault());

System.out.println(currentTimeInLosAngeles);
System.out.println(currentTimeInLocalSystem);

}
}
```

## 5. LocalDateTime:

- **This class handles both date and time, without a time zone.**
- **It is one of the core classes of the Date-Time API.**
- **This class is used to represent date (month-day-year) together with time (hour-minute-second-nanosecond) i.e combination of LocalDate with LocalTime.**

*JAVA8LAB16.JAVA*

```
import java.time.LocalDateTime;
import java.time.ZoneId;

public class Java8Lab16 {
public static void main(String[] args) {
LocalDateTime dateTime = LocalDateTime.now();
System.out.println(dateTime);

System.out.println("Year :"+dateTime.getYear());
System.out.println("Month :"+dateTime.getMonth());
System.out.println("Mon Value :"+dateTime.getMonthValue());
System.out.println("Date :"+dateTime.getDayOfMonth());
System.out.println("Hour :"+dateTime.getHour());
System.out.println("Min :"+dateTime.getMinute());
System.out.println("Sec :"+dateTime.getSecond());
System.out.println("Nano Sec :"+dateTime.getNano());

System.out.println("-2 Hour :"+dateTime.minusHours(2));
System.out.println("-2 Min :"+dateTime.minusMinutes(2));
System.out.println("-2 Sec :"+dateTime.minusSeconds(2));

System.out.println("+2 Hour :"+dateTime.plusHours(2));
System.out.println("+2 Min :"+dateTime.plusMinutes(2));
System.out.println("+2 Sec :"+dateTime.plusSeconds(2));
System.out.println(dateTime);
System.out.println("with Hour-12 :"+dateTime.withHour(12));
System.out.println("with Year-2019 :"+dateTime.withYear(2019));

LocalDateTime currentTimeInLosAngeles = LocalDateTime.now(ZoneId.of("America/Los_Angeles"));
LocalDateTime currentTimeInLocalSystem = LocalDateTime.now(ZoneId.systemDefault());

System.out.println(currentTimeInLosAngeles);
System.out.println(currentTimeInLocalSystem);

}
}
```

*JAVA8LAB17.JAVA*

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;
import java.time.temporal.ChronoField;
import java.time.temporal.ChronoUnit;

public class Java8Lab17 {
public static void main(String[] args) {
LocalDateTime dateTime = LocalDateTime.now();
LocalDate date = LocalDate.now();
LocalTime time = LocalTime.now();
LocalDateTime dateTime1 = date.atTime(time);
LocalDateTime dateTime2 = time.atDate(date);
System.out.println(dateTime);
System.out.println(dateTime1);
System.out.println(dateTime2);

System.out.println(dateTime.truncatedTo(ChronoUnit.HOURS));

LocalDate localDate = dateTime.toLocalDate();
LocalTime localTime = dateTime.toLocalTime();
System.out.println(localDate);
System.out.println(localTime);

System.out.println(dateTime);
System.out.println(dateTime.plus(2, ChronoUnit.HOURS));
System.out.println(dateTime.plus(2, ChronoUnit.YEARS));
System.out.println(dateTime.get(ChronoField.YEAR));
System.out.println(dateTime.get(ChronoField.HOUR_OF_AMPM));
System.out.println(dateTime.get(ChronoField.HOUR_OF_DAY));
}
}
```

## 6. Instant

- One of the core classes of the Date-Time API is the Instant class.
- It represents the start of a nanosecond on the timeline.
- This class is useful for generating a time stamp to represent machine time.
- A value returned from the Instant class counts time beginning from the first second of January 1, 1970 (1970-01-01T00:00:00Z) also called the EPOCH.
- An instant that occurs before the epoch has a negative value, and an instant that occurs after the epoch has a positive value.
- The constants
    - MIN representing the smallest possible (far past) instant,
    - MAX, representing the largest (far future) instant.

*JAVA8LAB18.JAVA*

```
import java.time.Instant;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.temporal.ChronoUnit;
import java.util.Date;

public class Java8Lab18 {
public static void main(String[] args) {
System.out.println(Instant.MIN);
System.out.println(Instant.MAX);
System.out.println(Instant.EPOCH);

Instant instant = Instant.now();
System.out.println(instant);

System.out.println(instant.minus(2, ChronoUnit.MILLIS));
System.out.println(instant.minusMillis(2));

System.out.println(instant.getEpochSecond());
long secondsFromEpoch = Instant.ofEpochSecond(0L).until(Instant.now(),ChronoUnit.SECONDS);
System.out.println(secondsFromEpoch);


long millisFromEpoch = Instant.ofEpochSecond(0L).until(Instant.now(),ChronoUnit.MILLIS);
System.out.println(millisFromEpoch);
Date dt = new Date();
System.out.println(dt.getTime());

LocalDateTime dateTime = LocalDateTime.ofInstant(instant, ZoneId.systemDefault());
LocalDateTime dateTimeUTC = LocalDateTime.ofInstant(instant, ZoneId.of("UTC"));
System.out.println(instant);
System.out.println(dateTimeUTC);
System.out.println(dateTime);

}
}
```

## 7. Duration

- A Duration is most suitable in situations that measure machine-based time.
- A Duration object is measured in seconds or nanoseconds and does not use date-based constructs such as years, months, and days.
- This class provides methods that convert to days, hours, and minutes.
- A Duration can have a negative value, if it is created with an end point that occurs before the start point.

*JAVA8LAB19.JAVA*

```
import java.time.Duration;
import java.time.Instant;
import java.time.temporal.ChronoUnit;

public class Java8Lab19 {
public static void main(String[] args) {
Instant t1 = Instant.now();
Instant t2 = Instant.parse("2017-03-05T10:15:30.00Z");

Duration d1=Duration.between(t1, t2);
System.out.println(d1.toDays());
System.out.println(d1.toHours());
System.out.println(d1.toMinutes());

long value = ChronoUnit.DAYS.between(t1,t2);
System.out.println(value);


Duration gap = Duration.ofSeconds(10);
Instant t3=t1.plus(gap);
System.out.println(t1);
System.out.println(t3);

long value1 = ChronoUnit.SECONDS.between(t1,t3);
System.out.println(value1);

}
}
```

## 8. Period

- This can be used to define an amount of time with date-based values (years, months, days).
- The Period class provides various get methods, such as getMonths, getDays, and getYears, so that you can extract the amount of time from the period.
- The total period of time is represented by all three units together: months, days, and years.
- To present the amount of time measured in a single unit of time, such as days, you can use the ChronoUnit.between method.

*JAVA8LAB20.JAVA*

```
import java.time.LocalDate;
import java.time.Month;
import java.time.Period;
import java.time.temporal.ChronoUnit;

public class Java8Lab20 {
public static void main(String[] args) {
LocalDateTime today = LocalDateTime.now();
LocalDateTime birthday = LocalDateTime.of(1987, Month.DECEMBER, 25,00,00,00);
Duration gap = Duration.between(birthday, today);
System.out.println(gap.toDays());

LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1987, Month.DECEMBER, 25);
Period p1 = Period.between(birthday, today);
long days1 = ChronoUnit.DAYS.between(birthday, today);
System.out.println("You are " + p1.getYears() + " years, " + p1.getMonths() +" months, and " + p1.getDays() +"
days old. (" + days1 + " days total)");

LocalDate nextBDay = birthday.withYear(today.getYear());
if (nextBDay.isBefore(today) || nextBDay.isEqual(today)) {
nextBDay = nextBDay.plusYears(1);
}

Period p2 = Period.between(today, nextBDay);
long days2 = ChronoUnit.DAYS.between(today, nextBDay);
System.out.println("There are " + p2.getMonths() + " months, and " +p2.getDays() + " days until your next
birthday. (" +days2 + " days total)");

}
}
```

## 4. Using this as Method Parameter

**From Java 8,**

- You can define the method parameter as this.
- The type of this parameter must be current class type.
- Only the first formal parameter can be declared explicitly as 'this'.
- When you are defining this as parameter then while invoking that method you should not provide any argument for this parameter.
- This can't be used for static method.

```
class Hello {
        static void show(Hello this) { } // INVALID
}
```

- this can't be used as parameter for the Constructor because while invoking constructor we don't have the current object of the class.

```
class Hello {
        Hello(Hello this) { }          // INVALID
}
```

- You can use this as parameter for the Inner class Constructor but it should be of type Outer class and this must be qualified with Outer class name.

```
class A {
        class B {
                B(B this) { }    // INVALID
        }
        class C{
                C(A this){}      // INVALID
        }
        class D{
                D(A A.this){}    // VALID
        }
}
```

*JAVA8LAB21.JAVA*

```
public class Java8Lab21 {
public static void main(String[] args) {
Hello h = new Hello();
h.show("MANISH");
}
}
```

```
class Hello {
int var = 99;
void show(String var) {
System.out.println("LOCAL: "+var);
System.out.println("INS: "+this.var);
}
}
```

```
// INVALID Before Java 7 but Valid from Java 8
class Hello {
int var = 99;
void show(Hello this,String var) {
System.out.println("LOCAL: "+var);
System.out.println("INS: "+this.var);
}
}

// INVALID
class Hello {
int var = 99;
void show(String var,Hello this) {
System.out.println("LOCAL: "+var);
System.out.println("INS: "+this.var);
}
}
```

*JAVA8LAB22.JAVA*

```
public class Java8Lab22 {
public static void main(String[] args) {
Hello h = new Hello();
h.show("MANISH");
// h.show(h,"MANISH");
}
}

class Hello {
int var = 99;
void show(Hello this,String var) {
System.out.println("LOCAL: "+var);
System.out.println("INS: "+this.var);
}
}
```

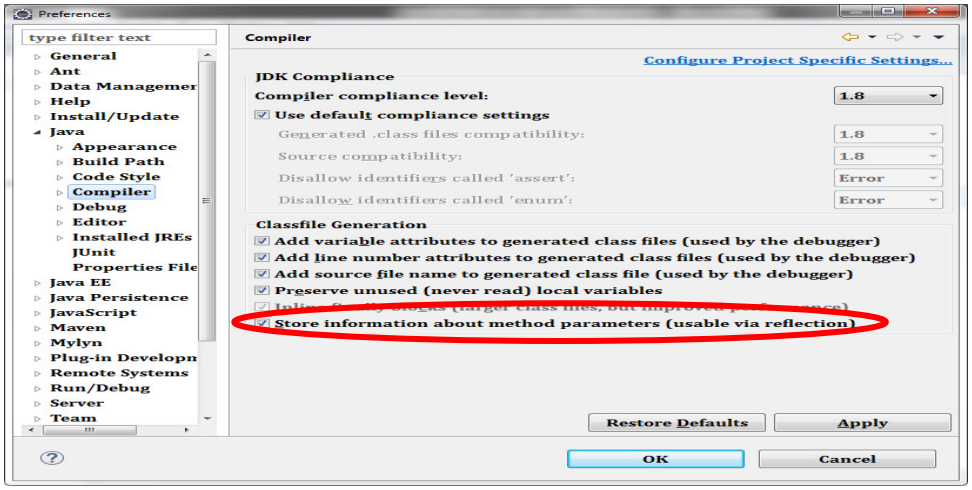
## 5. Accessing Parameter Name Using Reflection

- Using reflection, you can inspect the information of classes dynamically at runtime time.
- Upto Java 7, using reflection you can't access method parameter names. You can access only the type of parameter.
- From Java 8, you can access the name of parameter defined for the method.
- When you want to access the method parameters then you can use the following method from java.lang.reflect.Method class object.

  **Parameter [] getParameters()**
- With the java.lang.reflect.Parameter type object you can access the following methods

  **public boolean isNamePresent();**
  **public String getName();**
  **public boolean isVarArgs();**
  **etc**
- When you want to access the method parameter names using reflection then you need to provide the –parameters option while compiling.

  **javac -parameters Lab.java**
- When you work with Eclipse Luna Release (4.4.0) or higher then you need to Select the following CheckBox in Windows -> Preferences -> Java -> Compiler
  - Store information about method parameters (usable via reflection)




*JAVA8LAB23.JAVA*

```
// Compile with -parameters option
import java.lang.reflect.Method;
import java.lang.reflect.Parameter;
class Hello {
	void show(int sid, String name, String... email) { }
	void display(long val, boolean valid, char choice) { }
}
public class Java8Lab23 {
	public static void main(String[] args) {
		Class cl = Hello.class;
		Method ms[] = cl.getDeclaredMethods();
		for (Method m : ms) {
			String nm = m.getName();
			System.out.println("\nMethod :" + nm);
			Parameter params[] = m.getParameters();
			for(Parameter pr:params){
				boolean pre=pr.isNamePresent();
				String pnm=pr.getName();
				boolean isVarArgs=pr.isVarArgs();
				String type=pr.getType().getName();
				System.out.println(pre+"\t"+pnm+"\t"+type+"\t"+isVarArgs);
			}
			System.out.println();
		}
	}
}
```

## 6. Functional Programming

*JAVA8LAB24.JAVA*

```
import java.util.ArrayList;
import java.util.List;

public class Java8Lab24{
public static void main(String[] args) {
List<Student> students = new ArrayList<>();
students.add(new Student("Manish", "BLR", 45));
students.add(new Student("Sri", "BLR", 89));
students.add(new Student("Rahul", "BLR", 95));

students.add(new Student("Kumar", "HYD", 92));
students.add(new Student("Ranjan", "HYD", 56));
students.add(new Student("Manoj", "DEL", 89));
students.add(new Student("Dande", "DEL", 67));
students.add(new Student("Ranjan", "DEL", 45));

System.out.println(filterBLRStudent(students));
}

static List<Student> filterBLRStudent(List<Student> students) {
List<Student> result = new ArrayList<>();
for (Student student : students) {
if ("BLR".equals(student.getCity())) {
result.add(student);
}
}
return result;
}
}

class Student {
String name;                String city;                int totalMarks;

public Student(String name, String city, int totalMarks) {
this.name = name;                this.city = city;                this.totalMarks = totalMarks;
}

// Setter and Getter Method

@Override
public String toString() {
return "Name :" + name + " City :" + city + " Marks " + totalMarks;
}
}
```

*JAVA8LAB25.JAVA*

```
import java.util.ArrayList;
import java.util.List;

public class Java8Lab25{
public static void main(String[] args) {
List<Student> students = new ArrayList<>();
students.add(new Student("Manish", "BLR", 45));
students.add(new Student("Sri", "BLR", 89));
students.add(new Student("Rahul", "BLR", 95));

students.add(new Student("Kumar", "HYD", 92));
students.add(new Student("Nivas", "HYD", 69));
students.add(new Student("Ranjan", "HYD", 56));
students.add(new Student("Manoj", "DEL", 89));
students.add(new Student("Dande", "DEL", 89));
students.add(new Student("Ranjan", "DEL", 89));

System.out.println(filterStudentForCity(students,"BLR"));
System.out.println(filterStudentForCity(students,"DEL"));
}
static List<Student> filterStudentForCity(List<Student> students,String city) {
List<Student> result = new ArrayList<>();
for (Student student : students) {
if (city.equals(student.getCity())) {
result.add(student);
}
}
return result;
}
}
class Student {
String name;                String city;                int totalMarks;

public Student(String name, String city, int totalMarks) {
this.name = name;                this.city = city;                this.totalMarks = totalMarks;
}

// Setter and Getter Method

@Override
public String toString() {
return "Name :" + name + " City :" + city + " Marks " + totalMarks;
}
}
```

*JAVA8LAB26.JAVA*

```
import java.util.ArrayList;
import java.util.List;

public class Java8Lab26{
public static void main(String[] args) {
// Same as Java8Lab24

CityPredicate blrCityPredicate = new CityPredicate("BLR");
CityFailPredicate cityFailPredicate = new CityFailPredicate("HYD");

System.out.println(filterStudent(students,blrCityPredicate));
System.out.println(filterStudent(students,cityFailPredicate));
}
static List<Student> filterStudent(List<Student> students,Predicate predicate) {
List<Student> result = new ArrayList<>();
for (Student student : students) {
if (predicate.test(student)) {
result.add(student);
}
}
return result;
}
}

interface Predicate{
boolean test(Student student);
}

class CityPredicate implements Predicate{
private String city;
CityPredicate(String city){   this.city = city;   }
public boolean test(Student student) {
return city.equals(student.getCity());
}
}
class CityFailPredicate implements Predicate{
private String city;

CityFailPredicate(String city){   this.city = city;  }

public boolean test(Student student) {
return city.equals(student.getCity()) && student.getTotalMarks() < 60;
}
}

class Student {
// Same as Java8Lab24
}
```

*JAVA8LAB27.JAVA*

```
import java.util.ArrayList;
import java.util.List;

public class Java8Lab27{
public static void main(String[] args) {
// Same as Java8Lab24
List<Student> result=filterStudent(students,(Student student) -> "BLR".equals(student.getCity()));
System.out.println(result);
result=filterStudent(students,(Student student) -> "HYD".equals(student.getCity()));
System.out.println(result);
result=filterStudent(students,(Student student) -> "HYD".equals(student.getCity()) &&
student.getTotalMarks()<60);
System.out.println(result);
}

static <T> List<T> filterStudent(List<T> list,Predicate<T> predicate) {
List<T> result = new ArrayList<>();
for (T type : list) {
if (predicate.test(type)) {
result.add(type);
}
}
return result;
}
}
interface Predicate<T>{
boolean test(T type);
}
class Student {
// Same as Java8Lab24
}
```

## We are trying to understand the approach to program.

- **Declarative programming style has some certain improvements**
  - **No need to deal with mutable variables**
  - **Implicit iteration i.e no need to iterate explicitly**
  - **Less cluter in code**
  - **Less error in code**
  - **Freedom from low level operations**
  - **Easy to maintain and support**
  - **Java 8 provides specialized operation like map, reduce, filter etc.**
  - **These operations composed together to form a business logic implementation.**
  - **Java 8 allows programmers to switch the context between serial and parallel execution very easily**

- Java supports imperative programming.
- Java 8 is adapting the functional programming.
- It help in coading in declaring manner then usual.

**Imperative Code**

```
public static void findNameImperative(final List<String> names) {
        boolean found = false;
        for (String name : names) {
                if (name.equals("Manish")) {
                        found = true;
                        break;
                }
        }
        if (found) {
                System.out.println("Manish Found");
        }
}
```

**Declarative Code**

```
public static void findNameDeclarative(final List<String> names) {
        if (names.contains("Manish")) {
                System.out.println("Manish Found");
        }
}
```

**Imperative Code**

```
public static void totalFactorrPricesImperative(final List<Double> prices) {
        Double totalFactorPrice = 0.0;
        for (Double price : prices) {
                if (price.compareTo(23.0) > 0) {
                        totalFactorPrice += price * (1 + Double.valueOf(0.10));
                }
        }
        System.out.println(totalFactorPrice.intValue());
}
```

**Declarative Code**

```
public static void totalFactorrPricesDeclarative(final List<Double> prices) {
Double totalFactorPrice=
        prices.stream().filter(price -> price > 23)
                        .map(price1 -> price1* (1 + Double.valueOf(0.10)))
                        .reduce(Double.valueOf(0),(a,b)-> a+b);
System.out.println(totalFactorPrice.intValue());
}
```