DIGITAL TRANSFORMATION CONSULTING & SERVICES

Java SE 11 Developer Certification 1Z0-819

Ministério da Justiça e Segurança Pública | MJSP

Meta

Digital. Simple. Human.

Apresentação - Unidade Curricular

Unidade Curricular: Java SE 11 Developer Certification 1Z0-819
Carga Horária: 24 horas
Objetivo geral: Permitir que desenvolvedores, profissionais de TI sejam referência desta tecnologia, testem e implementem

Proposta de trabalho

- Enfoque teórico e prático com uso de muitos exercícios, realizados em sala e em casa para fixação;
- Desenvolvimento de projetos com foco no mercado e com o uso da tecnologia estudada.

Planejamento

Semana 1
Introdution to Java
Primitive Types, Operators and Flow Control Statements
Text, Date, Time and Numeric Objects
Classes and Objects

Semana 2
Improved Class Design
Inheritance
Interfaces

Semana 3
Arrays and Loops
Collections
Java Streams API

Planejamento

Semana 4

Handle Exceptions and Fix Bugs
Java IO API
Java Concurrency and Multithreading
Java Modules

Cronograma – Aula

Aula	Conteúdo Programático	Carga Horária
01-06	Semana 1 (segunda, quarta)	6 horas
07-12	Semana 2 (segunda, quarta)	6 horas
Prática de Exercícios		
DOJO (2h)		
13-19	Semana 3 (segunda, quarta)	6 horas
20-24	Semana 4 (segunda, quarta)	6 horas
Prática de Exercícios		
DOJO (2h)		
Prática de Exercícios		
Test Killer		
Certificação		

Prova

Número de questões: 50
Score: 68% (34 questões)
Tempo: 90 minutos
Voucher: R$ 1.292,00 (6 meses)

Java SE 11 Developer Certification 1Z0-819

[Text, Date, Time and Numeric Objects]

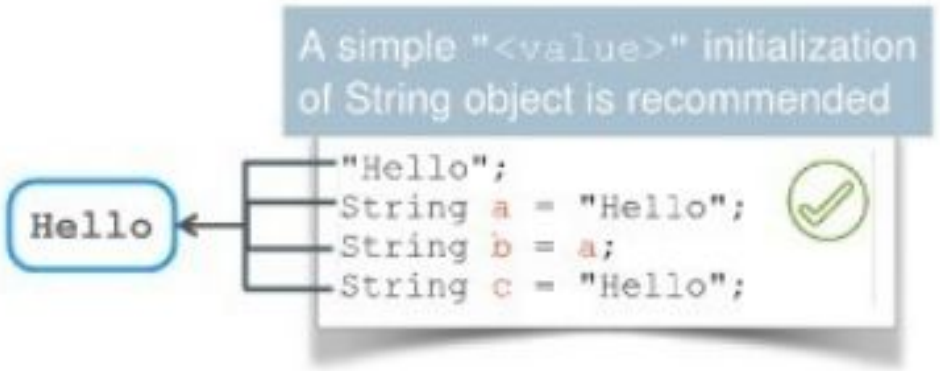
Meta

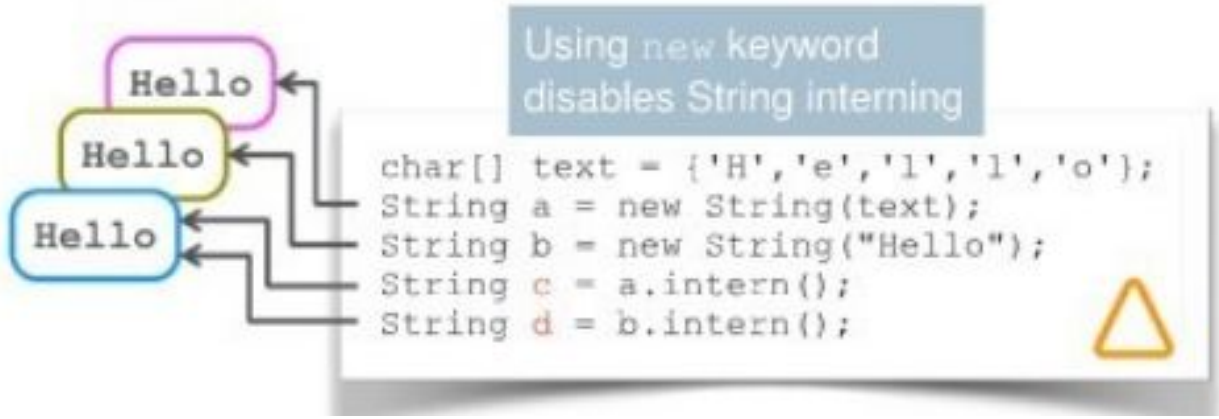
Digital. Simple. Human.

Inicialização de string

- java.lang.String é a classe que representa uma sequência de caracteres
- String é uma classe **não é um tipo primitivo**. Sua instância pode representar uma sequência de caracteres
- Criação: use **new** | Uso: “ ” (aspas duplas)
- Para otimização de memória um objeto do tipo String usa o método **interning**. (internamento).
- O internamento de strings acelera as comparações de strings, que às vezes são um gargalo de desempenho em aplicativos (como compiladores e runtimes de linguagem de programação dinâmica) que dependem muito de matrizes associativas com chaves de string para pesquisar os atributos e métodos de um objeto. Sem internar, comparar duas strings distintas pode envolver examinar todos os caracteres de ambas.

Inicialização de string





Operações em string

public class String

	String(String s)	*create a string with the same value as* `s`
	String(char[] a)	*create a string that represents the same sequence of characters as in* `a[]`
int	length()	*number of characters*
char	charAt(int i)	*the character at index* `i`
String	substring(int i, int j)	*characters at indices* `i` *through* `(j-1)`
boolean	contains(String substring)	*does this string contain* `substring`*?*
boolean	startsWith(String prefix)	*does this string start with* `prefix`*?*
boolean	endsWith(String postfix)	*does this string end with* `postfix`*?*
int	indexOf(String pattern)	*index of first occurrence of* `pattern`
int	indexOf(String pattern, int i)	*index of first occurrence of* `pattern` *after* `i`
String	concat(String t)	*this string, with* `t` *appended*
int	compareTo(String t)	*string comparison*
String	toLowerCase()	*this string, with lowercase letters*
String	toUpperCase()	*this string, with uppercase letters*
String	replace(String a, String b)	*this string, with* `a`*s replaced by* `b`*s*
String	trim()	*this string, with leading and trailing whitespace removed*
boolean	matches(String regexp)	*is this string matched by the regular expression?*
String[]	split(String delimiter)	*strings between occurrences of* `delimiter`
boolean	equals(Object t)	*is this string's value the same as* `t`*'s?*
int	hashCode()	*an integer hash code*


```
String s = "";
s=1+1+"u";   // s is 2u
s="u"+1+1;   // s is u11
s="u"+(1+1); // s is u2
```

Operações em string

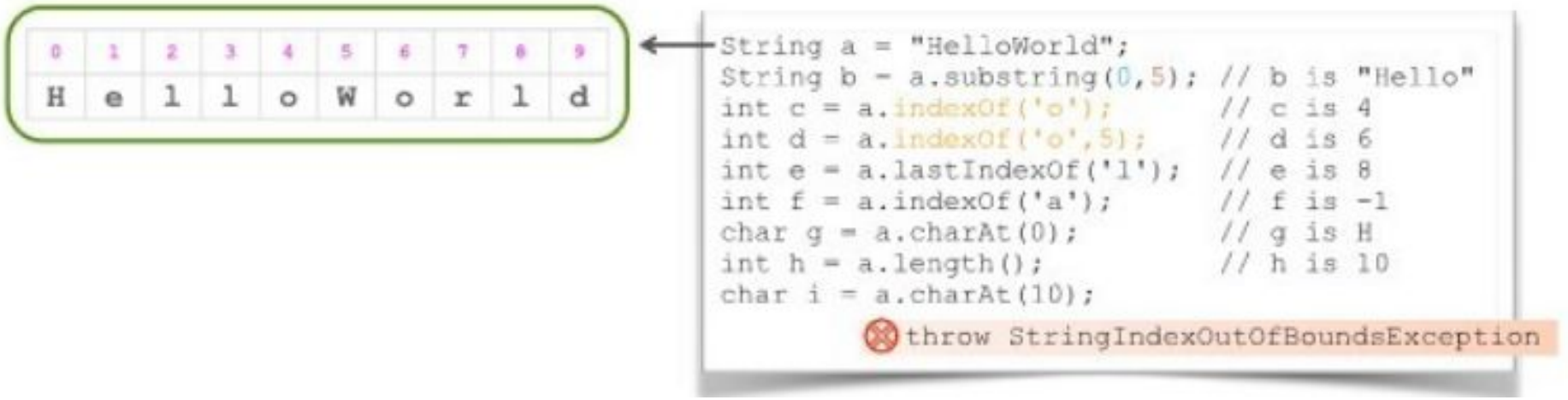
```
String myString = "Both".concat(" fickle")
.concat(" dwarves")
.concat(" jinx")
.concat(" my")
.concat(" pig")
.concat(" quiz");
```

```
String myString = String.format("%s %s %.2f %s %s, %s...", "I",
"ate",
2.5056302,
"blueberry",
"pies",
"oops");
```

```
String[] strings = {"I'm", "running", "out", "of", "pangrams!"};
String myString = String.join(" ", strings);
```

Indexação de string

- String contém um sequência indexada por um inteiro (**integer**)
- O índice da string começa em zero (0).
- Substring retorna uma parte da string. O início do índice está incluído no resultado, porém o último índice não.
- Se um substring não for encontrada, o indexOf retorna -1
- indexOf e lastIndexOf podem ser sobreescritas
- Índice inválido é retornada uma exceção.

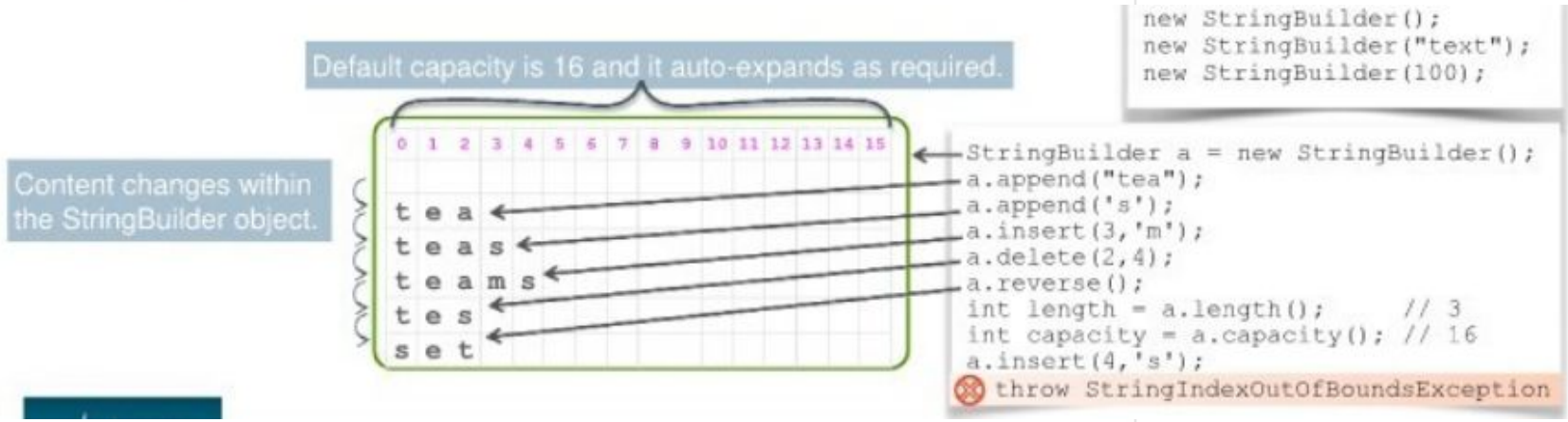


StringBuilder: Introdução

- java.lang.StringBuilder
- StringBuilder os objetos são mutáveis, permite modificações dos valores armazenados
- A modificação dos textos com StringBuilder reduz o número de objetos do tipo String que precisam ser criados.
- Alguns métodos são identicos a classe String: substring, indexOf, charAt
- Metódos adicionais: append, insert, delete, reverse.
- Objetos devem ser instanciados com a palavra **new**.
- Uma instância StringBuilder é predefinida pelo seu conteúdo ou por sua capacidade (capacity).

StringBuilder: Introdução

```java
public final class StringBuilder extends AbstractStringBuilder implements Serializable, Comparable<StringBuilder>, CharSequence {
    static final long serialVersionUID = 4383685877147921099L;

    @HotSpotIntrinsicCandidate
    public StringBuilder() {
        super( capacity: 16);
    }

    @HotSpotIntrinsicCandidate
    public StringBuilder(int capacity) { super(capacity); }

    @HotSpotIntrinsicCandidate
    public StringBuilder(String str) {
        super( capacity: str.length() + 16);
        this.append(str);
    }

    public StringBuilder(CharSequence seq) {
        this( capacity: seq.length() + 16);
        this.append(seq);
    }
```



StringBuilder: Introdução

- Diferenças entre StringBuilder e **StringBuffer** em Java. Simplificando, **StringBuilder** foi introduzido no Java 1.5 como um substituto para StringBuffer.
- **StringBuffer** é sincronizado e, portanto, thread-safe.

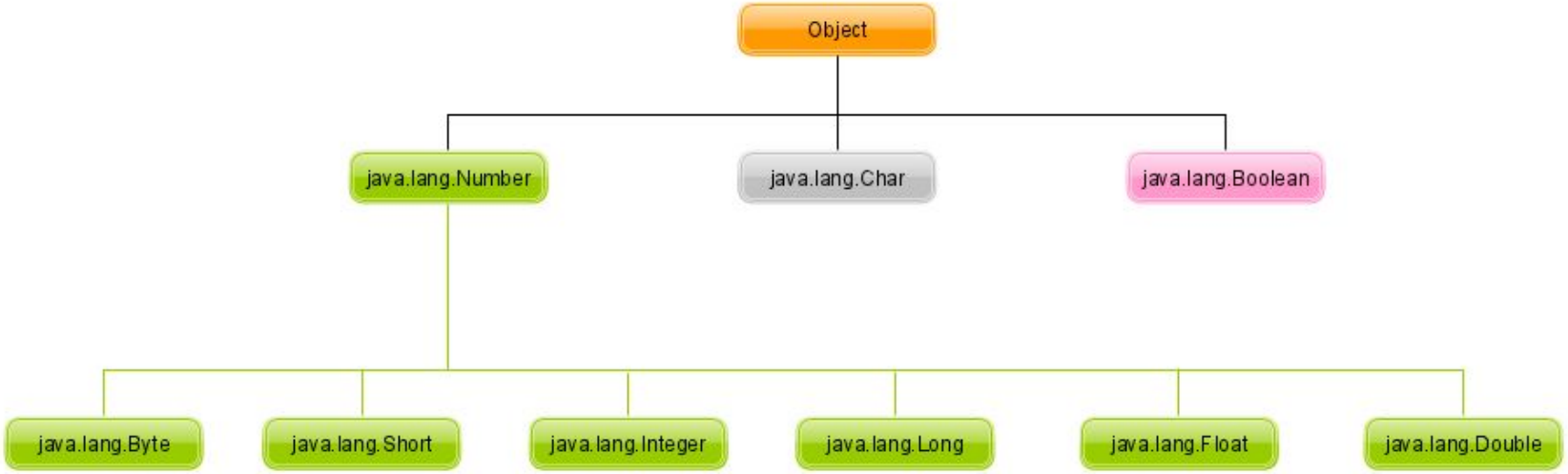
- O **StringBuilder** é compatível com a API StringBuffer , mas sem garantia de sincronização. Por não ser uma implementação thread-safe, é mais rápida e é recomendável usá-la em locais onde não há necessidade de thread safety.

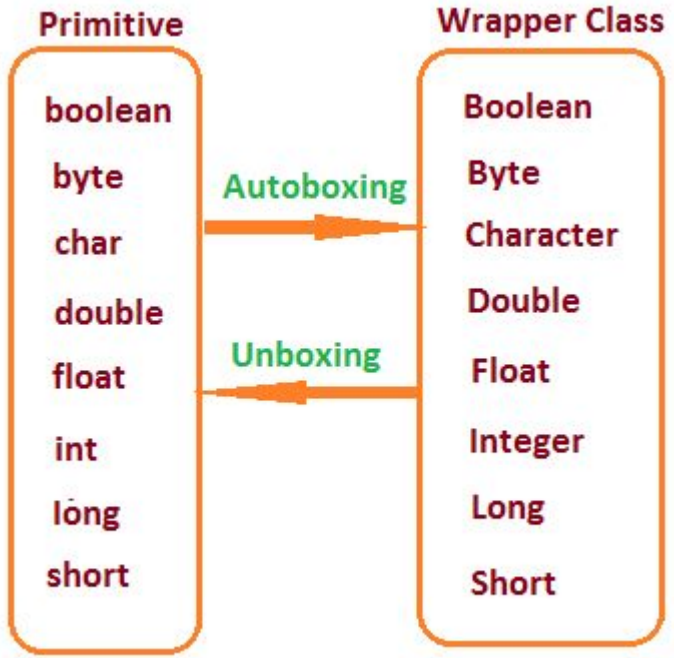
```java
public class TesteStringBuilder {

    // Create a StringBuilder object
    // using StringBuilder() constructor
    public static void main(String[] argv) {
        StringBuilder str = new StringBuilder();


        str.append("Treinamento Oracle Certificação ");

        // print string
        System.out.println("String = " + str.toString());

        // create a StringBuilder object
        // using StringBuilder(CharSequence) constructor
        StringBuilder str1
                = new StringBuilder("AAAABBBCCCC");

        // print string
        System.out.println("String1 = " + str1.toString());

        // create a StringBuilder object
        // using StringBuilder(capacity) constructor
        StringBuilder str2 = new StringBuilder( capacity: 10);

        // print string
        System.out.println("String2 capacity = "
                + str2.capacity());

        // create a StringBuilder object
        // using StringBuilder(String) constructor
        StringBuilder str3
                = new StringBuilder(str1.toString());

        // print string
        System.out.println("String3 = " + str3.toString());

    }
}
```

Meta

Wrapper Classes para tipos primitivos

A classe Wrapper em Java converte ou agrupa tipos de dados primitivos como objetos. Isso significa que podemos converter valores primitivos em objetos e vice-versa. Existem 8 tipos de dados primitivos que possuem uma classe de invólucro equivalente. Essas classes de wrapper estendem a classe Number, que é a classe pai.





Exemplos

```
int a = 42;
Integer b = Integer.valueOf(a);
int c = b.intValue();
b = a;
c = b;
String d = "12.25";
Float e = Float.valueOf(d);
float f = d.parseFloat(d);
String b = String.valueOf(a);
Short.MIN_VALUE;
Short.MAX_VALUE;
```

Autoboxing

O Autoboxing converte automaticamente os valores primitivos em suas respectivas classes de wrapper. Por exemplo, podemos converter int em Integer, etc.

```
Run | Debug
public static void main(String[] args) {

        int i = 50;
        Integer it = i;
        System.out.println("Integer: " + it);

        char c = 'j';
        Character ch = c;
        System.out.println("Character: " + ch);
    }


}
```

Unboxing

Unboxing é o processo reverso de autoboxing no qual podemos converter automaticamente objetos da classe wrapper em seus tipos de dados primitivos correspondentes

```
public class UnboxingExample {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Integer integer = new Integer(100);
        int i = integer; // Unboxing

        display(new Boolean(false)); // Unboxing
    }

    public static void display(boolean status) {
        System.out.println("Status :: " + status);
    }

}
```

Representando números usando bigdecimal

- java.math.BigDecimal
- Decimal com precisão exata
- Todos os wrappers primitivos e BigDecimal, são imutáveis e signed, ou seja, não pode ser alterados e são representados por números positivos e negativos
- BigDecimal tem uma precisão arbritária, **Double** tem uma precisão limitada a um numero binário de 64 bits.
- Operações de scale e round
- Operações aritméticas: add, substract, divide, mutiply, remainder.
- Pela precisão, utilizada para valores comerciais (fiscal, taxas, monetários).

```
BigDecimal price = BigDecimal.valueOf(12.99);
BigDecimal taxRate = BigDecimal.valueOf(0.2);
BigDecimal tax = price.multiply(taxRate);                    // tax is 2.598
price = price.add(tax).setScale(2,RoundingMode.HALF_UP); // price is 15.59
```

Meta

Method Chaining

- Método do encadiamento
- Técnica que toda operação retorna um objeto.

```
String s1 = "Hello";
Strung s2 = s1.concat("World").substring(3,6); // s2 is "loW"

BigDecimal price = BigDecimal.valueOf(12.99);
BigDecimal taxRate = BigDecimal.valueOf(0.2);
BigDecimal tax = price.multiply(taxRate);                    // tax is 2.598
price = price.add(tax).setScale(2,RoundingMode.HALF_UP); // price is 15.59
```

```
BigDecimal taxedPrice = price.add(tax);
price = taxedPrice.setScale(2,RoundingMode.HALF_UP);
```

Method Chaining

```
class A {

    private int a;
    private float b;

    A() { System.out.println("Calling The Constructor"); }

    int setint(int a)
    {
        this.a = a;
        return this.a;
    }

    float setfloat(float b)
    {
        this.b = b;
        return this.b;
    }

    void display()
    {
        System.out.println("Display=" + a + " " + b);
    }
}

// Driver code
public class Example {
    public static void main(String[] args)
    {
        // This will return an error as
        // display() method needs an object but
        // setint(10) is returning an int value
        // instead of an object reference
        new A().setint(10).display();
    }
}
```

```
class A {

    private int a;
    private float b;

    A() { System.out.println("Calling The Constructor"); }

    public A setint(int a)
    {
        this.a = a;
        return this;
    }

    public A setfloat(float b)
    {
        this.b = b;
        return this;
    }

    void display()
    {
        System.out.println("Display=" + a + " " + b);
    }
}

// Driver code
public class Example {
    public static void main(String[] args)
    {
        // This is the "method chaining".
        new A().setint(10).setfloat(20).display();
    }
}
```

Meta

Local Date and Time Values

- java.time
- LocalDate, LocalTime, LocalDateTime

Local Date and Time Values

- A classe Java LocalDate é uma classe imutável que representa Date com um formato padrão de aaaa-mm-dd. Ele herda a classe Object e implementa a interface ChronoLocalDate

Local Date and Time Values

```java
import java.time.LocalDate;
public class LocalDateExample1 {
  public static void main(String[] args) {
    LocalDate date = LocalDate.now();
    LocalDate yesterday = date.minusDays(1);
    LocalDate tomorrow = yesterday.plusDays(2);
    System.out.println("Today date: "+date);
    System.out.println("Yesterday date: "+yesterday);
    System.out.println("Tomorrow date: "+tomorrow);
  }
}
```

```
Today date: 2017-01-13
Yesterday date: 2017-01-12
Tomorrow date: 2017-01-14
```

```java
import java.time.LocalDate;
public class LocalDateExample2 {
  public static void main(String[] args) {
    LocalDate date1 = LocalDate.of(2017, 1, 13);
    System.out.println(date1.isLeapYear());
    LocalDate date2 = LocalDate.of(2016, 9, 23);
    System.out.println(date2.isLeapYear());
  }
}
```

```
false
true
```

```java
import java.time.*;
public class LocalDateExample3 {
  public static void main(String[] args) {
    LocalDate date = LocalDate.of(2017, 1, 13);
    LocalDateTime datetime = date.atTime(1,50,9
    System.out.println(datetime);
  }
}
```

```
2017-01-13T01:50:09
```

Local Date and Time Values

```java
import java.time.LocalDate;
import java.time.format.DateTimeFormatter;
public class LocalDateExample4
{
   public static void main(String ar[])
   {
      // Converting LocalDate to String
      // Example 1
      LocalDate d1 = LocalDate.now();
      String d1Str = d1.format(DateTimeFormatter.ISO_DATE);
      System.out.println("Date1 in string :  " + d1Str);
      // Example 2
      LocalDate d2 = LocalDate.of(2002, 05, 01);
      String d2Str = d2.format(DateTimeFormatter.ISO_DATE);
      System.out.println("Date2 in string :  " + d2Str);
      // Example 3
      LocalDate d3 = LocalDate.of(2016, 11, 01);
      String d3Str = d3.format(DateTimeFormatter.ISO_DATE);
      System.out.println("Date3 in string :  " + d3Str);
   }
}
```

```
Date1 in string :  2021-09-13
Date2 in string :  2002-05-01
Date3 in string :  2016-11-01
```

Local Date and Time Values

```java
import java.time.LocalDate;
// String to LocalDate in java 8
public class LocalDateExample5
{
   public static void main(String ar[])
   {
      // Example 1
      String dInStr = "2011-09-01";
      LocalDate d1 = LocalDate.parse(dInStr);
      System.out.println("String to LocalDate : " + d1);
      // Example 2
      String dInStr2 = "2015-11-20";
      LocalDate d2 = LocalDate.parse(dInStr2);
      System.out.println("String to LocalDate : " + d2);
   }
}
```

```
String to LocalDate : 2011-09-01
String to LocalDate : 2015-11-20
```

LocalTime

- A classe Java LocalTime é uma classe imutável que representa a hora com um formato padrão de hora-minuto-segundo. Ele herda a classe Object e implementa a interface Comparable.

```java
import java.time.LocalTime;
public class LocalTimeExample1 {
  public static void main(String[] args) {
    LocalTime time = LocalTime.now();
    System.out.println(time);
  }
}
```

```
15:19:47.459
```

```java
import java.time.LocalTime;
public class LocalTimeExample2 {
  public static void main(String[] args) {
    LocalTime time = LocalTime.of(10,43,12);
    System.out.println(time);
  }
}
```

```
10:43:12
```

LocalTime

```java
import java.time.LocalTime;
public class LocalTimeExample3 {
  public static void main(String[] args) {
    LocalTime time1 = LocalTime.of(10,43,12);
    System.out.println(time1);
    LocalTime time2=time1.minusHours(2);
    LocalTime time3=time2.minusMinutes(34);
    System.out.println(time3);
  }
}
```

```
10:43:12
08:09:12
```

```java
import java.time.LocalTime;
public class LocalTimeExample4 {
  public static void main(String[] args) {
    LocalTime time1 = LocalTime.of(10,43,12);
    System.out.println(time1);
    LocalTime time2=time1.plusHours(4);
    LocalTime time3=time2.plusMinutes(18);
    System.out.println(time3);
  }
}
```

```
10:43:12
15:01:12
```

```java
import java.time.*;
import java.time.temporal.ChronoUnit;
public class LocalTimeExample5 {
  public static void main(String... args) {
    ZoneId zone1 = ZoneId.of("Asia/Kolkata");
    ZoneId zone2 = ZoneId.of("Asia/Tokyo");
    LocalTime time1 = LocalTime.now(zone1);
    System.out.println("India Time Zone: "+time1);
    LocalTime time2 = LocalTime.now(zone2);
    System.out.println("Japan Time Zone: "+time2);
    long hours = ChronoUnit.HOURS.between(time1, time2);
    System.out.println("Hours between two Time Zone: "+hours);
    long minutes = ChronoUnit.MINUTES.between(time1, time2);
    System.out.println("Minutes between two time zone: "+minutes);
  }
}
```

```
India Time Zone: 14:56:43.087
Japan Time Zone: 18:26:43.103
Hours between two Time Zone: 3
Minutes between two time zone: 210
```

Meta

LocalDateTime

- A classe java.time.LocalDateTime , introduzida no Java 8 Date Time API , representa um objeto de data e hora sem um fuso horário geralmente visto como ' ano-mês-dia-hora-minuto-segundo '. Ele representa um instante na linha do tempo local com precisão de nanossegundos, por exemplo 2007-12-03T10:15:30:55.000000.

- Podemos usar as instâncias LocalDateTime representar os timestamps sem a necessidade do fuso horário ou referência de deslocamento. Se precisarmos de um carimbo de data/hora em uma zona específica, devemos usar a instância ZonedDateTime .

LocalDateTime

- Use o método now() para obter a data e hora local atual. Observe que podemos obter o timestamp local atual em outra zona passando o id da zona.

```
LocalDateTime now = LocalDateTime.now();

//Current timestamp in UTC
LocalDateTime utcTimestamp = LocalDateTime.now(ZoneId.of("UTC"));
```

- Criar LocalDateTime com valores

```
//Nonoseconds precision
LocalDateTime localDateTime1 =
        LocalDateTime.of(2019, 03, 28, 14, 33, 48, 640000);

//Using Month Enum
LocalDateTime localDateTime2 =
        LocalDateTime.of(2019, Month.MARCH, 28, 14, 33, 48, 000000);

//Seconds level precision
LocalDateTime localDateTime3 =
        LocalDateTime.of(2019, Month.MARCH, 28, 14, 33, 48);

//Minutes level precision
LocalDateTime localDateTime4 =
        LocalDateTime.of(2019, Month.MARCH, 28, 14, 33);
```

LocalDateTime

- Combinar LocalDate e LocalTime

```
//local date + local time
LocalDate date = LocalDate.of(2109, 03, 28);
LocalTime time = LocalTime.of(10, 34);

LocalDateTime localDateTime5 = LocalDateTime.of(date, time);
```

LocalDateTime

- Analisando uma String para LocalDateTime

A `LocalDateTime`classe tem dois métodos **parse()** sobrecarregados para converter a hora na string para a instância *LocalDateTime* .

```
parse(CharSequence text)    //1

parse(CharSequence text, DateTimeFormatter formatter)    //2
```

- Use o primeiro método se a string contiver tempo no `ISO_LOCAL_DATE_TIME`padrão, ou seja `2019-03-27T10:15:30`. Este é **o padrão padrão de LocalDateTime** em Java.
- Para qualquer outro padrão de data e hora, precisamos usar o segundo método onde passamos a hora como string, bem como um DateTimeFormatter que representa o padrão dessa string de data e hora.

```
//1 - default time pattern
String time = "2019-03-27T10:15:30";
LocalDateTime localTimeObj = LocalDateTime.parse(time);

//2 - specific date time pattern
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss a");
String time1 = "2019-03-27 10:15:30 AM";
LocalDateTime localTimeObj1 = LocalDateTime.parse(time1, formatter);
```

Meta

LocalDateTime

- Formatando LocalDateTime
- Use LocalDateTime.format(DateTimeFormatter)o método para formatar um LocalDateTime para a representação de string desejada.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss a");

LocalDateTime now = LocalDateTime.now();

String dateTimeString = now.format(formatter);  //2019-03-28 14:47:33 PM
```

LocalDateTime

- Modificando LocalDateTime
  - plusYears()
  - plusMonths()
  - plusDays()
  - plusHours()
  - plusMinutes()
  - plusSeconds()
  - plusNanos()
  - minusYears()
  - minusMonths()
  - minusDays()
  - minusHours()
  - minusMinutes()
  - minusSeconds()
  - minusNanos()

```
LocalDateTime now = LocalDateTime.now();

//3 hours later
LocalDateTime localDateTime1 = now.plusHours(3);

//3 minutes earliar
LocalDateTime localDateTime2 = now.minusMinutes(3);

//Next year same time
LocalDateTime localDateTime2 = now.plusYears(1);

//Last year same time
LocalDateTime localDateTime2 = now.minusYears(1);
```

LocalDateTime

- A classe Java LocalDateTime é um objeto de data e hora imutável que representa uma data e hora, com o formato padrão como aaaa-MM-dd-HH-mm-ss.zzz. Ele herda a classe de objeto e implementa a interface ChronoLocalDateTime.

```java
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class LocalDateTimeExample1 {
   public static void main(String[] args) {
      LocalDateTime now = LocalDateTime.now();
      System.out.println("Before Formatting: " + now);
      DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
      String formatDateTime = now.format(format);
      System.out.println("After Formatting: " + formatDateTime);
   }
}
```

```
Before Formatting: 2017-01-13T17:09:42.411
After Formatting: 13-01-2017 17:09:42
```

LocalDateTime

```java
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class LocalDateTimeExample2 {
  public static void main(String[] args) {
    LocalDateTime datetime1 = LocalDateTime.now();
    DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");
    String formatDateTime = datetime1.format(format);
    System.out.println(formatDateTime);
  }
}
```

```
14-01-2017 11:42:32
```

```java
import java.time.LocalDateTime;
import java.time.temporal.ChronoField;
public class LocalDateTimeExample3 {
  public static void main(String[] args) {
    LocalDateTime a = LocalDateTime.of(2017, 2, 13, 15, 56);
    System.out.println(a.get(ChronoField.DAY_OF_WEEK));
    System.out.println(a.get(ChronoField.DAY_OF_YEAR));
    System.out.println(a.get(ChronoField.DAY_OF_MONTH));
    System.out.println(a.get(ChronoField.HOUR_OF_DAY));
    System.out.println(a.get(ChronoField.MINUTE_OF_DAY));
  }
}
```

```
1
44
13
15
956
```

LocalDateTime

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class LocalDateTimeExample4 {
public static void main(String[] args) {
  LocalDateTime datetime1 = LocalDateTime.of(2017, 1, 14, 10, 34);
  LocalDateTime datetime2 = datetime1.minusDays(100);
  System.out.println("Before Formatting: " + datetime2);
  DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm");
  String formatDateTime = datetime2.format(format);
  System.out.println("After Formatting: " + formatDateTime );
}
}
```

```
Before Formatting: 2016-10-06T10:34
After Formatting: 06-10-2016 10:34
```

```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
public class LocalDateTimeExample5 {
public static void main(String[] args) {
  LocalDateTime datetime1 = LocalDateTime.of(2017, 1, 14, 10, 34);
  LocalDateTime datetime2 = datetime1.plusDays(120);
  System.out.println("Before Formatting: " + datetime2);
  DateTimeFormatter format = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:
  String formatDateTime = datetime2.format(format);
  System.out.println("After Formatting: " + formatDateTime );
}
}
```

```
Before Formatting: 2017-05-14T10:34
After Formatting: 14-05-2017 10:34
```

Meta

Instantes, Duração e Períodos

- java.time.Duration, representa o tempo em nanosegundos
- java.time.Period, representa o tempo em dias ou anos
- java.time.Instant, time-stamp (tempo presente)

```
LocalDate today = LocalDate.now();
LocalDate foolsDay = LocalDate.of(2019, Month, APRIL, 1);
Instant timeStamp = Instant.now();
int nanoSecondsFromLastSecond = timeStamp.getNano();
Period howLong = Period.between(foolsDay, today);
Duration twoHours = Duration.ofHours(2);
long seconds = twoHours.minusMinutes(15).getSeconds();
int days = howLong.getDays();
```

ZonedDate and Time

- java.time.ZoneDateTime
  - representa data e tempo de acordo com o “time zone”
  - Geralmente manipulado por LocalDateTime
  - Pode prover uma zona específica

```
ZoneId london = ZoneId.of("Europe/London");
ZoneId la   = ZoneId.of("America/Los_Angeles");
LocalDateTime someTime = LocalDateTime.of(2019,Month.APRIL,1,07,14);
ZonedDateTime londonTime = ZonedDateTime.of(someTime, london);
ZonedDateTime laTime = londonTime.withZoneSameInstant(la);
```

- java.time.ZoneId

```
ZoneId.of("America/Los_Angeles");
ZoneId.of("GMT+2");
ZoneId.of("UTC-05:00");
ZoneId.systemDefault();
```

Representando Países e Línguas

- java.util.Locale

```
                                                    Language Country Variant
Locale uk = new Locale("en","GB");          // English  Britain
Locale uk = new Locale("en","GB","EURO"); // English  Britain Euro (custom variant)
Locale us = new Locale("en","US");          // English  America
Locale fr = new Locale("fr","FR");          // French   France
Locale cf = new Locale("fr","CA");          // French   Canada
Locale fr = new Locale("fr","029");         // French   Caribbean
Locale es = new Locale("fr");               // French
Locale current = Locale.getDefault();      // current default locale
// Example constructing locale that uses Thai numbers and Buddhist calendar:
Locale th = Locale.forLanguageTag("th-TH-u-ca-buddhist-nu-thai");
```

Formantando e convertendo valores numéricos

- java.text.NumberFormat

```
BigDecimal price = BigDecimal.valueOf(2.99);
Double tax = 0.2;
int quantity = 12345;
Locale locale = new Locale("en","GB");
NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
NumberFormat percentageFormat = NumberFormat.getPercentInstance(locale);
NumberFormat numberFormat = NumberFormat.getNumberInstance(locale);
String formattedPrice = currencyFormat.format(price);
String formattedTax = percentageFormat.format(tax);
String formattedQuantity = numberFormat.format(quantity);
```

value initializations

locale initialization

formatter initializations

formatting values

- método parse returna um Number i pode ser convertido para um wrapper numérico primitivo ou um tipo BigDecimal

```
BigDecimal newPrice = (BigDecimal)currencyFormat.parse("£1.75");
Double newTax = (Double)percentageFormat.parse("12%");
int newQuantity = numberFormat.parse("54,321").intValue();
```

Formatando e convertando Data e horas

- java.time.format.DateTimeFormatter
- java.time.format.FormatSytle (enum)

```
LocalDate date = LocalDate.of(2019,Month.APRIL,1);
Locale locale = new Locale("en","GB");
DateTimeFormatter format = DateTimeFormatter.ofPattern("EEEE dd MMM yyyy",locale);
String result = date.format(formatter);
```

```
Monday 01 Apr 2019
```

value initialization
locale initialization
formatter initialization
format value
formatted result

```
date = LocalDate.parse("Tuesday 31 Mar 2020",dateFormatter);
locale = new Locale("ru");
format = DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM).localizedBy(locale);
result = date.format(formatter);
```

```
31 мар. 2020 г.
```

parse value
reset locale
reset formatter
format value
formatted result

Formatando e convertando Data e horas

- java.time.format.DateTimeFormatter
- java.time.format.FormatSytle (enum)

```
LocalDateTime someTime =
LocalDateTime.of(2019,Month.APRIL,1,17,42);

DateTimeFormatter dateFormatter =
DateTimeFormatter.ofPattern("EEEE dd MMMM YYYY, hh:mm a",
new Locale("en","GB"));

String result = dateFormatter.format(someTime);
```

Trabalhando com Resources

Format Message Patterns

```
Locale locale = new Locale("en","GB");
ResourceBundle bundle = ResourceBundle.getBundle("resource.messages",locale);
// assume following values are already formatted:
String name = "Cookie",
String price = currency.format(price);
String quantity = number.format(quantity);
String bestBefore = date.format(dateFormatter);

String pattern = bundle.getString("product");
String message = MessageFormat.format(pattern, name, price, quantity, bestBefore);
```

```
Cookie, price £2.99, quantity 4, best before 1 Apr 2019
```

initialize locale and bundle

prepare formatters and values

get pattern

substitute values

formatted result

Formatting and Localization

resources/messages_en_GB.properties

```
product={0}, price {1}, quantity {2}, best before {3}
```

```
String name = "Cookie";
BigDecimal price = BigDecimal.valueOf(2.99);
LocalDate bestBefore = LocalDate.of(2019,Month.APRIL,1);
int quantity = 4;

Locale locale = new Locale("en","GB");
ResourceBundle bundle = ResourceBundle.getBundle("resource.messages",locale);

NumberFormat currencyFormat = NumberFormat.getCurrencyInstance(locale);
NumberFormat numberFormat = NumberFormat.getNumberInstance(locale);
DateTimeFormatter dateFormat = DateTimeFormatter.ofPattern("dd MMM yyyy",locale);

String fPrice = currencyFormat.format(price);
String fQuantity = numberFormat.format(quantity);
String fBestBefore = dateFormat.format(bestBefore); // or bestBefore.format(dateFormat);

String pattern = bundle.getString("product");
String message = MessageFormat.format(pattern, name, fPrice, fQuantity, fBestBefore);
```

```
Cookie, price £2.99, quantity 4, best before 1 Apr 2019
```

Java SE 11 Developer Certification 1Z0-819

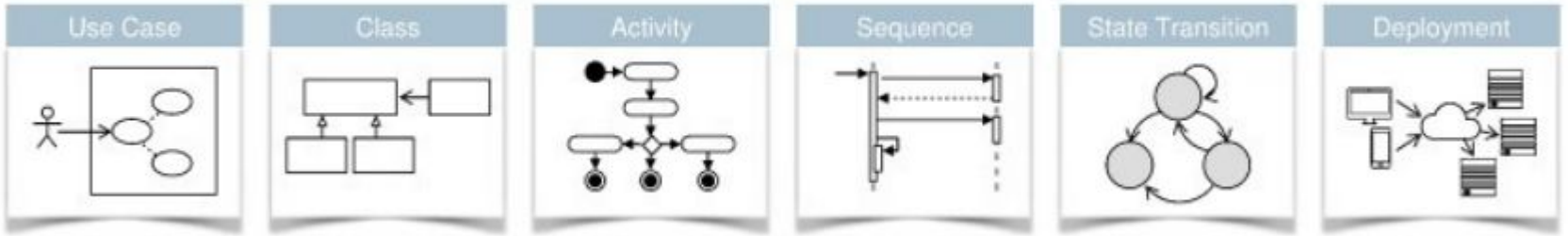
[Classes e Objetos]

Meta

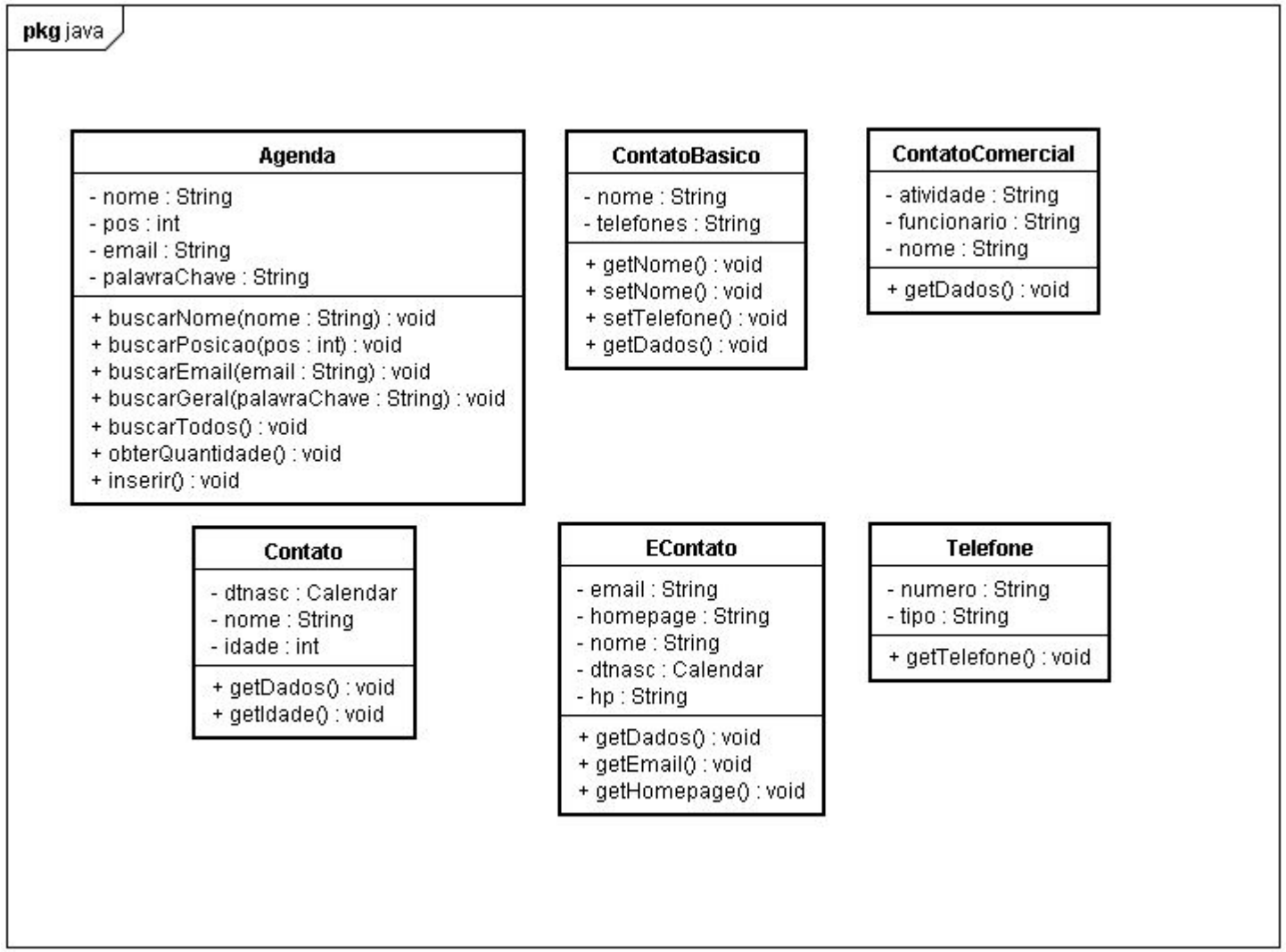
Digital. Simple. Human.

Introdução a UML

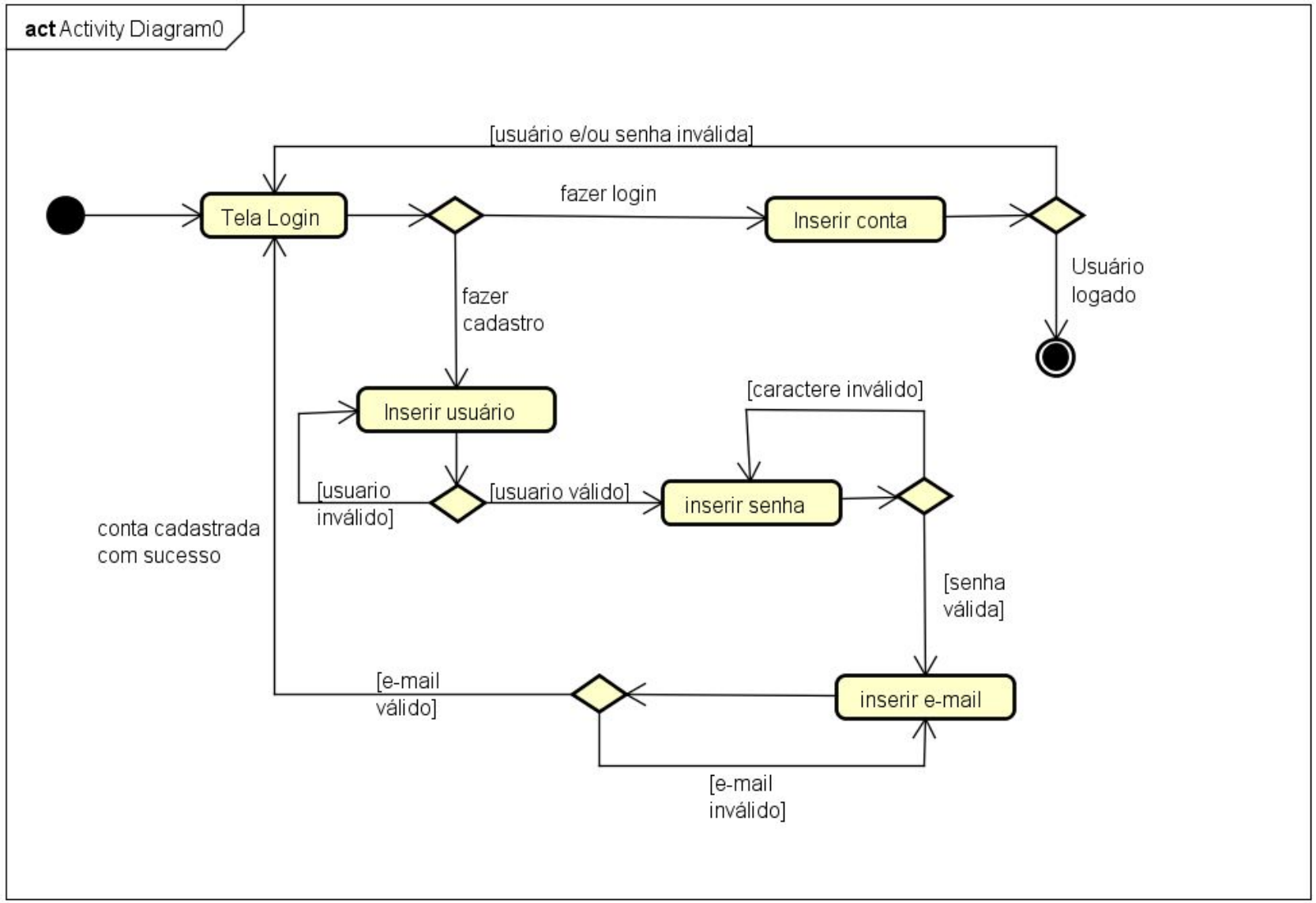
- A UML (do inglês Unified Modeling Language, em português Linguagem de Modelagem Unificada) é uma linguagem-padrão para a elaboração da estrutura de projetos de software. Ela poderá ser empregada para a visualização, a especificação, a construção e a documentação de artefatos que façam uso de sistemas complexos de software. Em outras palavras, na área de Engenharia de Software, a UML é uma linguagem de modelagem que permite representar um sistema de forma padronizada (com intuito de facilitar a compreensão pré-implementação).

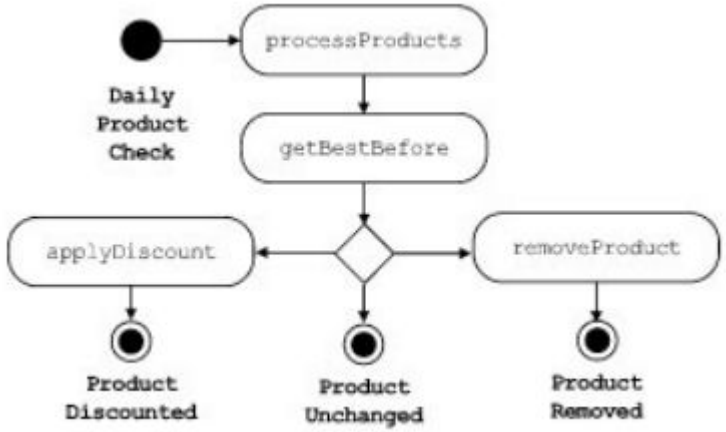


Modelando classes

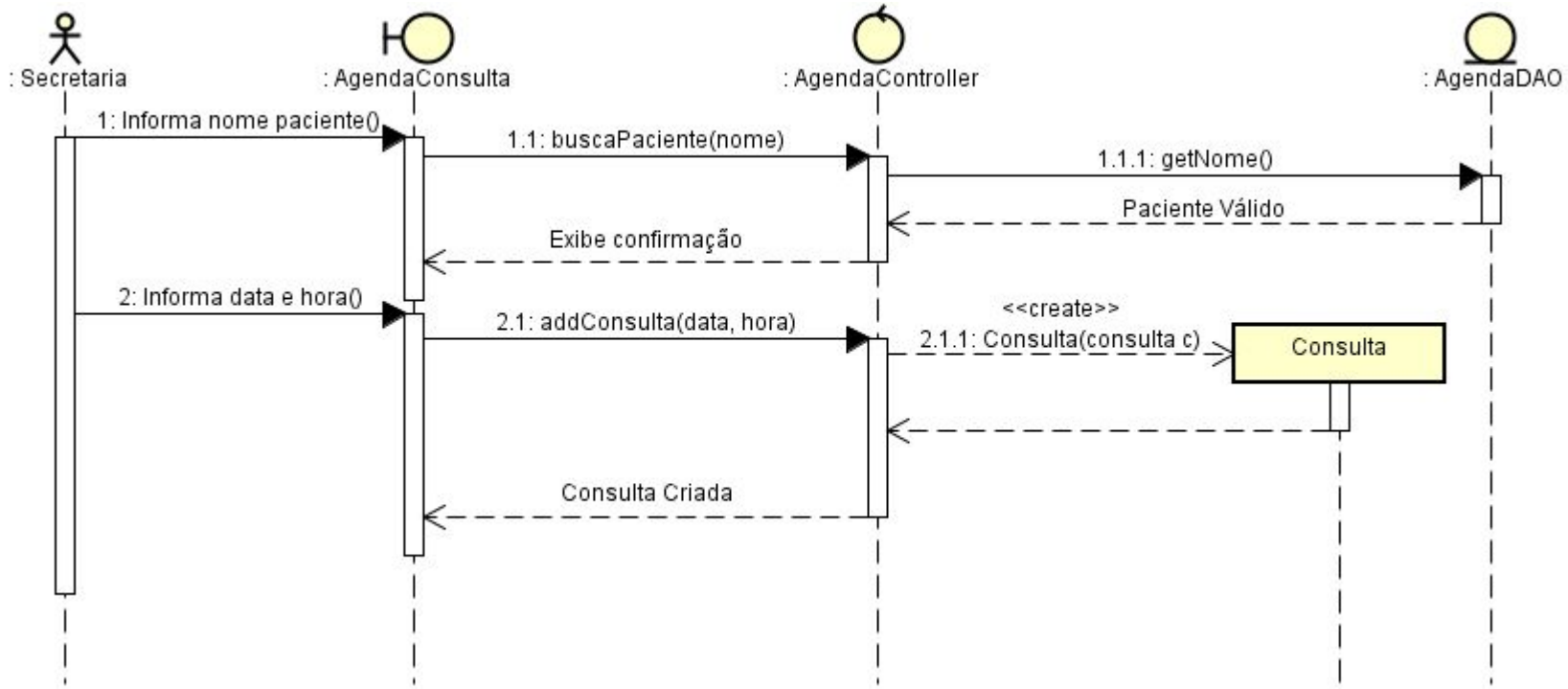


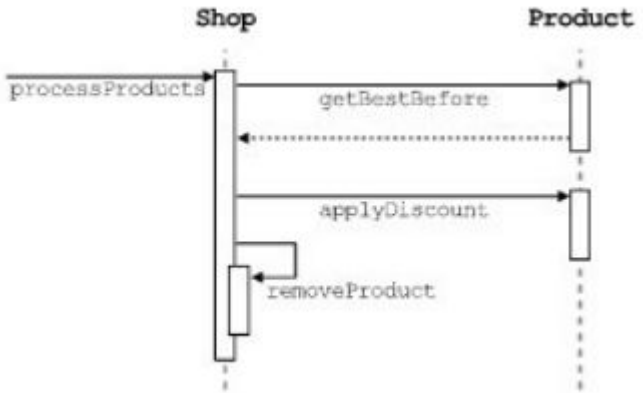
Modelando interação e atividades





Modelando interação e atividades





Design de classes

```
package <package name>;
import <package name>.<OtherClassName>;
<access modifier> class <ClassName> {
  // variables and methods
}
```

```
package demos.shop;
import java.math.BigDecimal;
public class Product {
  private BigDecimal price;
  public BigDecimal getPrice() {
    return price;
  }
  public void setPrice(double value) {
    price = BigDecimal.valueOf(value);
  }
}
```

Criando objetos

```
Product p1 = new Product();
p1.setPrice(1.99);
BigDecimal price = p1.getPrice();
```

- Note: A reference is a typed variable that points to an object in memory.

```
package demos.shop;
import java.math.BigDecimal;
public class Product {
  private BigDecimal price;
  public BigDecimal getPrice() {
    return price;
  }
  public void setPrice(double value) {
    price = BigDecimal.valueOf(value);
  }
}
```

Definindo variáveis

```
package <package name>;
import <package name>.<ClassName>;
<access modifier> class <ClassName> {
  <access modifier> <variable type> <variable name> = <variable value>;
}
```

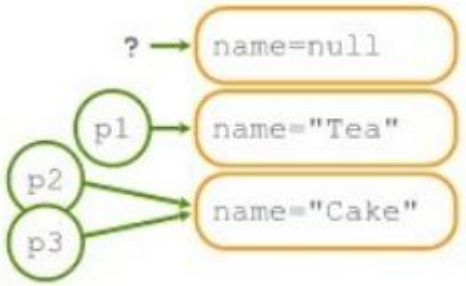
```
package demos.shop;
import java.mathBigDecimal;
import java.time.LocalDate;
public class Product {
  private int id;
  private String name;
  private BigDecimal price;
  private LocalDate bestBefore = LocalDate.now().plusDays(3);
}
```

Definindo métodos

```
package <package name>;
<access modifier> class <ClassName> {
  <access modifier> <return type> <method name>(<ParameterType> <parameterName>,
                                                 <ParameterType> <parameterName>) {

    return <value>;
  }
}
```

```
package demos.shop;
public class Product {
   private String name;
   public String getName() {
      return name;
   }
   public void setName(String newName) {
      name = newName;
   }
}
```

Objetos: Criação e acesso



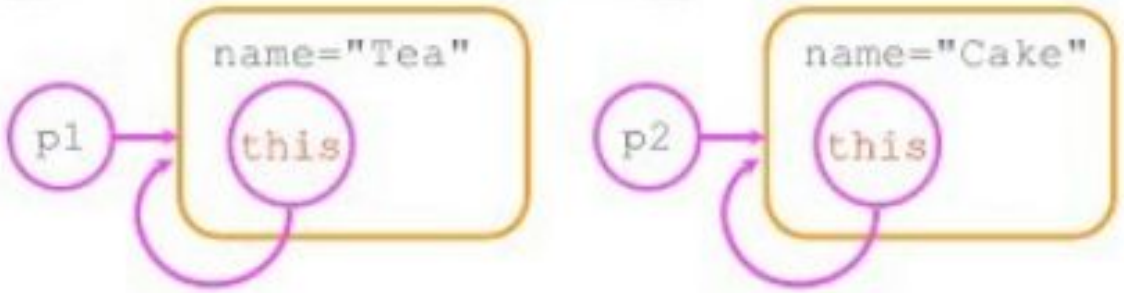

```
package demos.shop;
public class Shop {
   public static void main(String[] args) {
      new Product();
      Product p1 = new Product();
      Product p2 = new Product();
      Product p3 = p2;
      p1.setName("Tea");
      p2.setName("Cake");
      System.out.println(p1.getName()+" in a cup");
      System.out.println(p2.getName()+" on a plate");
      System.out.println(p3.getName()+" to share");
      p1.name = "Coffee";
   }
}
```

```
package demos.shop;
public class Product {
   private String name;
   public void setName(String newName) {
      name = newName;
   }
   public String getName() {
      return name;
   }
}
```

```
>java demos.shop.Shop
Tea in a cup
Cake on a plate
Cake to share
```

Variáveis locais e objetos recursivos

```
Product p1 = new Product();
Product p2 = new Product();
p1.setName("Tea");
p2.setName("Cake");
```




```
public class Product {
    private String name;
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        if (name == null) {
            String dummy = "Unknown";
            return dummy;
        }
        return name;
    }
    public String consume() {
        String feedback = "Good!";
        return feedback;
    }
}
```

Variáveis locais type inference (tipos inferidos)

```
public void someOperation(int param) {
  var value1 = "Hello"; // infers String
  var value2 = param;   // infers int
}
```

Definindo constantes

```
public class Product {
  private final String name = "Tea";
  private final BigDecimal price = BigDecimal.ZERO;
  public BigDecimal getDiscount(final BigDecimal discount) {
    return price.multiply(discount);
  }
}
```

```
public class Shop {
  public static void main(String[] args) {
    Product p = new Product();
    BigDecimal percentage = BigDecimal.valueOf(0.2);
    final BigDecimal amount = p.getDiscount(percentage);
  }
}
```

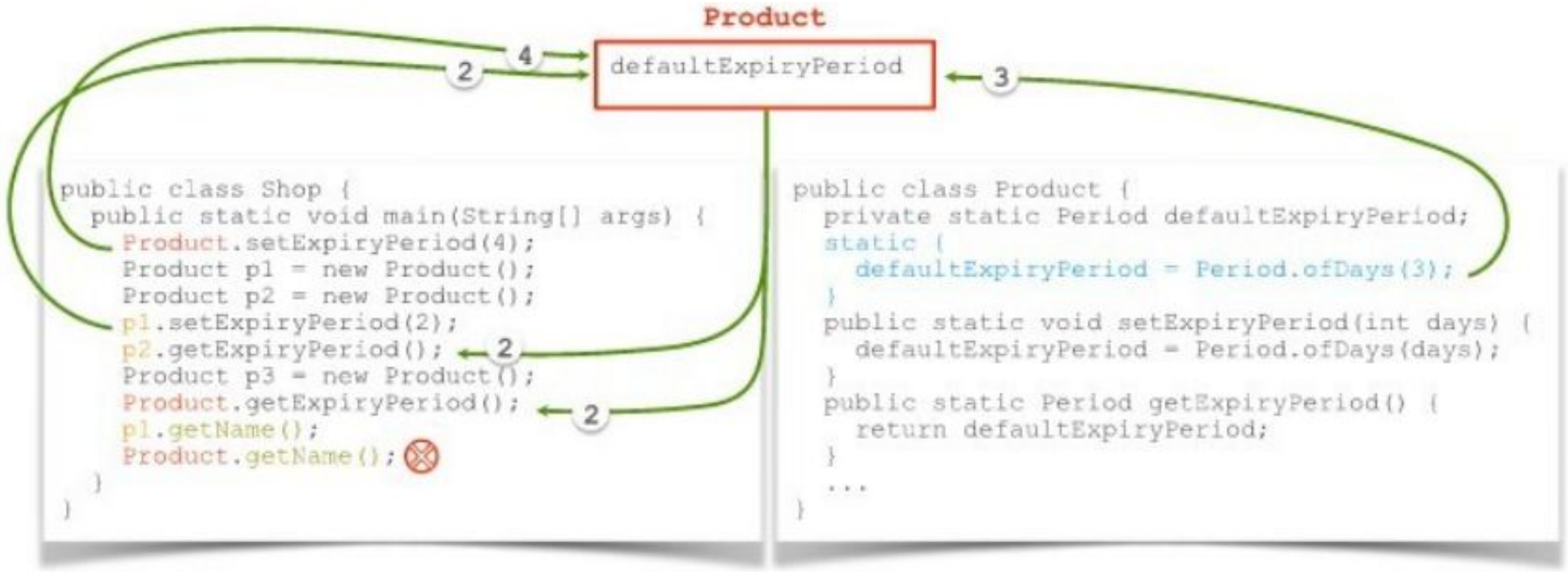
Contextos estáticos

- Objetos podem acessar contextos estáticos compartilhados
- A instância (this) não tem significância dentro de um contexto estático

```
public class Product {
  private static Period defaultExpiryPeriod = Period.ofDays(3);
  private String name;
  private BigDecimal price;
  private LocalDate bestBefore;
  public static void setDefaultExpiryPeriod(int days){
    defaultExpiryPeriod = Period.ofDays(days);
    String name = this.name;
  }
}
```

```
Product p1 = new Product();
Product p2 = new Product();
```

Acessando um contexto estático



Combinando static e final

```
public class Product {
  public static final int MAX_EXPIRY_PERIOD = 5;
  ...
}
```

```
public class Shop {
  public static void main(String[] args) {
    Period expiry = Period.days(Product.MAX_EXPIRY_PERIOD);
    ...
  }
}
```

Exemplos de contextos estáticos (static context)

```
import static Math.random;
public class Shop {
  public static void main(String[] args) {
    Math.round(1.99);
    double value = random();
    BigDecimal.valueOf(1.99);
    LocalDateTime.now();
    ZoneId.of("Europe/London");
    ResourceBundle.getBundle("messages",Locale.UK);
    NumberFormat.getCurrencyInstance(Locale.UK);
    System.out.println("Hello World");
  }
}
```

- imports estáticos, podem referenciar variáveis e métodos estáticos de outra classe.
- Você pode ter a referência de um objeto a partir de um método/atributo estático

```
public class BigDecimal {
  public static BigDecimal valueOf(double val) {
    return new BigDecimal(Double.toString(val));
  }
...
}
```

Obrigado, vamos juntos nessa jornada de Transformação Digital.

Meta

Digital. Simple. Human.