GAMA ACADEMY

Bem-vindos

Acelera Atos

Java™

GAMA ACADEMY

POO

Iniciando o entendimento

Classes

01

Estrutura de uma classe básica

O que é uma classe

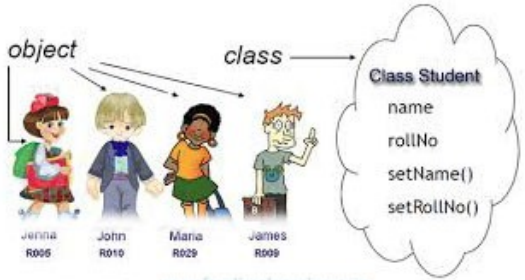
Classe é um tipo de dado

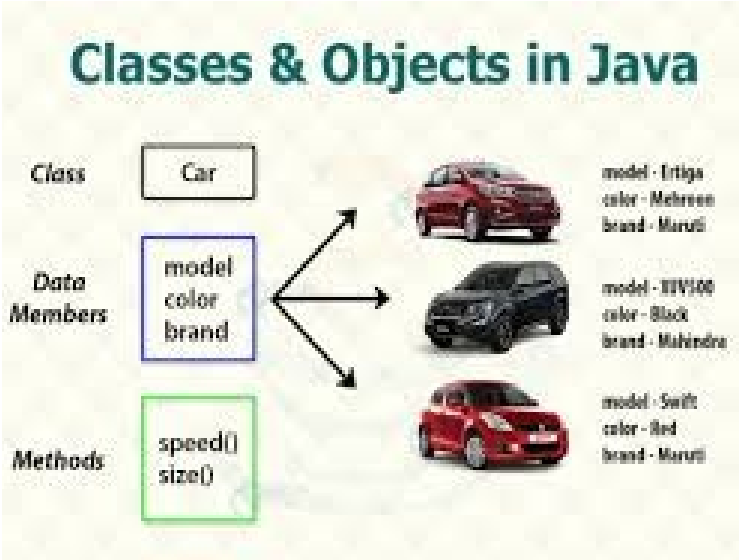
Toda Classe herda Object

A Classe serve para a definição de um Objeto

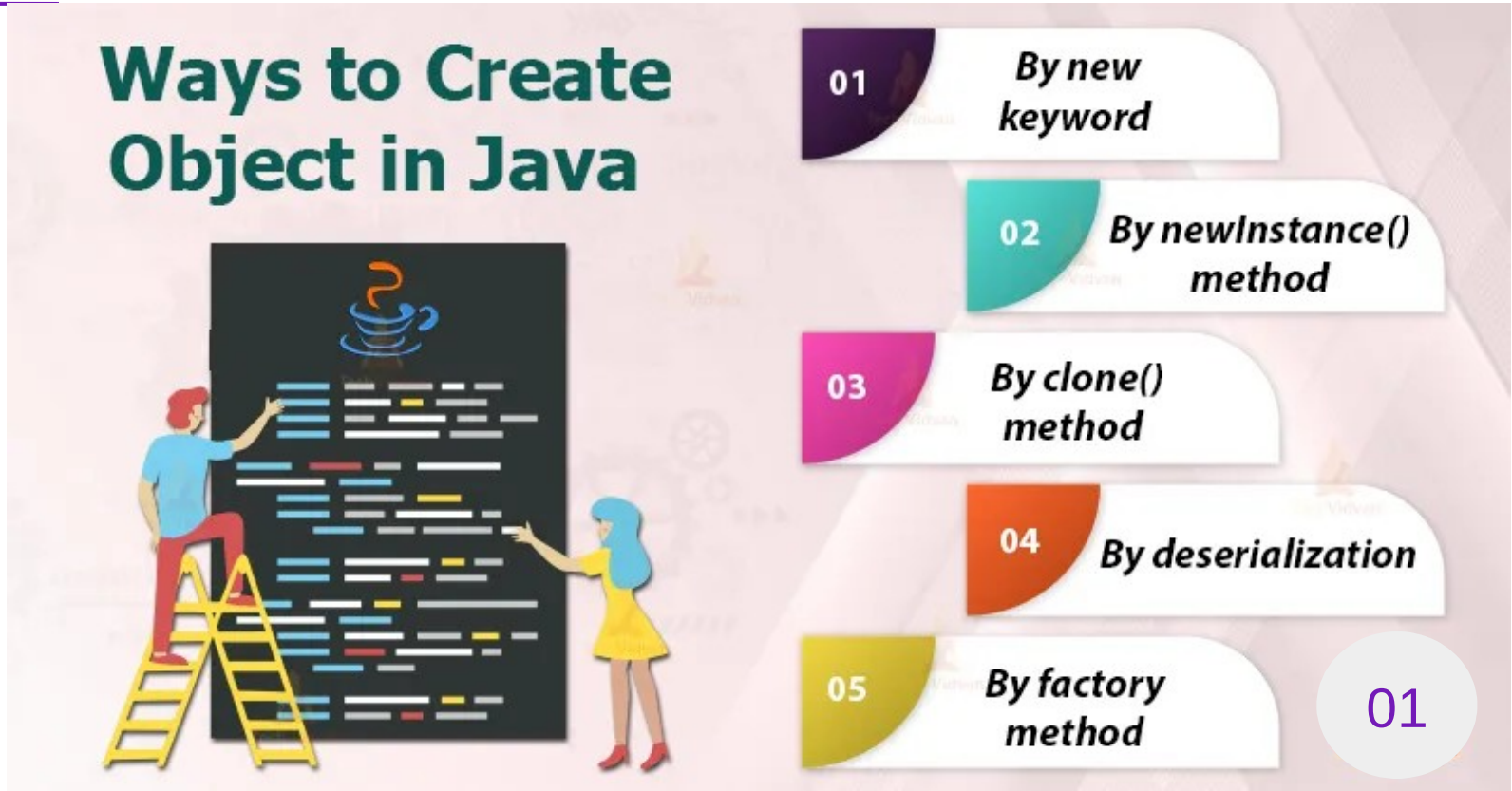
Uma classe deve ser coesa e ter baixo acoplamento

Objeto é a instância de uma classe

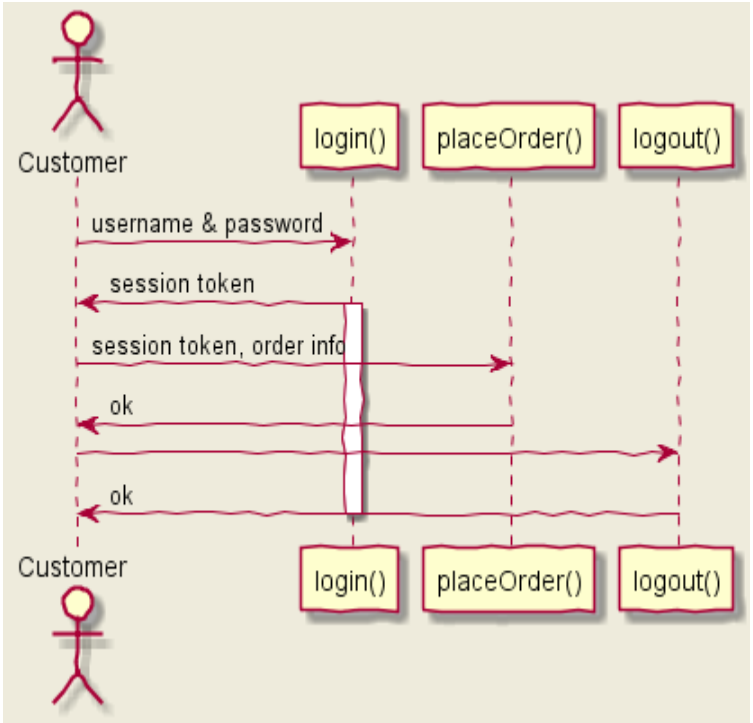


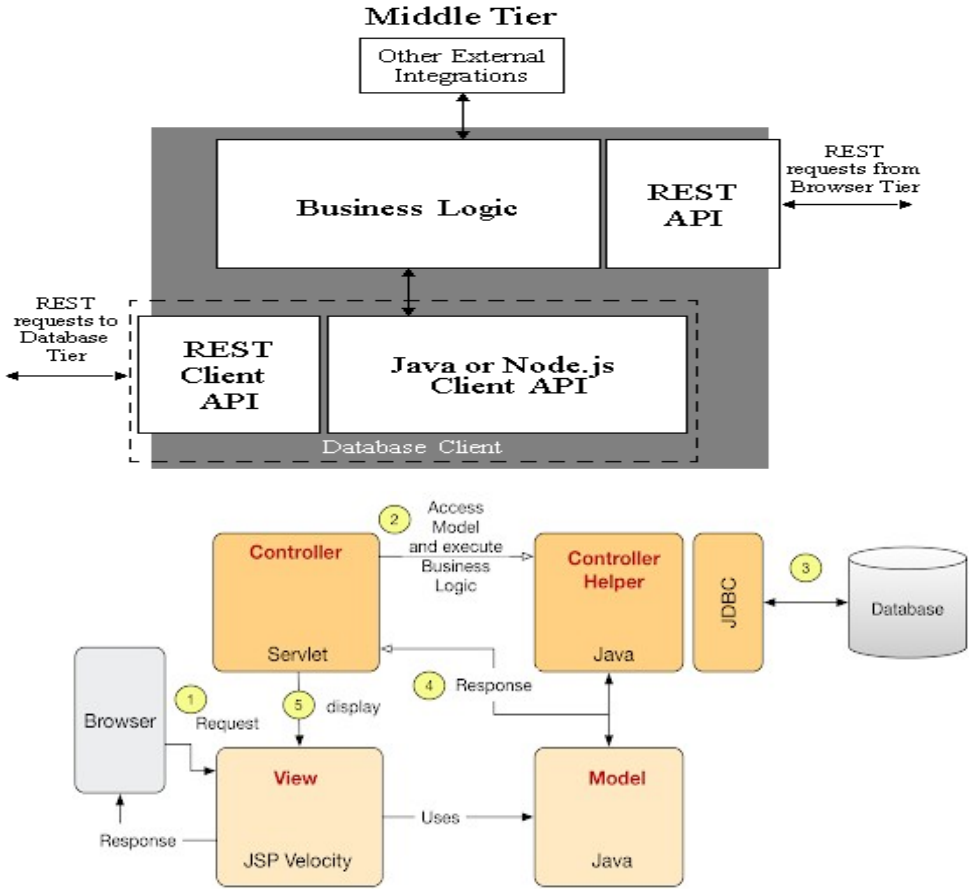


Como criar um Objeto

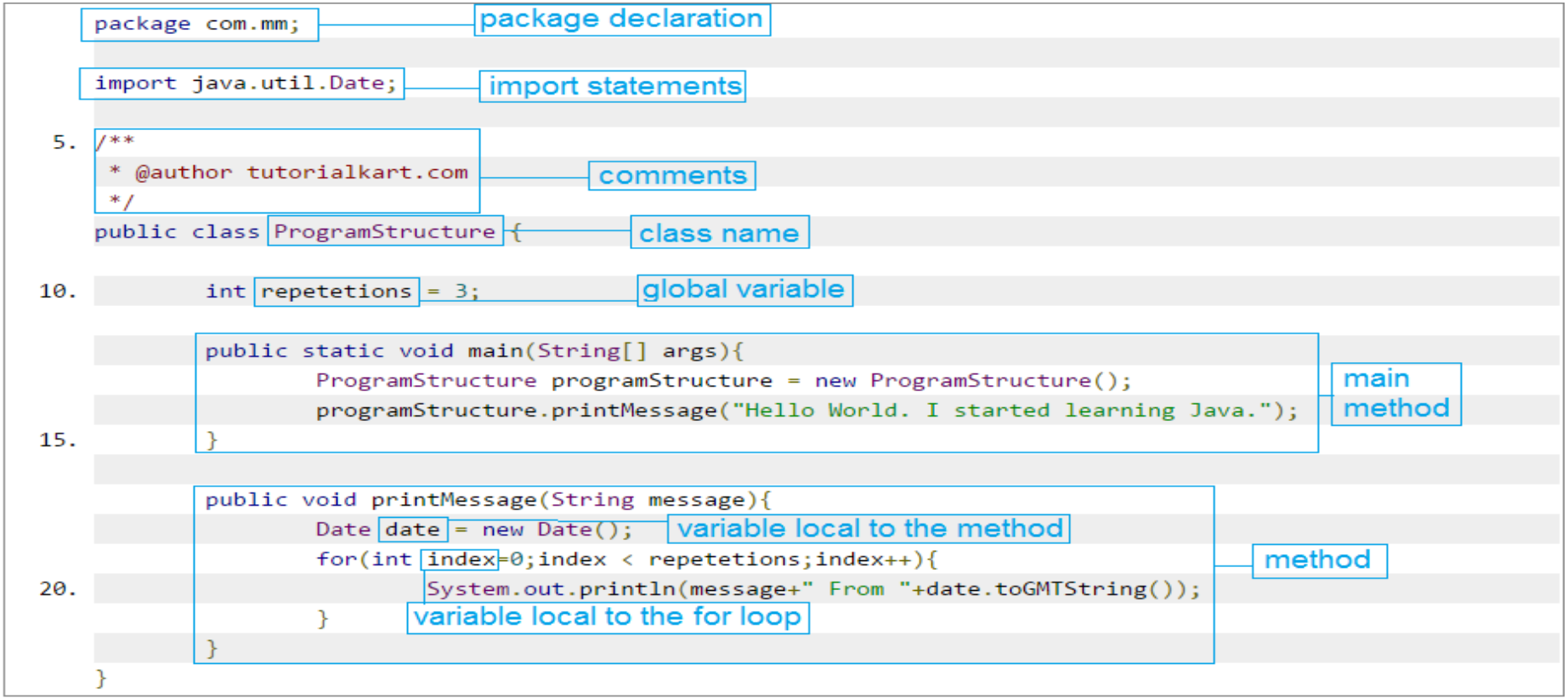


Papel de um Objeto Java no mundo dos sistemas

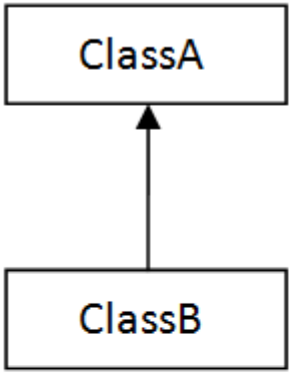




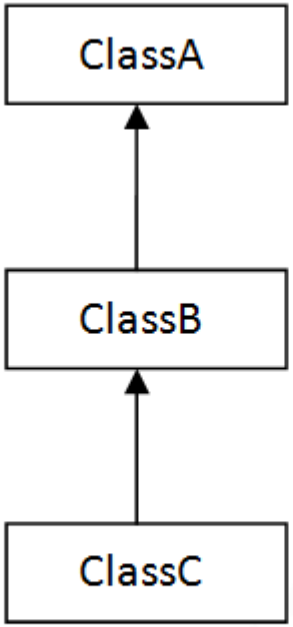
Estrutura de uma Classe



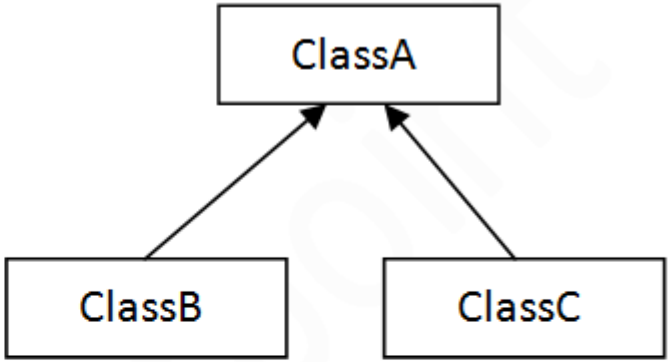
Herança (extends)



1) Single

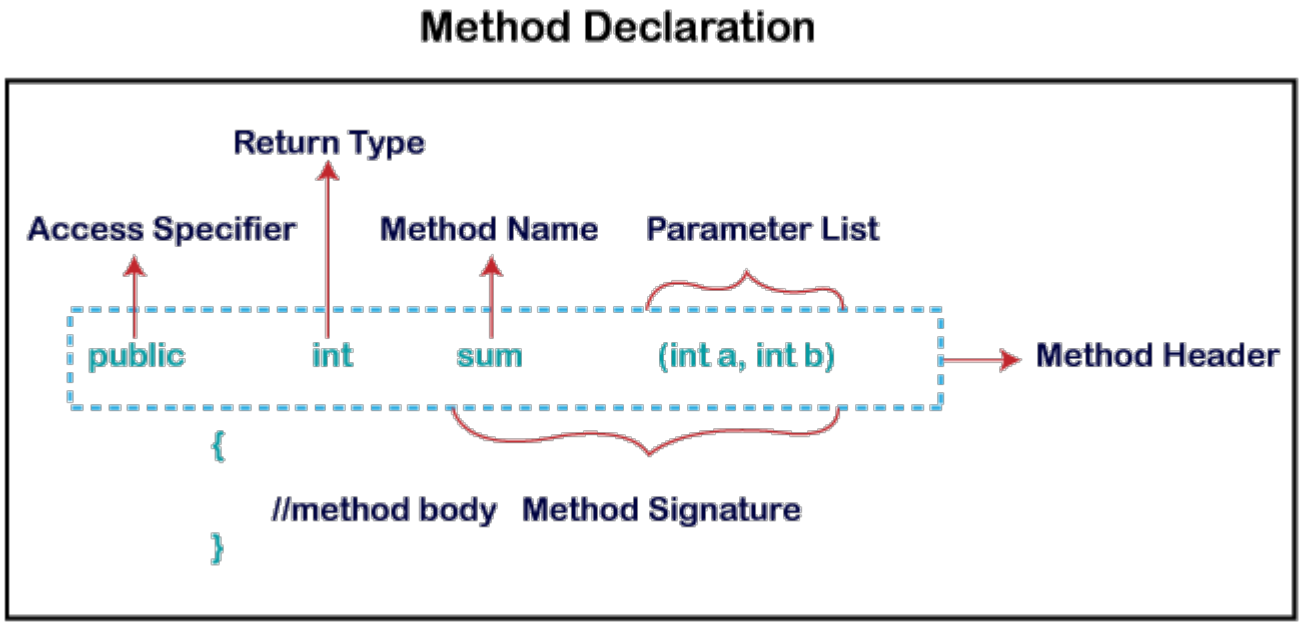


2) Multilevel

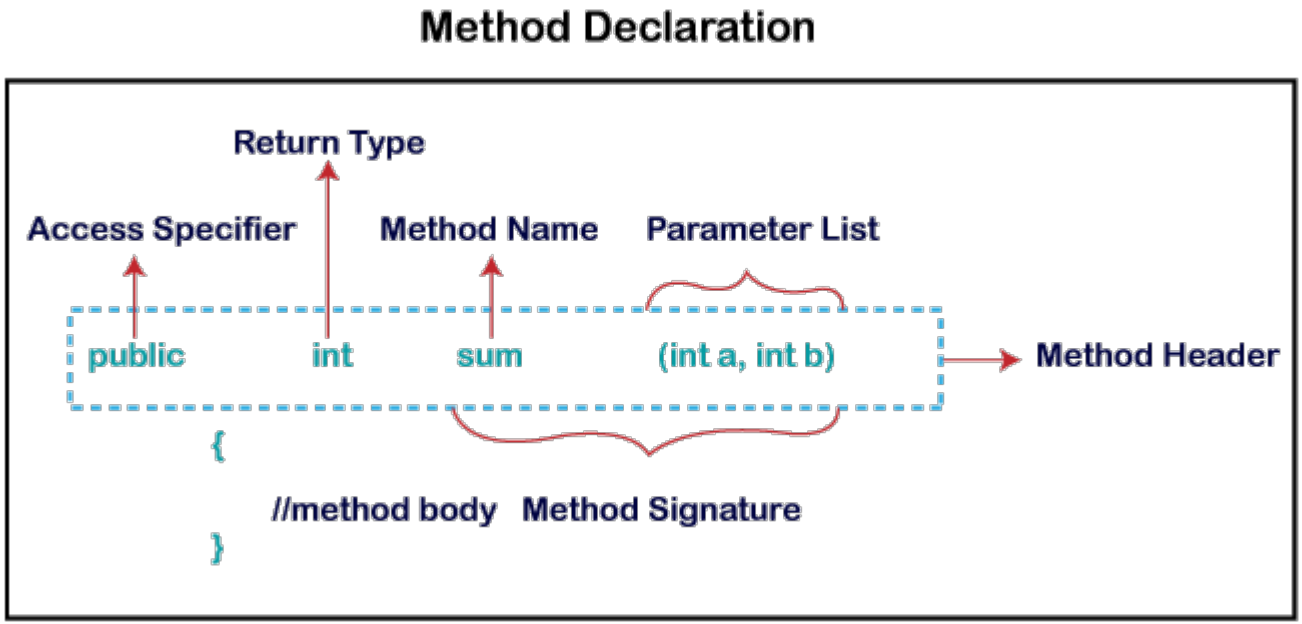


3) Hierarchical

Métodos


Method Declaration
Return Type
Access Specifier
Method Name
Parameter List
public
int
sum
(int a, int b)
Method Header
{
//method body
Method Signature
}

Métodos


Method Declaration
Return Type
Access Specifier
Method Name
Parameter List
public
int
sum
(int a, int b)
Method Header
{
//method body
Method Signature
}

Sobrescrita e Sobrecarga dos Métodos

Overriding

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }
}
class Hound extends Dog{
    public void sniff(){
        System.out.println("sniff ");
    }

    public void bark(){
        System.out.println("bowl");
    }
}
```

Same Method Name,
Same parameter

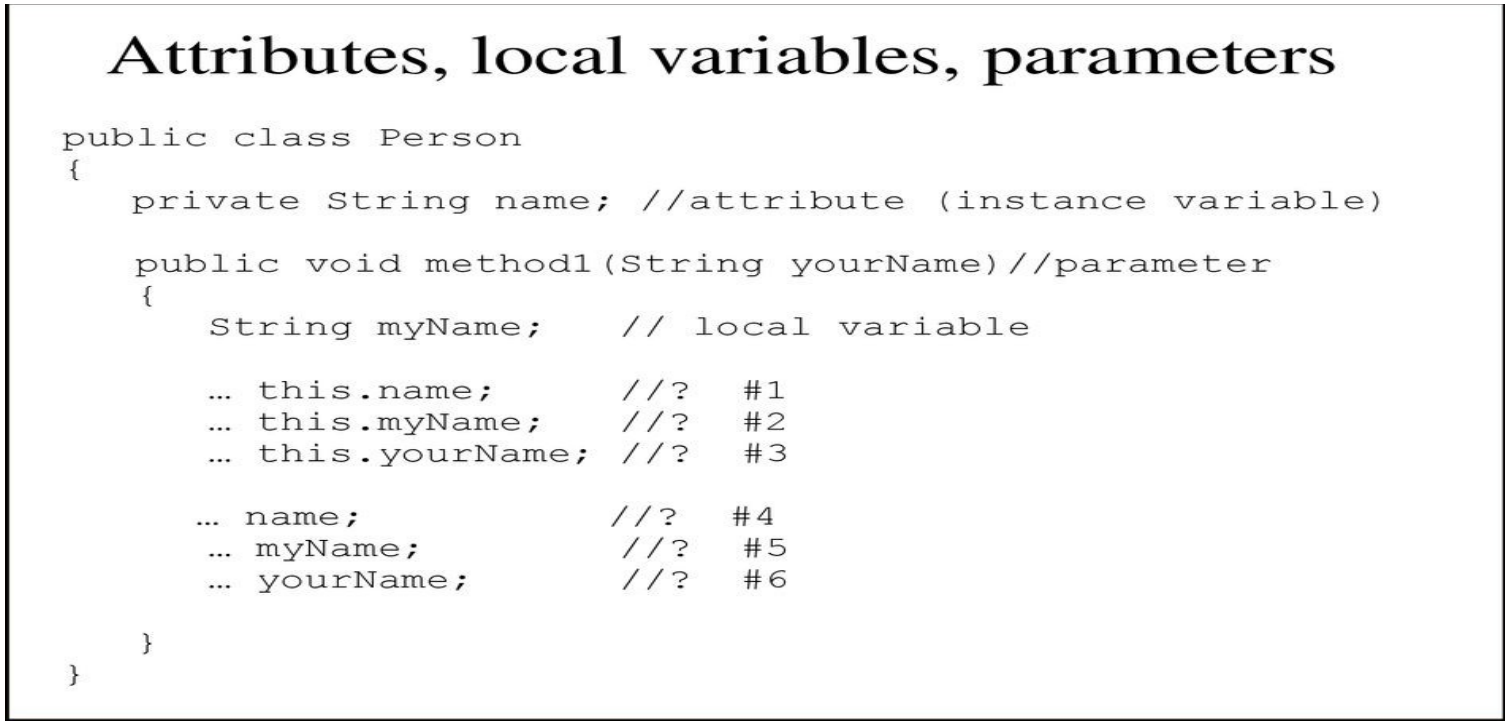
Overloading

```
class Dog{
    public void bark(){
        System.out.println("woof ");
    }

    //overloading method
    public void bark(int num){
        for(int i=0; i<num; i++)
            System.out.println("woof ");
    }
}
```

Same Method Name,
Different Parameter

Atributos de Classe, Variável local e parâmetros



Classes, Métodos e atributos estáticos

Classe estática

Método estático

Atributos/variáveis estáticas

Bloco estáticos

Classes estáticas

InnerClassDemo.java

```java
public class InnerClassDemo {
    public static void main(String[] args) {
        // Accessing the Static Inner class
        InnerClassDemo.StaticInnerClass innerClass = new StaticInnerClass();
        System.out.println("StaticInnerClass value  : " + innerClass.getValue());
        // Accesing the Normal Inner class
        InnerClassDemo.NormalInnerCalss normalInnerCalss = new InnerClassDemo().new NormalInnerCalss();
        System.out.println("normalInnerCalss Value : "+ normalInnerCalss.getValue());
    }
    static class StaticInnerClass {
        int a = 10;
        public int getValue() {
            return a;
        }
    }
    class NormalInnerCalss {
        int instVar = 20;
        public int getValue() {
            return instVar;
        }
    }
}
```

Métodos estáticos

Os métodos estáticos podem ser acessados diretamente usando o nome da classe.

Os métodos estáticos não podem ser substituídos.

Os métodos não estáticos podem acessar métodos estáticos apenas usando o nome da classe.

Os métodos estáticos também podem acessar os métodos não estáticos usando a instância da classe.

Os métodos estáticos e não estáticos não são acessados diretamente.

Um método estático não pode se referir a “this” ou “super” em qualquer lugar.

Atributos/Variáveis estáticas

Variáveis estáticas são declaradas com a palavra-chave static.

Variáveis estáticas também são chamadas de variáveis de classe.

As variáveis de classe pertencem a toda a classe e não a uma instância específica da classe.

Uma única variável estática pode ser compartilhada por todas as instâncias de uma classe.

Não podemos acessar as variáveis estáticas dos métodos normais.

Variáveis de classe são alocadas na memória apenas uma vez no momento do carregamento da classe, e que pode ser comumente acessado por todas as instâncias da classe.

Variáveis estáticas são alocadas na memória do pool estático.

Como a memória para as variáveis da classe é alocada no momento do carregamento da própria classe, podemos acessar as variáveis estáticas diretamente com o próprio nome da classe.

Blocos estáticos

Temos diferentes tipos de blocos em Java, como bloco de inicialização, bloco sincronizado e bloco estático.
Cada bloco tem sua própria importância.
Aqui, um bloco estático é um bloco de instruções, que é definido usando a palavra-chave static.

```
static{
    //Code
}
```

Objetos e referência

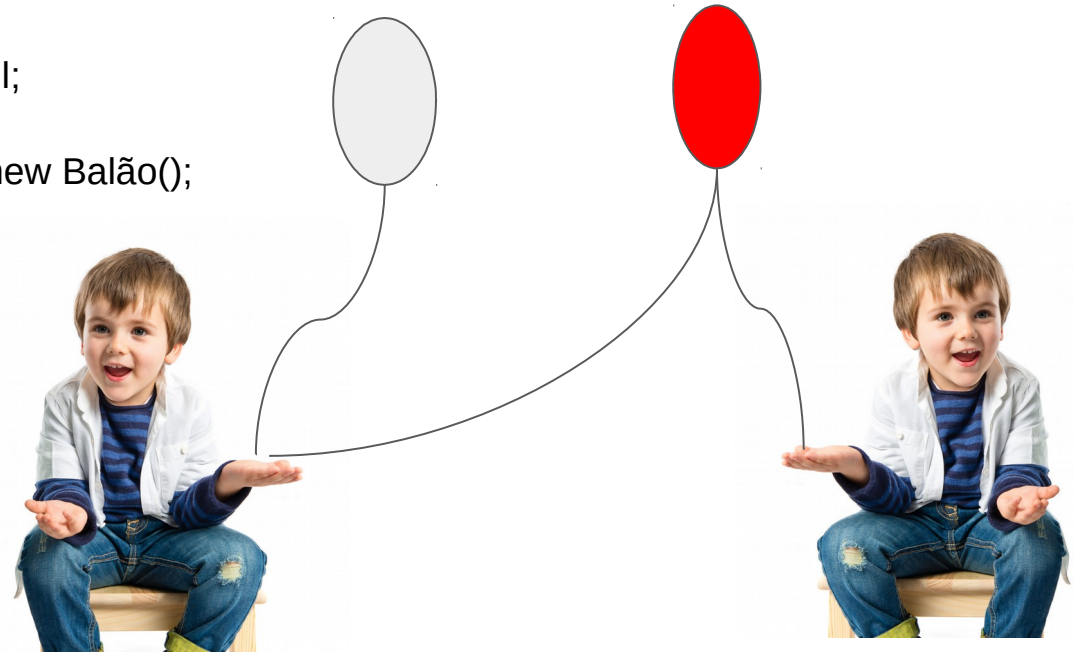
Uma referência é utilizada para armazenar o endereço de um objeto alocado na memória.
Pela referência é manipulado o estado do objeto.
Pode ser um atributo, variável local ou argumentos de métodos.

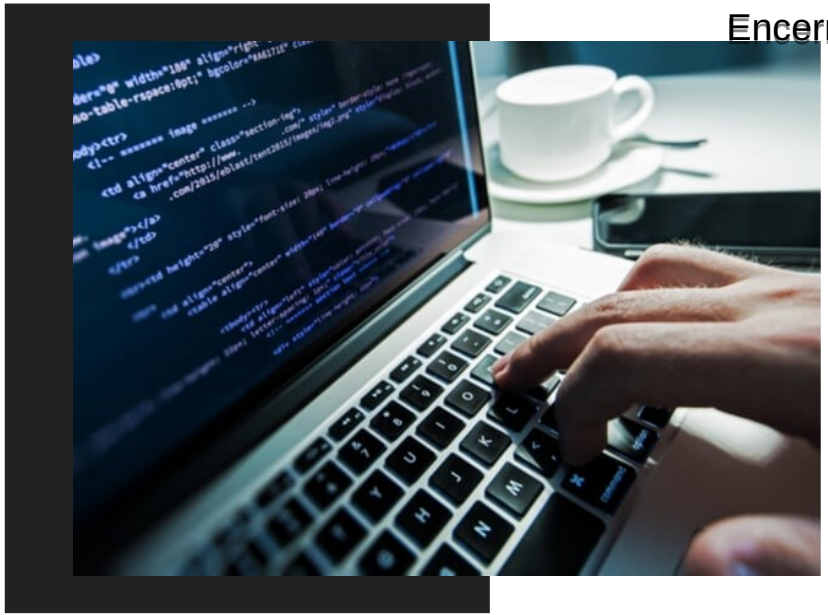
Balão criança = null;

Balão criança = new Balão();

criança = new Balão();

Balão criança2 = criança;





Encerramos por hoje !!!

GAMA ACADEMY

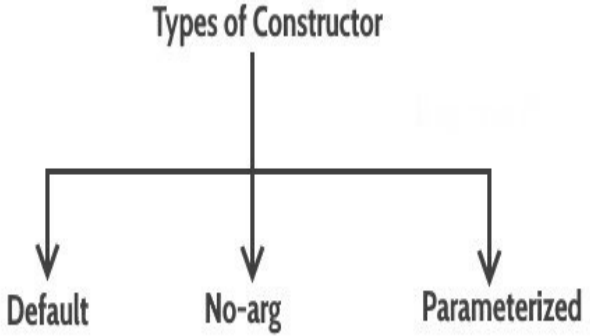
POO

Continuando o entendimento

Construtores

um bloco de código que executa na instanciação de uma classe

Eles são invocados implicitamente



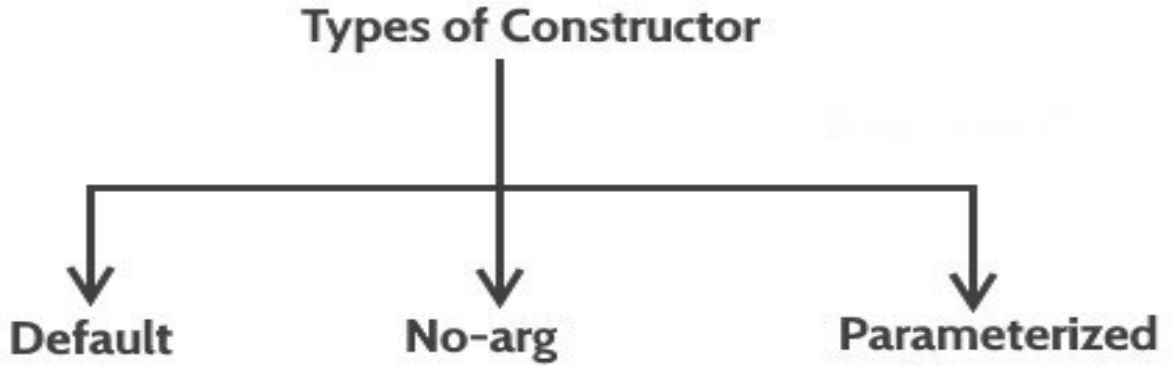
Toda Classe já possui um construtor default herdado do Class

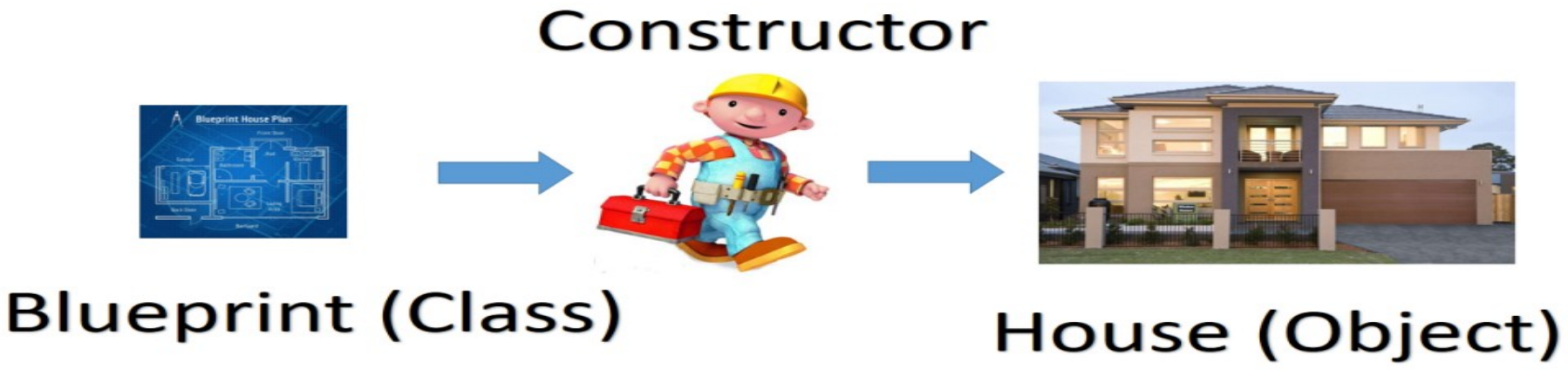
Construtores não possuem retornos

Devem conter o mesmo nome da Classe

Podem conter inúmeros parâmetros respeitando as regras da sobrecarga.

Construtores

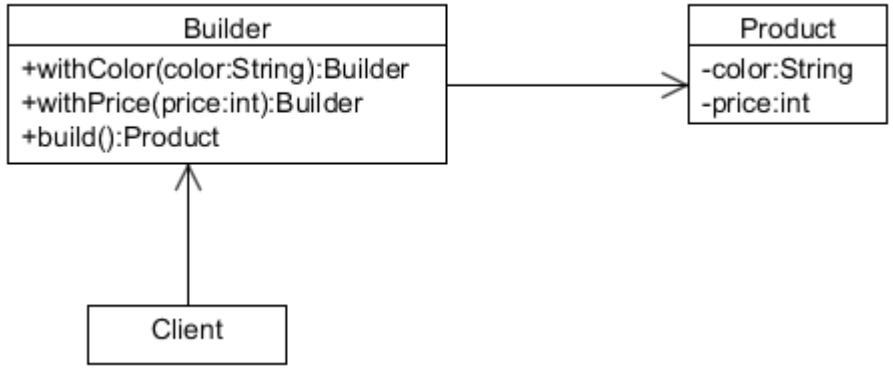




Padrão Builder

Responsabilidade de criar o objeto

Remove a complexidade de quando o objeto necessita de vários atributos para ser criado



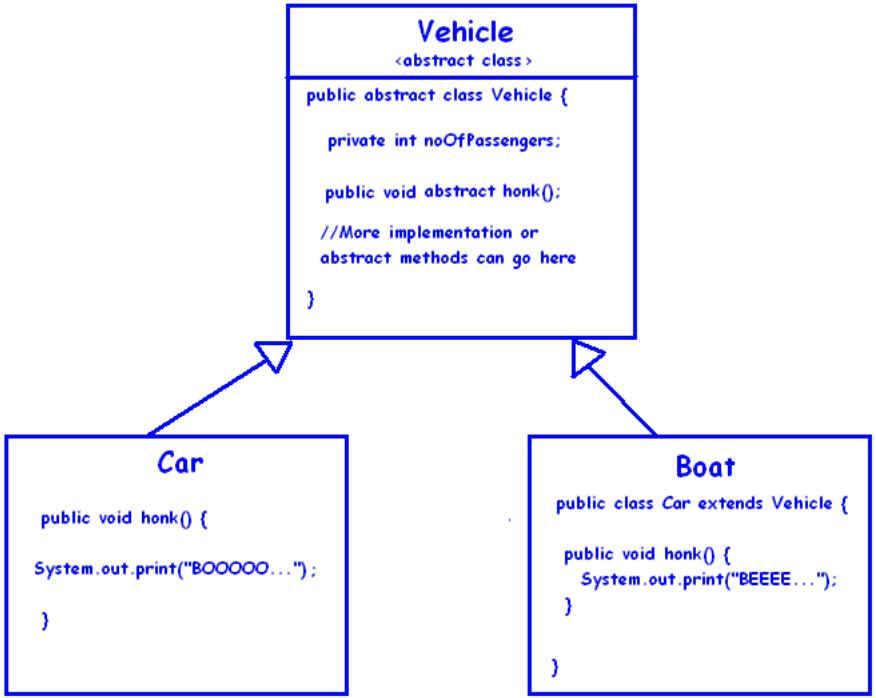
Classes e métodos Abstratas

Classes Abstratas servem como contrato

Não se pode instanciar diretamente uma c
abastrata

Pode conter implementação

Métodos abstratos não tem implementaçã



Classes, métodos, atributos/variáveis e final

- Classes final não pode ser herdada
- Métodos final não podem ser sobrescritos
- Atributos/Variáveis final são constantes que não podem mudar o valor

Pacotes e visibilidade

Visibilidade	public	protected	default	private
A partir da mesma classe	✓	✓	✓	✓
Qualquer classe no mesmo pacote	✓	✓	✓	⊘
Qualquer classe filha no mesmo pacote	✓	✓	✓	⊘
Qualquer classe filha em pacote diferente	✓	✓	⊘	⊘
Qualquer classe em pacote diferente	✓	⊘	⊘	⊘

Encapsulamento e a proteção dos atributos

Encapsulation in Java

- A way to achieve abstraction for objects' data.
- Hides object properties from outer world.
- Provides methods to get/set object data.
- Also called "data-hiding".
- Advantages:
  - Loosely coupled code
  - Better access control and security
  - Reusable code
  - Easy to test

Polimorfismo

polimorfismo é a capacidade de um objeto ser referenciado de diversas formas diferentes e com isso realizar as mesmas tarefas de diferentes formas.

Polimorfismo significa "muitas formas", é o termo definido em linguagens orientadas a objeto, como por exemplo Java, C# e C++, que permite ao desenvolvedor usar o mesmo elemento de formas diferentes.

Polimorfismo denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem. No Polimorfismo temos dois tipos:

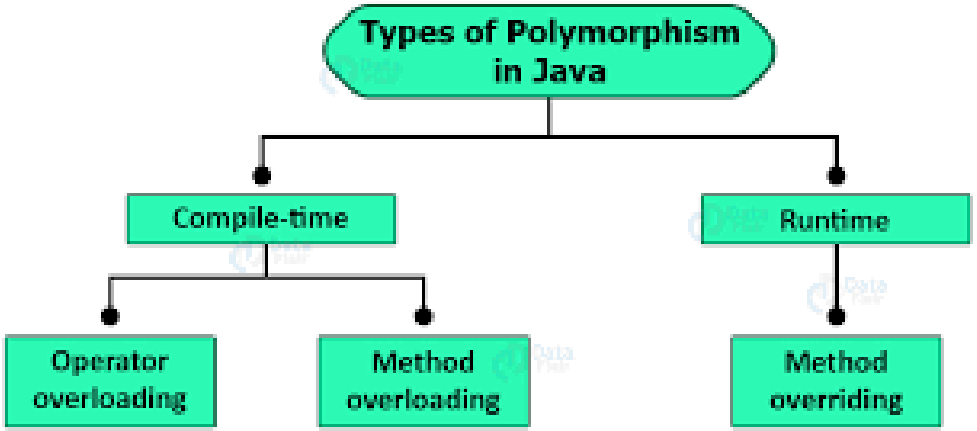
- Polimorfismo Estático ou Sobrecarga
- Polimorfismo Dinâmico ou Sobreposição

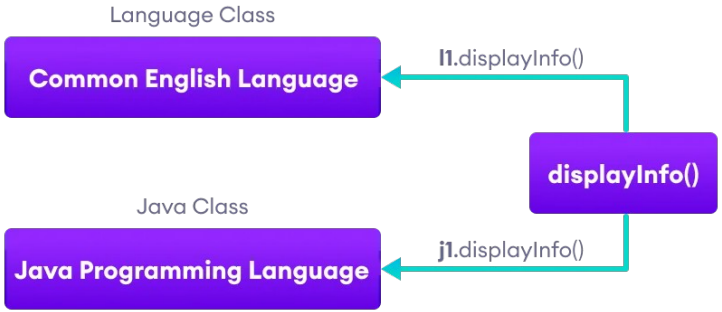
Polimorfismo

Boa parte dos padrões de projeto de software baseia-se no uso de polimorfismo, por exemplo: Abstract Factory, Composite, Observer, Strategy, Template Method, etc.

É notavel a importância do Polimorfismo para a redução de código, simplicidade, flexibilidade, etc. O polimorfismo é utilizado em diversas refatorações e muitos Padrões de Projetos, portanto entendê-lo é fundamental para qualquer desenvolvedor.

Polimorfismo





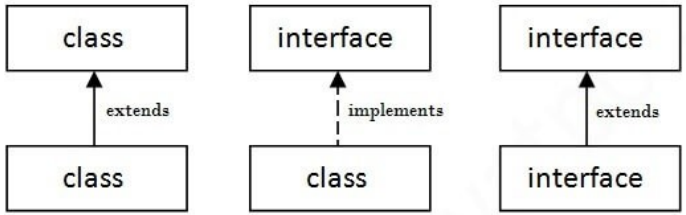
Interfaces

O papel da interface é criar um contrato

Não se pode criar objetos com interfaces

Uma interface pode herdar uma ou várias interfaces

Uma Classe pode implementar uma ou várias interfaces



var-args

Recurso implantado na versão Java 5

Permite a inclusão de 0 ou mais argumentos do mesmo tipo

Genéricos

Introduzido no java 5

Criado para evitar erros em runtime

Necessidade de cast na recuperação dos dados com tipagem vinculada na declaração

Wildcards comuns *List<?>, List<? extends Number>* e *List<? super Integer>*

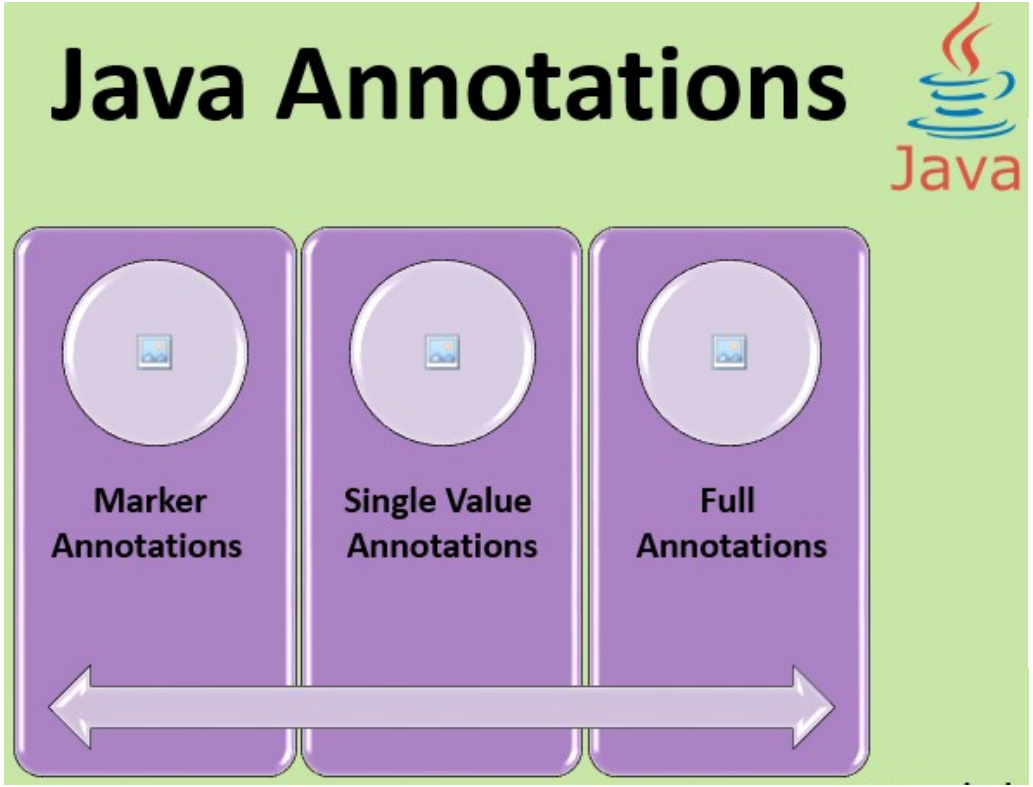
Enumeration

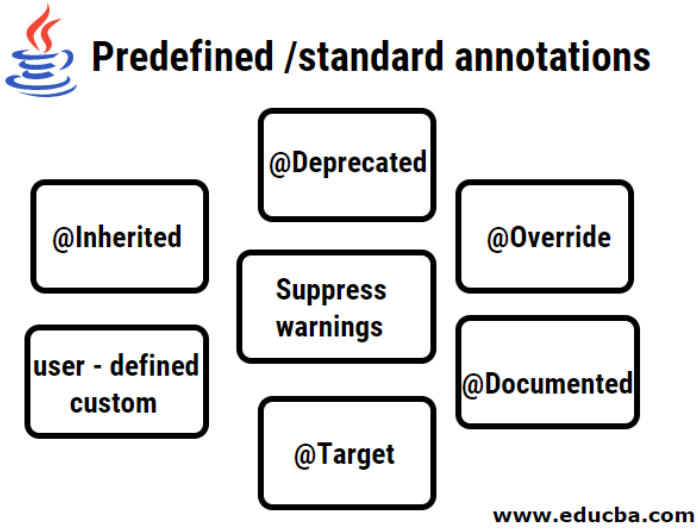
Introduzido no java 5

O tipo **Enum** no Java, é um tipo de valores constantes, pré definidas, que servem para várias situações do dia a dia de um programador.

A sintaxe de criação de um **Enum** no Java é muito semelhante a criação de uma classe, porém no lugar de usar class na declaração, usamos enum.

Anotações


Java Annotations
Java
Marker Annotations
Single Value Annotations
Full Annotations


Predefined /standard annotations
@Deprecated
@Inherited
@Override
Suppress warnings
user - defined custom
@Documented
@Target
www.educba.com

Anotações

As anotações ajudam a definir metadados no código de maneira padronizada. Além disso, as anotações ajudam a fornecer instruções para o seu compilador java seguir ao compilar esse código java.

Pontos importantes a serem lembrados são que todas as anotações estendem a interface java.lang.annotation.Annotation. Além disso, as anotações não podem incluir nenhuma cláusula extended.

Anotações

Ao usar as anotações, usamos o sinal ‘@’ seguido pelo nome de sua anotação para que o compilador a trate como uma anotação. É importante notar que as anotações podem ser adicionadas antes de uma declaração de -

- Classe
- Variável membro
- construtor
- método
- parâmetros de métodos/construtores
- Variável local.

Conceito SOLID

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. **I**nterface Segregation
5. **D**ependency Inversion

Single Responsibility

```
public class Book {

    private String name;
    private String author;
    private String text;

    //constructor, getters and setters

    // methods that directly relate to the book
properties
    public String replaceWordInText(String word){
        return text.replaceAll(word, text);
    }

    public boolean isWordInText(String word){
        return text.contains(word);
    }
}
```

```
public class Book {
    //...

    void printTextToConsole(){
        // our code for formatting and
printing the text
    }
}
```

```
public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String text){
        //our code for formatting and printing the
text
    }

    void printTextToAnotherMedium(String
text){
        // code for writing to any other location..
    }
}
```

Open/Closed

```
public class BookPrinter {

    // methods for outputting text
    void printTextToConsole(String
text){
        //our code for formatting and
printing the text
    }

    void
printTextToAnotherMedium(String
text){
        // code for writing to any other
location..
    }
}
```

```
public class
SuperCoolGuitarWithFlames
extends Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

Liskov Substitution

```
public interface Car {

    void turnOnEngine();
    void accelerate();
}
```

```
public class MotorCar implements Car
{

    private Engine engine;

    //Constructors, getters + setters

    public void turnOnEngine() {
        //turn on the engine!
        engine.on();
    }

    public void accelerate() {
        //move forward!
        engine.powerOn(1000);
    }
}
```

```
public class ElectricCar implements Car {

    public void turnOnEngine() {
        throw new
          AssertionError("I    don't have an engine!");
    }

    public void accelerate() {
        //this acceleration is crazy!
    }
}
```

Interface Segregation

```
public interface BearKeeper {
    void washTheBear();
    void feedTheBear();
    void petTheBear();
}
```

```
public interface BearCleaner {
    void washTheBear();
}

public interface BearFeeder {
    void feedTheBear();
}

public interface BearPetter {
    void petTheBear();
}
```

```
public class BearCarer implements BearCleaner, BearFeeder {

    public void washTheBear() {
        //I think we missed a spot...
    }

    public void feedTheBear() {
        //Tuna Tuesdays...
    }
}
```

Dependency Inversion

```
public class Windows98Machine {

    private final StandardKeyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine() {
        monitor = new Monitor();
        keyboard = new StandardKeyboard();
    }

}
```

```
public interface Keyboard { }
```

```
public class Windows98Machine{

    private final Keyboard keyboard;
    private final Monitor monitor;

    public Windows98Machine(Keyboard
keyboard, Monitor monitor) {
        this.keyboard = keyboard;
        this.monitor = monitor;
    }
}
```

```
public class StandardKeyboard implements Keyboard { }
```

GAMA ACADEMY

Recursos

APIs e controles de fluxos

Datas com java

java.util.Date
java.sql.Date

java.util.Calendar e java.util.GregorianCalendar

Formatação de Datas com java

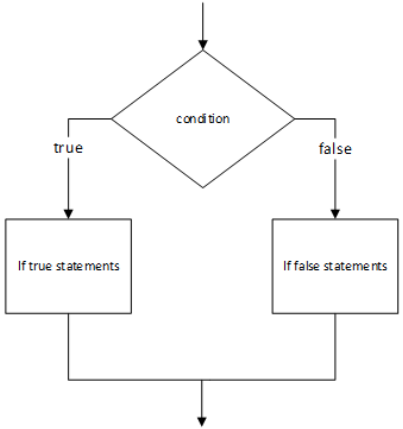
```
SimpleDateFormat formato = new SimpleDateFormat("dd/MM/yyyy");
Date data = formato.parse("23/11/2015");
```

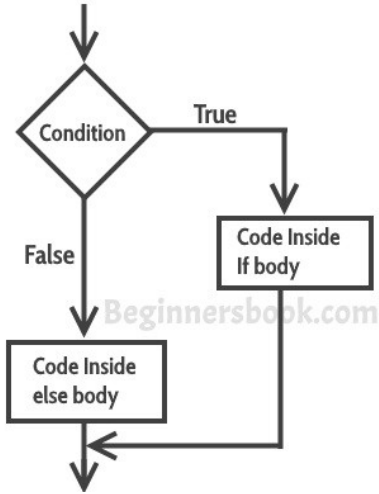
https://www.devmedia.com.br/utilizando-recursos-do-java-para-formatacao-de-datas/5720

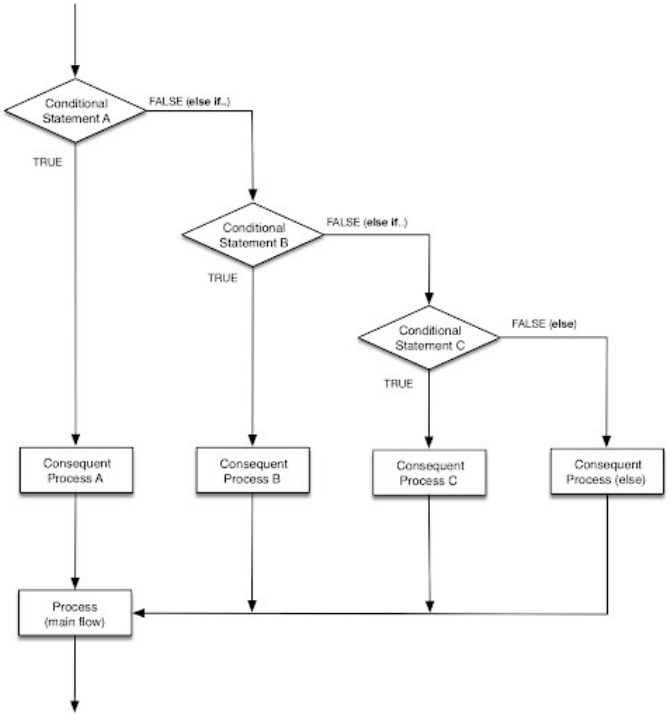
Nova biblioteca de datas

LocalDate, LocalTime e LocalDateTime

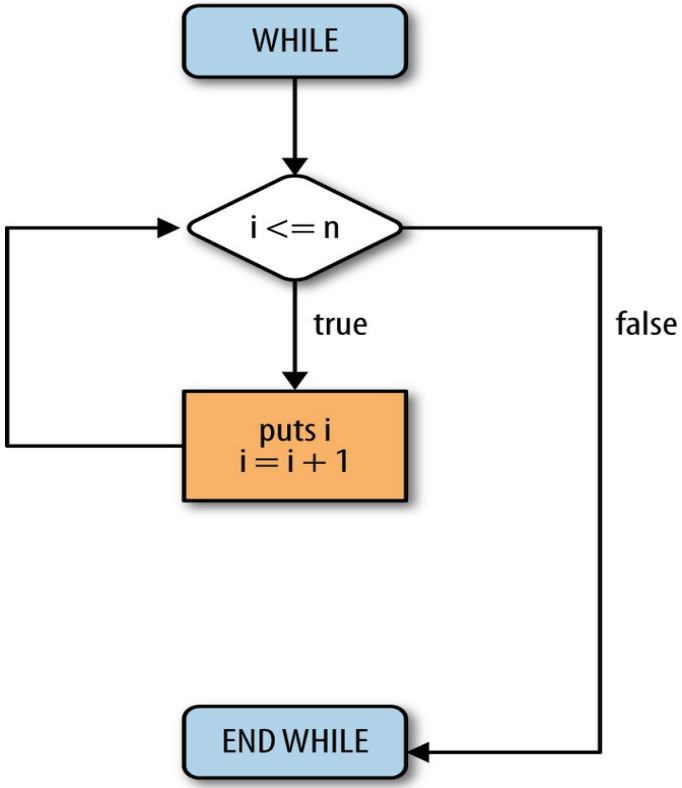
Fluxo de decisão (if, else, switch)



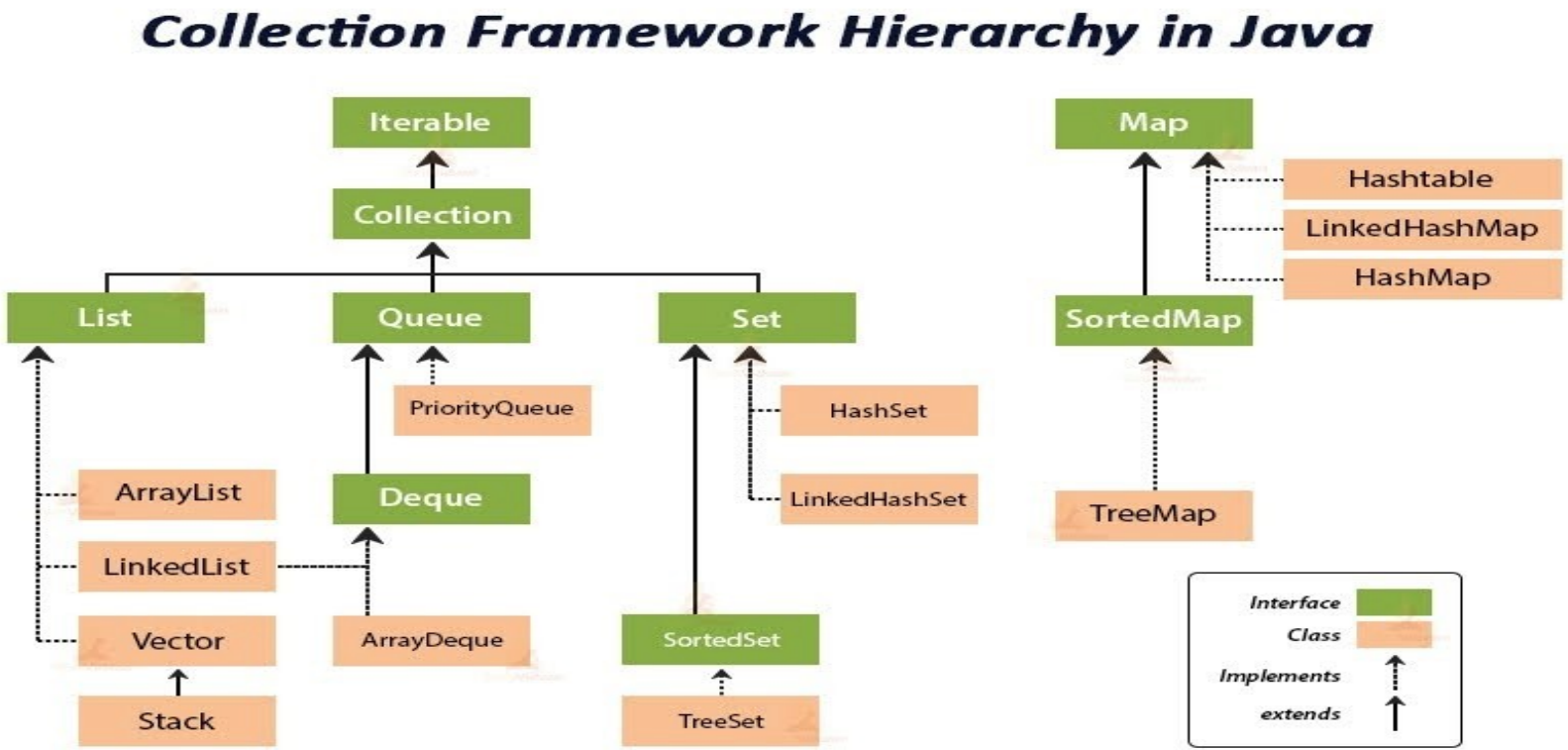




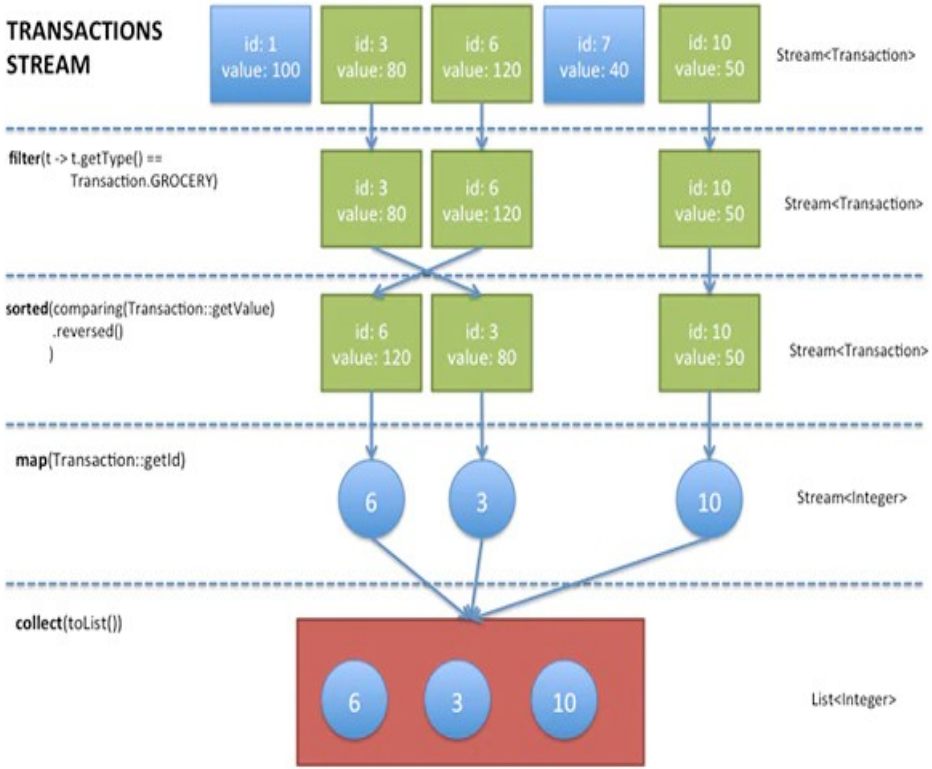
Fluxo de repetição (for/do/while/break/continue)

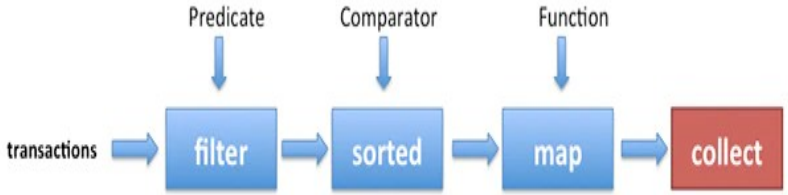


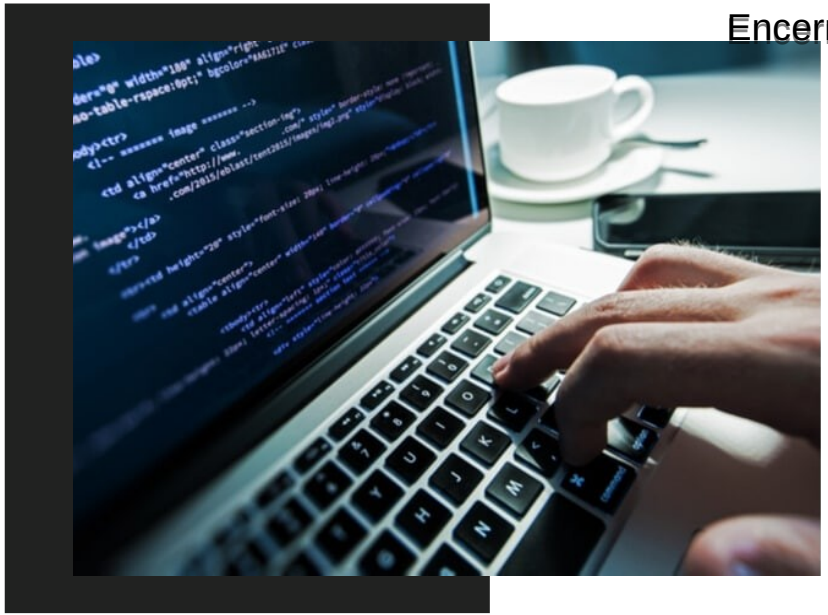
Collections



Streams


TRANSACTIONS STREAM
id: 1 value: 100
id: 3 value: 80
id: 6 value: 120
id: 7 value: 40
id: 10 value: 50
Stream<Transaction>
filter(t -> t.getType() == Transaction.GROCERY)
id: 3 value: 80
id: 6 value: 120
id: 10 value: 50
Stream<Transaction>
sorted(comparing(Transaction::getValue) .reversed())
id: 6 value: 120
id: 3 value: 80
id: 10 value: 50
Stream<Transaction>
map(Transaction::getId)
6
3
10
Stream<Integer>
collect(toList())
6
3
10
List<Integer>


Predicate
Comparator
Function
transactions
filter
sorted
map
collect



Encerramos por hoje !!!