Algoritmos e Estruturas de Dados

Universidade do Algarve

Néstor Cataño

nccollazos@ualg.pt

Method and Language

- **Classes and Specification**
- **Classes as modules and Classes as Types**
- **Computational Mechanism**
- **Information Hiding**
- **Exception Handling**
- **Static typing**
- **Genericity**
- **Inheritance**
- **Redefinition**
- **Polymorphism**
- **Dynamic Binding**
- **Run-time type interrogation**
- **Deferred features and classes**

Classes

The method and the language should have the notion of class as their central concept

- A class is a software element describing an abstract data type and its partial or total implementation
- Examples: Person, Animal, Stack, Car, Binary Tree, Interface, Computer, etc.
- The object-oriented method is based on the notion of class

Class "Pessoa" in Java

```
class Pessoa {
  String nome = "";
  int idade = 0;

  int getIdade() { return idade; }

  String getNome() { return nome;}

  void setIdade(int i) { idade = i; }

  void setNome(String n) { nome = n; }
}
```

Class "Pessoa" in Java

Declaração da clase
Pessoa

```
class Pessoa {
  String nome = "";
  int idade = 0;

  int getIdade() { return idade; }

  String getNome() { return nome;}

  void setIdade(int i) { idade = i; }

  void setNome(String n) { nome = n; }
}
```

Class "Pessoa" in Java

Declaração da clase `Pessoa`

Declaração do campo `idade` **de tipo** `int`

```
class Pessoa {
  String nome = "";
  int idade = 0;

  int getIdade() { return idade; }

  String getNome() { return nome;}

  void setIdade(int i) { idade = i; }

  void setNome(String n) { nome = n; }
}
```

Class "Pessoa" in Java

Declaração da clase `Pessoa`

Declaração do campo `idade` **de tipo** `int`

```
class Pessoa {
   String nome = "";
   int idade = 0;

   int getIdade() { return idade; }

   String getNome() { return nome;}

   void setIdade(int

   void setNome(String n) { nome = n; }
}
```

Declaração do método `setNome`, **cuja "signature" é :** `String -> void`

Classes as Types

Every type should be based on a class

- **Number**
- **Integer**
- **Short**
- **Float**
- **Double**

Classes as Modules

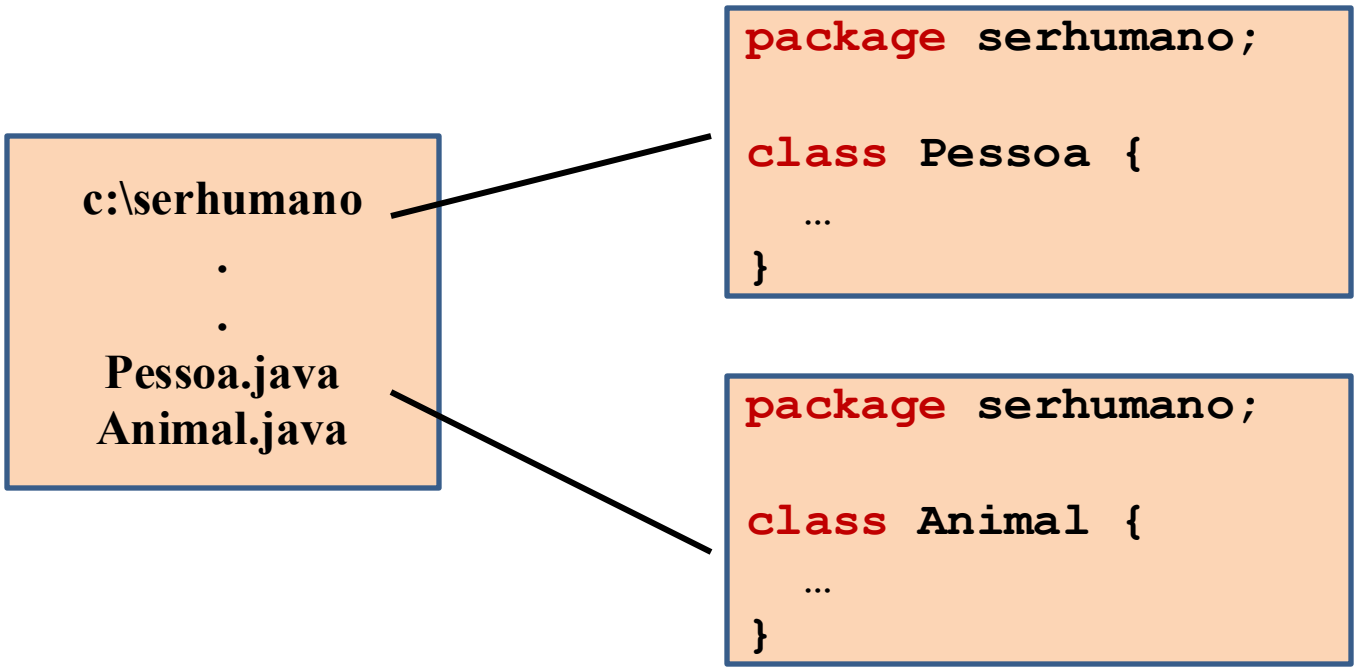
- Classes can be group into **clusters**
- **Clusters** barely correspond to the idea of `package` en Java

`package` Construct of **Java**

```
package serhumano;

class Pessoa {
   …
}
```

```
package serhumano;

class Animal {
   …
}
```

`package` Construct of **Java**



Computational Mechanism

- Objects are instances of classes
- In Object-Oriented Programming there is only one basic computational mechanism:

  "Given a certain object, that is, an instance of a class, call a method of that class on that object"

Computational Mechanism

```
class Pessoa {
  String nome = "";

  void setNome(String n) { nome = n; }
  String getNome() { return nome;}
}
```

Computational Mechanism

```
class Pessoa {
  String nome = "";

  void setNome(String n) { nome = n; }
  String getNome() { return nome;}
}

class Cliente {
  void imprimirNomePessoa(){
    Pessoa p = new Pessoa();
    p.setNome("Paulo");
    System.out.println(p.getNome());
  }
}
```

Computational Mechanism

```
class Pessoa {
  String nome = "";

  void setNome(String n) { nome = n; }
  String getNome() { return nome;}
}

class Cliente {
  void imprimirNome() {
    Pessoa p = new Pessoa();
    p.setNome("Paulo");
    System.out.println(p.getNome());
  }
}
```

Create object `p` of class `Pessoa`

Computational Mechanism

```
class Pessoa {
  String nome = "";

  void setNome(String n) { nome = n; }
  String getNome() { return nome;}
}

class Cliente {
  void imprimirNome() {
    Pessoa p = new Pessoa();
    p.setNome("Paulo");
    System.out.println(p.getNome());
  }
}
```

Create object `p` of class `Pessoa`

Computational Mechanism!!
Calling method `setNome` on object `p`

Class Constructor

- Class **constructors** provide a way to create and initilialize objects
- The declaration of a **constructor** is similar to declaration of a method except that :
  1. A constructor has the same name as the class
  2. A constructor does not return any value

Class Constructor

```
class Pessoa {
  String nome = "";

  Pessoa(String n) { nome = n; }
  void setNome(String n) { nome = n; }
  String getNome() { return nome;}
}

class Cliente {
  void imprimirNomePessoa(){
    Pessoa p = new Pessoa("Paulo");
    System.out.println(p.getNome());
  }
}
```

Class Constructor

```
class Pessoa {
  String n
  Pessoa(String n) { nome = n; }
  void setNome(String n) { nome = n; }
  String getNome() { return nome;}
}

class Cliente {
  void imprimirNomePessoa(){
    Pessoa p = new Pessoa("Paulo");
    System.out.println(p.getNome());
  }
}
```

Constructor of class `Pessoa`

Class Constructor

```
class Pessoa {
  String n
  Pessoa(String n) { nome = n; }
  void setNome(String n) { nome = n; }
  String getNome() { return nome;}
}

class Cliente {
  void imprimirNomePessoa(){
    Pessoa p = new Pessoa("Paulo");
    System.out.println(p.getNome());
  }
}
```

Constructor of class `Pessoa`

Creating an object of class Pessoa with `nome` equals to `"Paulo"`

Information Hiding

It should be possible for the author of a class to specify that a feature is available to all the clients, to no clients or to specific clients

- A **method** or a **field** can be used for internal computations of a class, so it might not be necessary for the method or the field to be visible for the clients of the class

Information Hiding

```
class Pessoa {
          String nome = "";

  public Pessoa( ) { nome = ""; }
  void setNome(String n) { nome = n; }
         String getNome() { return nome;}
}
```

Information Hiding

```
class Pessoa {
  private String nome = "";

  public Pessoa( ) { nome = ""; }
  void setNome(String n) { nome = n; }
  public String getNome() { return nome;}
}
```

Information Hiding

```
class Pessoa {
  private String nome = "";

  public Pessoa( ) { nome = ""; }
  void setNome(String n) { nome = n; }
  public String getNome() { return nome;}
}
class Cliente {
  void imprimirNomePessoa(){
    Pessoa p = new Pessoa();

  }
}
```

Information Hiding

```
class Pessoa {
  private String nome = "";

  public Pessoa( ) { nome = ""; }
  void setNome(String n) { nome = n; }
  public String getNome() { return nome;}
}
class Cliente {
  void imprimirNomePessoa(){
    Pessoa p = new Pessoa();
    p.nome = "Paulo";
  }
} // Compilation Error
```

Information Hiding

```
class Pessoa {
  private String nome = "";

  public Pessoa( ) { nome = ""; }
  void setNome(String n) { nome = n; }
  public String getNome() { return nome;}
}
class Cliente {
  void imprimirNomePessoa(){
    Pessoa p = new Pessoa();

  }
}
```

Information Hiding

```
class Pessoa {
  private String nome = "";

  public Pessoa( ) { nome = ""; }
  void setNome(String n) { nome = n; }
  public String getNome() { return nome;}
}
class Cliente {
  void imprimirNomePessoa(){
    Pessoa p = new Pessoa();
    p.setNome("Paulo");
  }
} // O.K
```

Exception Handling

The language should provide a mechanism to recover from unexpected abnormal situations

- Abnormal Situations:
  - Division by 0
  - **NullPointerException**
  - **IndexOutOfBoundException**
- **Exception Handling** is related to **Robustness**

Exception Handling

```
class Pessoa {
   int idade = 0;
     ...

   void setIdade(int i) {

      idade = i;

   }
}
```

Exception Handling

```
class Pessoa {
   int idade = 0;
     ...

   void setIdade(int i) throws PessoaException{
     if(i >= 0) {
       idade = i;
     }
     else throw new PessoaException();
   }
}
```

Static Typing

A well-defined type system should, by enforcing a number of type declaration and compatibility rules, guarantee the run-time type safety of the system it accepts

- For a typed language, it becomes possible to construct a **static type checker** that accepts **correctly typed** programs and rejects **incorrectly typed** ones

Genericity

It should be possible to write classes with formal generic parameters representing arbitrary types

- `T` represents a generic type
  - `List[T]`
  - `Stack[T]`
  - `Vector[T]`

Genericity

```
public class Vector {
 Object elementData[];
 int elementCount;

 public Object get(int index) {
   if(index >= elementCount)
     throw new ArrayIndexOutOfBoundsException(index);
   return elementData[index];
 }
}
```

Genericity

```
public class Vector <T> {
 Object[] elementData;
 int elementCount;

 public <T> get(int i) {
   if(index >= elementCount)
     throw new ArrayIndexOutOfBoundsException(i);
   return <T> elementData[i];
 }
}
```

Inheritance

It should be possible to define a class as inheriting from another

- **Inheritance** is one of the key concepts in object-oriented methodologies
- If class **`B`** inherits from class **`A`**, then **`A`** is called **ancestor** and **`B`** **descendent**

Inheritance

```
class Pessoa {
   String nome = "", sobrenome = "";
   int idade = 0;

   int getIdade(){return idade;}
   String getNome(){return nome;}
   String getSobreNome(){return sobrenome;}
   String getNome_E_Sobrenome()  {
      return nome +" " +sobrenome;
   }
   ...
```

Inheritance

```
   ...
   void setIdade(int i) {idade = i;}
   void setNome(String n) {nome = n;}
   void setSobrenome(String sn){sobrenome = sn;}
   void setNome_E_Sobrenome(String n, String sn){
     nome = n;
     sobrenome = sn;
   }
}
```

Inheritance

```
class Professor extends Pessoa {
   String actividade_profissional;

   void setActividadeProfissional(String ap){
    actividade_profissional = ap;
   }
   String getActividadeProfissional(){
    return actividade_profissional;
   }
}
```

Inheritance

Class `Professor` intherits from class `Pessoa`

```
class Professor extends Pessoa {
   String actividade_profissional;

   void setActividadeProfissional(String ap){
    actividade_profissional = ap;
   }
   String getActividadeProfissional(){
    return actividade_profissional;
   }
}
```

Inheritance

```
class Cliente {
  public static void main(String[] args) {
   Professor pr = new Professor();
   pr.setNome("Nestor");
   pr.setSobreNome("Catano");
   pr.setActividadeProfissional("Prof. Assistente");

   System.out.println(pr.getNome_E_Sobrenome());
   System.out.println(pr.getActividadeProfissional());
  }
}
```

Inheritance

```
class Cliente {
  public static v                                    gs) {
    Professor pr =
    pr.setNome("Nestor");
    pr.setSobreNome("Catano");
    pr.setActividadeProfissional("Prof. Assistente");

    System.out.println(pr.getNome_E_Sobrenome());
    System.out.println(pr.getActividadeProfissional());
   }
}
```

Method `setNome` is inherited from class `Pessoa`

Deferred Methods and Classes

It should be possible to write a class or a feature as deferred, that is to say specified but not fully implemented

- Methods and classes **partially implemented** are particularly important for the **analysis** and **design** of software
- **Abstract methods** and **abstract classes**

Abstract Methods and Abstract Classes

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
```

Abstract Methods and Abstract Classes

FormGeometrica is an abstract class

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
```

Abstract Methods and Abstract Classes

`FormGeometrica` **is an abstract class**

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
```

`area()` **is an abstract method**

Polymorphism

It is the ability for an entity to become attached to objects of various possible types

- The word **entity** refers to:
  - The **name** of a **variable**
  - The **name** of a **method**
  - The **name** of a **function**

Overridden Methods

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
class Quadrado extends FormaGeometrica {
  Quadrado(double lado) {
    comprimento = lado;
  }

  double area() {
   return comprimento * comprimento;
  }
}
```

Overridden Methods

FormaGeometrica is an abstract class

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
class Quadrado extends FormaGeometrica {
  Quadrado(double lado) {
    comprimento = lado;
  }

  double area() {
   return comprimento * comprimento;
  }
}
```

Overridden Methods

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
class Quadrado extends FormaGeometrica {
  Quadrado(double lado) {
    comprimento = lado;
  }

  double area() {
   return comprimento * comprimento;
  }
}
```

`area()` is an abstract method

Overridden Methods

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
class Quadrado extends FormaGeometrica {
  Quadrado(double lado) {
    comprimento = lado;
  }

  double area() {
   return comprimento * comprimento;
  }
}
```

Constructor of class `Quadrado`

Overridden Methods

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
class Quadrado extends FormaGeometrica {
  Quadrado(double lado) {
    comprimento = lado;
  }

  double area() {
   return comprimento * comprimento;
  }
}
```

Initialize field `comprimento` inherited from `FormaGeometrica`

Overridden Methods

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
class Quadrado extends FormaGeometrica {
  Quadrado(double lado) {
    comprimento = lado;
  }

  double area() {
   return comprimento * comprimento;
  }
}
```

Implementation of method `area()` inherited from class `FormaGeometrica`

Overridden Methods

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
class Quadrado extends FormaGeometrica {
  Quadrado(double lado) {
    comprimento = lado;
  }

  double area() {
   return comprimento * comprimento;
  }
}
```

Overridden Methods

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
class Circulo extends FormaGeometrica {
  Circulo(double raio) {
    comprimento = raio;
  }

  double area() {
   return getPI()*comprimento*comprimento;
  }
}
```

Overridden Methods

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;

  abstract double area();
  double getPI(){ return PI; }
}
class Circulo extends FormaGeometrica {
  Circulo(double raio) {
    comprimento = raio;
  }

  double area() {
   return getPI()*comprimento*comprimento;
  }
}
```

`area()` is a **polymorphic method**: it has a different implementation in class `Quadrado` than in class `Circulo`

Redefinition

It should be possible to redefine the specification, signature and implementation of an inherited method or field

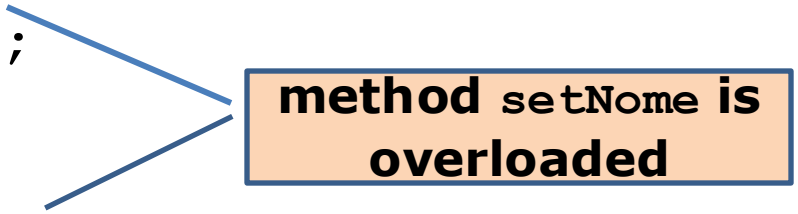
- In O-O terminology **redefinition** is called **overloading**
- **signature** refers to the parameters of a method and the value that it returns

Overloaded Methods

```
class Pessoa {
  String nome = "", sobrenome = "";
  int idade = 0;

  void setNome(String n) {
   nome = n;
  }

  void setNome(String n, String sn) {
    nome = n;
    sobrenome = sn;
  }
}
```

Overloaded Methods

```
class Pessoa {
  String nome = "", sobrenome = "";
  int idade = 0;

  void setNome(String n) {
   nome = n;
  }

  void setNome(String n, String sn) {
    nome = n;
    sobrenome = sn;
  }
}
```



Multiple Inheritance

It should be possible for a class to inherit from as many others as necessary, with an adequate mechanism for disambiguating name clashes

- **Name resolution problem**: when inherited fields or methods from different classes have the same name

Multiple Inheritance

- Java does not provide support to multiple inheritance **directly**
- Java provides **indirect** support to multiple inheritance through the use of **interfaces**
- A Java `interface` is a class
  - All the class methods are `abstract`
  - No fields
  - Constants

Multiple Inheritance
interface

```
public interface Traslacao {
   abstract void mover(double d);
   abstract double distancia_origem();
}

abstract class FormaGeometrica {
   double PI = 3.1416;
   double comprimento;

   abstract double area();
   double getPI(){ return PI; }
}
```

Multiple Inheritance interface

```
class Circulo extends FormaGeometrica
                implements Traslacao {
  double coordenadax, coordenaday;

  Circulo(double x, double y, double raio) {
    coordenadax = x;
    coordenaday = y;
    comprimento = raio;
  }
  double area() {
   return getPI()*comprimento*comprimento;
  }
  ...
```

Multiple Inheritance interface

```
class Circulo extends FormaGeometrica
                implements Traslacao
  double coordenadax, coordenaday;

  Circulo(double x, double y, double raio) {
    coordenadax = x;
    coordenaday = y;
    comprimento = raio;
  }
  double area() {
   return getPI()*comprimento*comprimento;
  }
  ...
```

Multiple inheritance in Java

Multiple Inheritance
interface

```
  ...
  public void mover(double d) {
    coordenadax += d;
    coordenaday += d;
  }
  public double distancia_origem(){
    sqrt(pow(coordenadax,2) +
         pow(coordanaday,2));
  }
}
```

Dynamic Binding

Calling a feature on an entity should always trigger the feature corresponding to the type of the attached run-time object, which is not necessarily the same in different executions of the call

- **Dynamic Binding** influences the way programmers construct their programs
- Calling a methods might have several meanings

Dynamic Binding

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;
  abstract double area();
}

class Quadrado extends FormaGeometrica {
  Quadrado(double lado) {
    comprimento = lado;
  }

  double area() {
   return comprimento * comprimento;
  }
}
```

Dynamic Binding

```
abstract class FormaGeometrica {
  double PI = 3.1416;
  double comprimento;
  abstract double area();
}

class Circulo extends FormaGeometrica {
  Circulo(double raio) {
    comprimento = raio;
  }

  double area() {
   return PI*comprimento*comprimento;
  }
}
```

Dynamic Binding

```
public class Test {
  public static void main(String[] args) {
    FormaGeometrica fg;
    Quadrado q = new Quadrado(10.0);
    Circulo c = new Circulo(10.0);

    fg = q;
    System.out.println("Area do quadrado é " +fg.area());

    fg = c;
    System.out.println("Area do círculo é " +fg.area());
  }
}
```

Dynamic Binding

```
public class Test {
  public static void main(String[] args) {
    FormaGeometrica fg;
    Quadrado q = new Quadrado(10.0);
    Circulo c = new Circulo(10.0);

    fg = q;
    System.out.println("Area do quadrado é " +fg.area());

    fg = c;
    System.out.println("Area do círculo é " +fg.area());

  }
}
```

Variable declaration:
no object is created

Dynamic Binding

```
public class Test {
  public static void main(String[] args) {
    FormaGeometrica fg;
    Quadrado q = new Quadrado(10.0);
    Circulo c = new Circulo(10.0);

    fg = q;
    System.out.println("Area do quadrado é " +fg.area());

    fg = c;
    System.out.println("Area do círculo é " +fg.area());

  }
}
```

Variable declaration: no object is created

`fg` is dynamically linked to an object `q` of type `Quadrado`

Dynamic Binding

```
public class Test {
  public static void main(String[] args) {
    FormaGeometrica fg;
    Quadrado q = new Quadrado(10.0);
    Circulo c = new Circulo(10.0);

    fg = q;
    System.out.println("Area do quadrado é " +fg.area());

    fg = c;
    System.out.println("Area do círculo é " +fg.area());

  }
}
```

Variable declaration: no object is created

`fg` is dynamically linked to an object `q` of type `Quadrado`

`fg` is dynamically linked to an object `c` of type `Circulo`

Runtime Type Interrogation

It should be possible to determine at run time whether the type of an object conforms to a statically given type

- Because of the **Dynamic Binding**, sometimes is not possible to predict the type of an object
- The **Java** `instanceof` operator provides a way to check the type of an object in runtime

Runtime Type Interrogation

```
public class Test {
  public static void main(String[] args) {
    ...
    if(fg instanceof Quadrado)
      System.out.println("Area do quadrado é "
                         +fg.area());

    if(fg instanceof Circulo)
      System.out.println("Area do círculo é "
                         +fg.area());
  }
}
```

The object `this` in **Java**

- **`this`** means **"the current object"**
- **`this`** can be used inside any non-static method or within a constructor

The object this in Java

```
class Ponto {
  int a, b;
  Ponto(int i, int j) {
    a = i;
    b = j;
  }
}
class Cliente {
  public static void main(String[] args) {
    Ponto p = new Ponto(3, 5);
  }
}
```

The object this in Java

```
class Ponto {
  int a, b;
  Ponto(int i, int j) {
    this.a = i;
    this.b = j;
  }
}
class Cliente {
  public static void main(String[] args) {
    Ponto p = new Ponto(3, 5);
  }
}
```

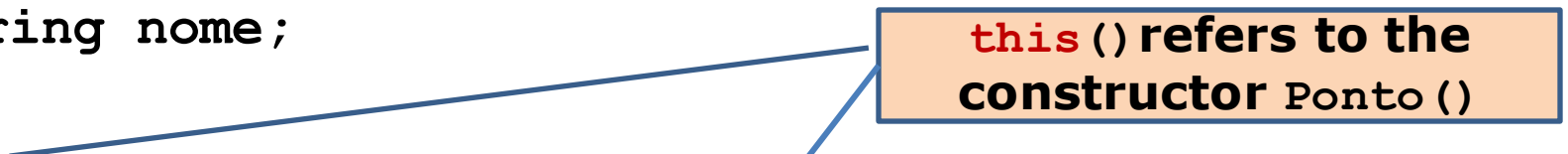
this is the object p
this.a means "the field a of the objec p"

The object this in Java

```
class Ponto {
  int a, b;
  String nome;

  Ponto(){ nome="?"; }
  Ponto(String s){ nome=s; }
  Ponto(int i, int j){ this(); a=i; b=j;}
  Ponto(int i, int j, String s){ this(s); a=i; b=j;}
}
```

The object this in Java

```
class Ponto {
  int a, b;
  String nome;

  Ponto(){ nome="?"; }
  Ponto(String s){ nome=s; }
  Ponto(int i, int j){ this(); a=i; b=j;}
  Ponto(int i, int j, String s){ this(s); a=i; b=j;}
}
```



The object `this` in **Java**

```
class Ponto {
  int a, b;
  String nome;

  Ponto(){ nome="?"; }
  Ponto(String s){ nome=s; }
  Ponto(int i, int j){ this(); a=i; b=j;}
  Ponto(int i, int j, String s){ this(s); a=i; b=j;}
}
```

`this()` refers to the constructor `Ponto()`

`this(s)` refers to the constructor `Ponto(String)`

The method `toString()` in **Java**

- In **Java**, every class inherits from class `Object`
- It is possible to override the definition of method `toString()` in class `Object`
- The method `toString()` is used for printing the internal representation of an object

The method `toString()` in **Java**

```
class Ponto {
  int a, b;
  String nome;
  Ponto(int i, int j, String s){ nome=s; a=i; b=j; }
}
```

The method `toString()` in **Java**

```
class Ponto {
  int a, b;
  String nome;
  Ponto(int i, int j, String s){ nome=s; a=i; b=j; }
  public String toString() {
   return nome +"(" +a +"," +b +")";
  }
}
```

The method `toString()` in **Java**

```
class Ponto {
  int a, b;
  String nome;
  Ponto(int i, int j, String s){ nome=s; a=i; b=j; }
  public String toString() {
   return nome +"(" +a +"," +b +")";
  }
}
class Cliente {
  public static void main(String[] args) {
    Ponto p1 = new Ponto(3, 5, "p1");
    System.out.println(p1);
  }
}
```

The method `toString()` in **Java**

```
class Ponto {
  int a, b;
  String nome;
  Ponto(int i, int j, String s){ nome=s; a=i; b=j; }
  public String toString() {
   return nome +"(" +a +"," +b +")";
  }
}
class Cliente {
  public static void main(String[] args) {
    Ponto p1 = new Ponto(3, 5, "p1");
    System.out.println(p1);
  }
}
```

```
System.out.println(p1.toString());
```

this and toString()

```
class Ponto {
  int a, b;
  String nome;
  Ponto(int i, int j, String s){ nome=s; a=i; b=j; }
  public String toString() {
   return nome +"(" +a +"," +b +")";
  }
  void imprimir(){ System.out.println("ponto: " +this);}
}
```

this and toString()

```
class Ponto {
  int a, b;
  String nome;
  Ponto(int i, int j, String s){ nome=s; a=i; b=j; }
  public String toString() {
   return nome +"(" +a +"," +b +")";
  }
  void imprimir(){ System.out.println("ponto: " +this);}
}
class Cliente {
  public static void main(String[] args) {
    Ponto p1 = new Ponto(3, 5, "p1");
    p1.imprimir();
  }
}
```