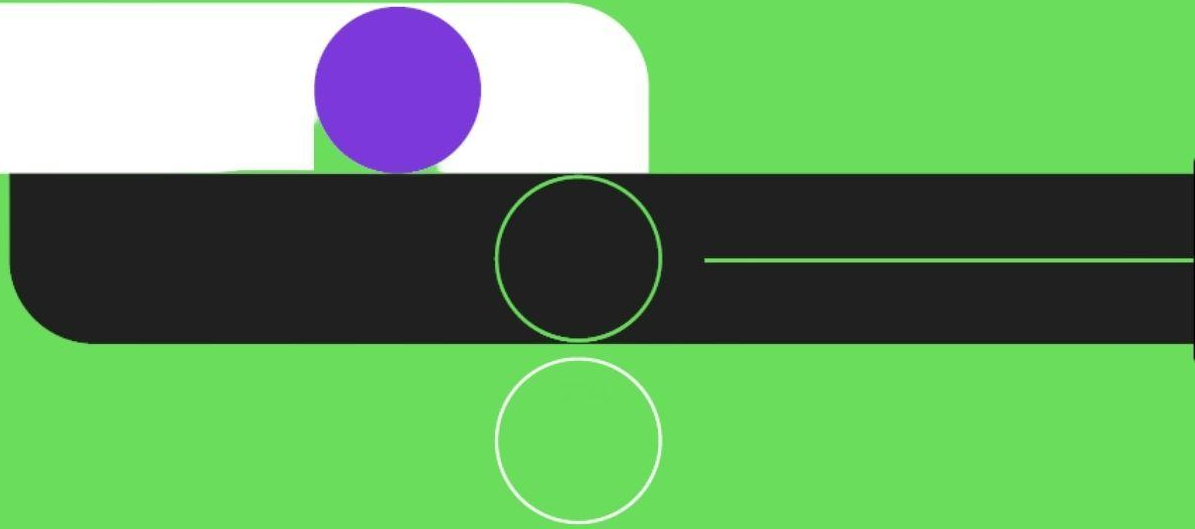
GAMA ACADEMY

Encapsulamento, Herança e Polimorfismo

Pilares POO



01 Modificadores de acesso

02 Encapsulamento

03 Herança

04 Polimorfismo

Modificadores de acesso Java

Os modificadores de acesso auxiliam na organização dos componentes da sua aplicação ao tornar os membros das classes mais ou menos acessíveis por outras partes do seu programa.

Os modificadores de acesso em ordem do mais restritivo para o menos restritivo são:

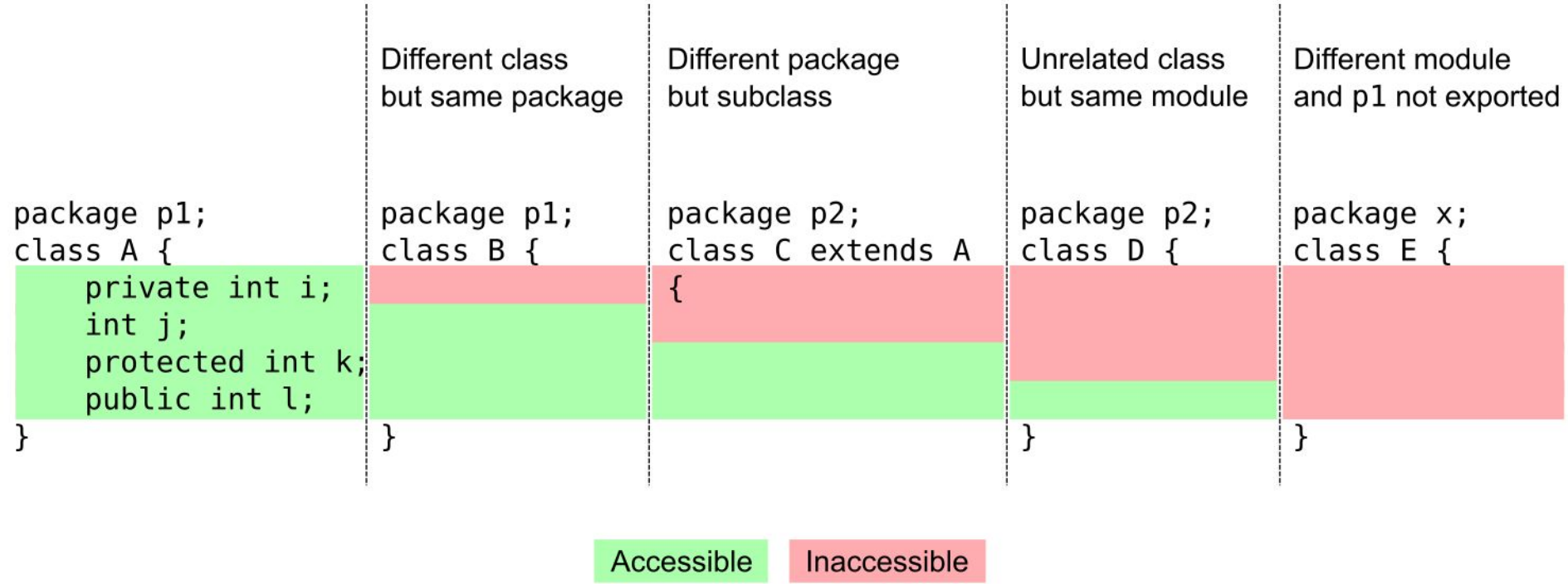
1 - private
2 - padrão ou acessibilidade de pacote
3 - protected
4 - public

Modificadores de acesso Java

public	O modificador de acesso public é o menos restritivo de todos. Ele permite que qualquer outra parte da sua aplicação tenha acesso ao componente marcado como public.
protected	Os membros das classes marcados com o modificador de acesso protected serão acessíveis por classes e interfaces dentro do mesmo pacote e por classes derivadas mesmo que estejam em pacotes diferentes.
padrão (package)	O modificador de acesso padrão é o modificador atribuído aos membros da classe que não foram marcados explicitamente com um outro modificador de acesso. Membros com acessibilidade de pacote só podem ser acessados por outras classes ou interfaces definidas dentro do mesmo pacote.
private	O modificador de acesso private é o mais restritivo modificador de acesso. Todo membro de uma classe definido com o modificador private só é acessível para a própria classe.

Modificadores de acesso Java

		Mesmo Pacote	Pacotes diferentes
public	Classes Derivadas	ok	ok
	Classes não relacionadas	ok	ok
protected	Classes Derivadas	ok	ok
	Classes não relacionadas	ok	✗
padrão(package)	Classes Derivadas	ok	✗
	Classes não relacionadas	ok	✗
private	Classes Derivadas	✗	✗
	Classes não relacionadas	✗	✗


Different class but same package
Different package but subclass
Unrelated class but same module
Different module and p1 not exported
package p1;
class A {
private int i;
int j;
protected int k;
public int l;
}
package p1;
class B {
}
package p2;
class C extends A
{
package p2;
class D {
}
package x;
class E {
}
Accessible
Inaccessible

Encapsulamento

Encapsulamento

Ocultar partes independentes da implementação, permitindo construir partes invisíveis ao mundo exterior.

Encapsular não é obrigatório, mas é uma boa prática para produzir Classes mais eficientes.

Objetos bem encapsulados produzem padrões e geram proteção aos seus produtos e aos seus usuários.

Lista de serviços fornecidos por um componente. É o contato com o mundo exterior, que define o que pode ser feito com um objeto dessa classe.

Encapsulamento

Benefícios do Encapsulamento

Modularidade	Objeto pode ser escrito e mantido independentemente de outros objetos, o que permite que cada objeto seja utilizado livremente no sistema.
Ocultação de informações	Mesmo com a possibilidade de comunicação via interface, o objeto pode manter informações privadas e métodos podem ser modificados em qualquer momento sem afetar os outros objetos que dependem dele.
Facilitar reutilização de códigos	Uma Classe bem encapsulada é possível a reutilização até em outros softwares

Encapsulamento

interface Controle
    //Metodos Abstratos
publico abstrato Metodo **ligar()**
publico abstrato Metodo **desligar()**
publico abstrato Metodo **acelerar()**
publico abstrato Metodo **frear()**
publico abstrato Metodo ...

classe Carro **implementa** Controlador
    //sobrescrevendo Métodos

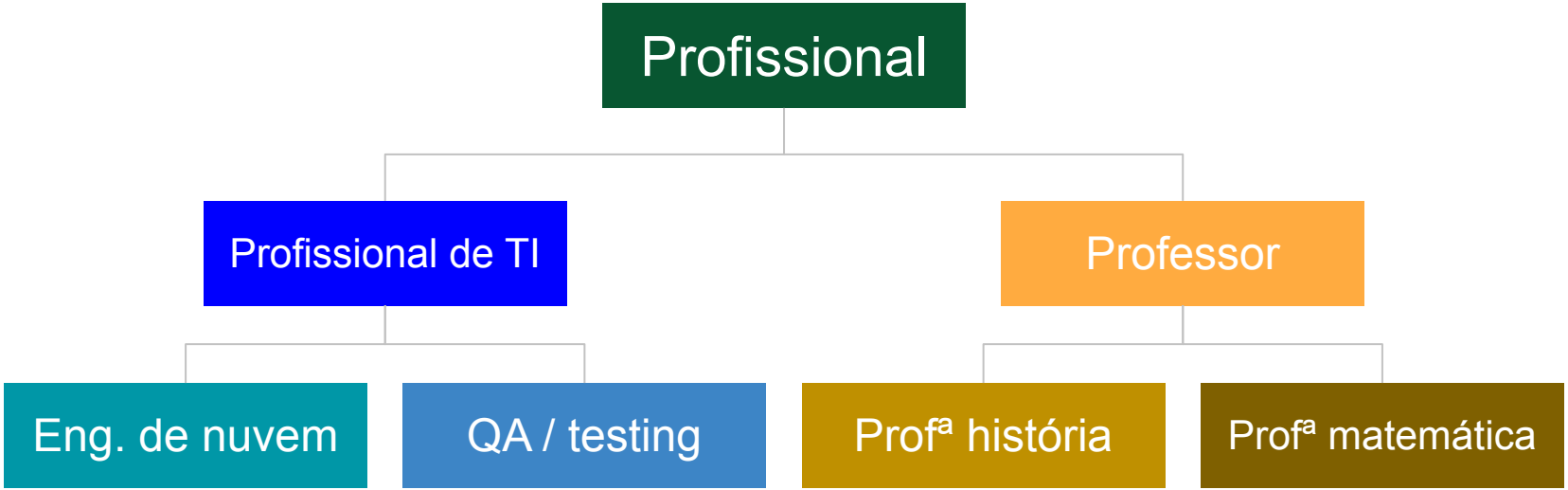
publico Metodo **ligar()**
    setLigado(verdadeiro)
publico Metodo **desligar()**
    setLigado(falso)
publico Metodo **acelerar()**

publico Metodo **frear()**

Herança

Herança

Herança é a capacidade de uma subclasse, de ter acesso às propriedades da superclasse, a ela relacionada. Ela permite basear uma nova classe na definição de uma outra classe previamente existente.



Tipos de Herança

Herança de implementação: Utiliza características e comportamentos da superclasse ou das classes ancestrais, porém não implemente nenhum novo método.

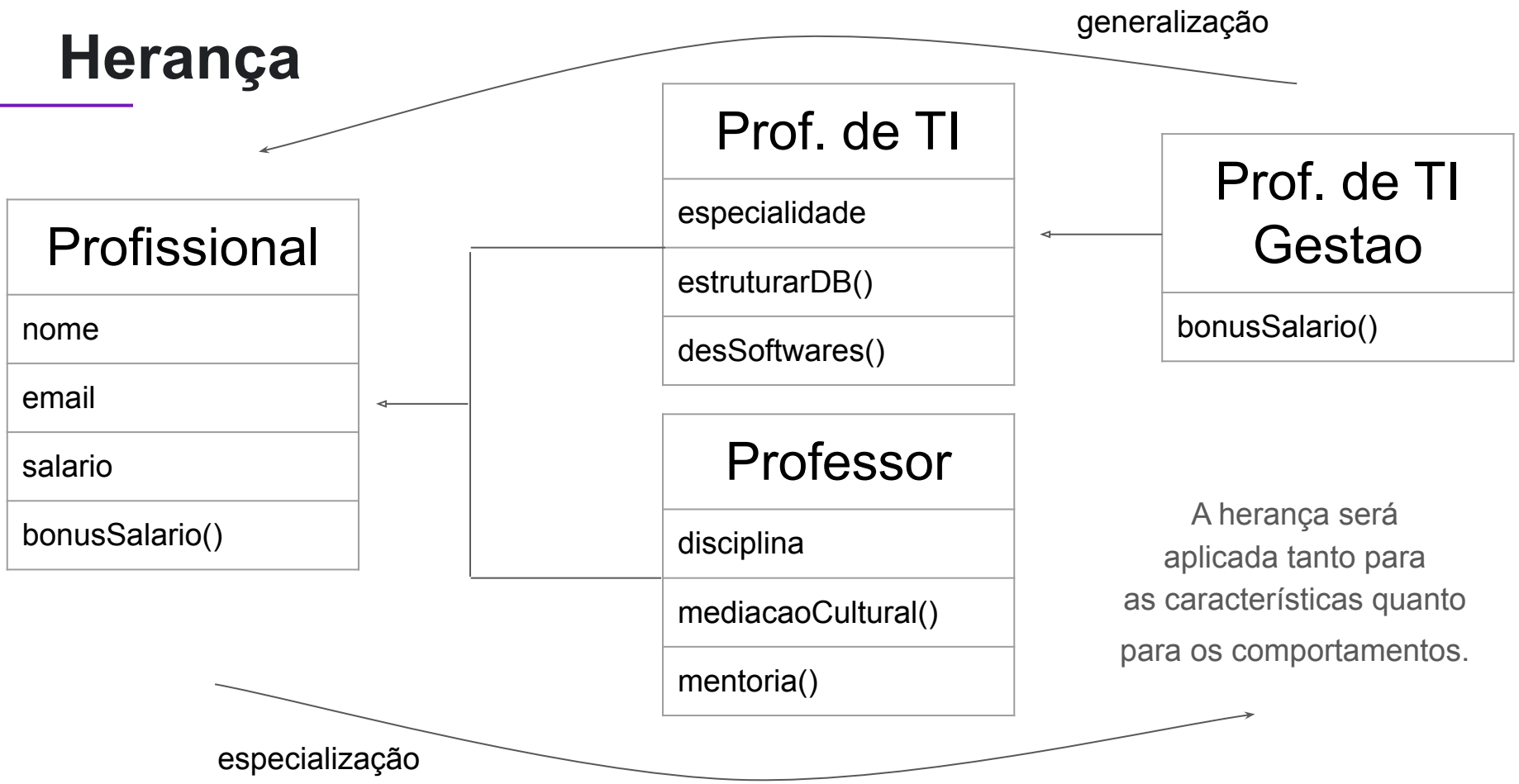
Herança para diferença: Além de utilizar características e comportamentos da superclasse, implementa novos atributos e/ou procedimentos.

Herança

Prof. de TI
nome
email
especialidade
salario
bonusSalario()
estruturarDB()
desSoftwares()

Professor
nome
email
disciplina
salario
bonusSalario()
mediacaoCultural()
mentoria()

Herança


generalização
Profissional
nome
email
salario
bonusSalario()
Prof. de TI
especialidade
estruturarDB()
desSoftwares()
Professor
disciplina
mediacaoCultural()
mentoria()
Prof. de TI Gestao
bonusSalario()
A herança será aplicada tanto para as características quanto para os comportamentos.
especialização

Tipos de Classes e Métodos

Classe Abstrata: Serve de modelo para outras **classes**. Ela sempre será uma superclasse genérica, e suas subclasses serão mais específicas

Método Abstrato: Método de uma classe abstrata que não possui implementação.

Classe Final: Não pode ser estendida, ou seja, não poder ter subclasses.

Método Final: Não pode ser sobrescrito pelas subclasses. Ele é obrigatoriamente herdado.

Polimorfismo

Polimorfismo

POLI = muitas
MORFO = formas

= Muitas formas de se fazer alguma coisa

Permite que classes pertencentes a uma mesma linha de herança possuam comportamentos diferentes para o mesmo método.

Exemplo simples: em um jogo de xadrez temos várias peças, cada peça se movimenta, porém cada tipo de peça se movimenta de uma maneira diferente (o peão vai pra frente, o cavalo anda em L, o bispo em diagonal e assim por diante)

• Desta maneira, todos (peão, cavalo e bispo) são peças, porém cada um se movimenta de uma maneira diferente!

Polimorfismo

O polimorfismo em Java se manifesta apenas na chamada de métodos

• Então ao passar uma mensagem para um objeto peça, dizendo para ele se mover, o Java identifica qual o tipo de peça é e fará o movimento de acordo com o tipo

```
public abstract class Peca {
        public abstract void mover();
}
```

```
public class  Peao extends Peca {
        @Override
        public  void mover() {
                System.out.println(“Andar para frente”);
        }
}
```

Polimorfismo

```
public abstract class Peca {
      public abstract void mover();
}
```

```
public class  Peao extends Peca {
      @Override
      public  void mover() {
            System.out.println(“Andar para frente”);
      }
}
```

```
public class  Cavalo extends Peca {
      @Override
      public  void mover() {
            System.out.println(“Andar em “L””);
      }
}
```

```
public class Bispo extends Peca {
      @Override
      public  void mover() {
            System.out.println(“Andar em diagonal”);
      }
}
```

GAMA ACADEMY

VAMOS PRATICAR?