Classes e Herança

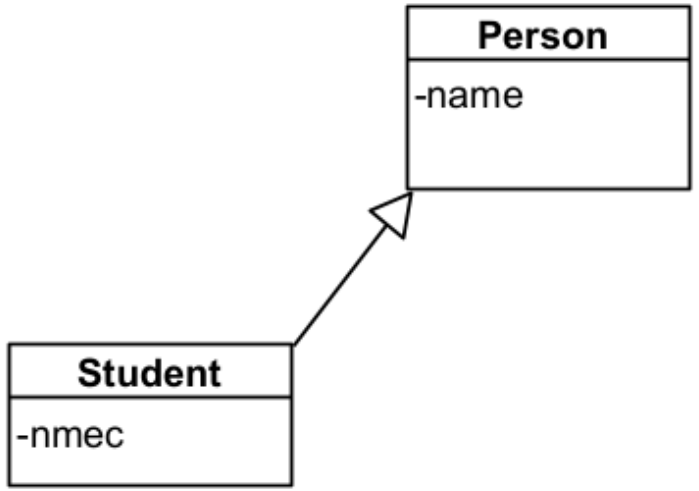
UA.DETI.POO

Relações entre Classes

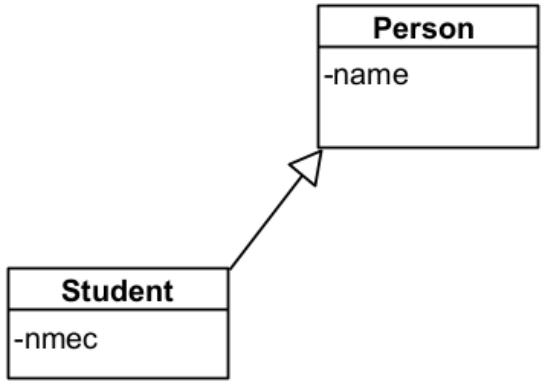
- Parte do processo de modelação em classes consiste em:
  - Identificar entidades candidatas a classes
  - Identificar relações entre estas entidades
- As relações entre classes identificam-se facilmente recorrendo a alguns modelos reais.
  - Por exemplo, um RelógioDigital e um RelógioAnalógico são ambos tipos de Relógio (especialização ou herança).
  - Um RelógioDigital, por seu lado, contém uma Pilha (agregação / composição).
- Relações:
  - IS-A
  - HAS-A

Herança (IS-A)

- IS-A indica especialização (herança) ou seja, quando uma classe é um sub-tipo de outra classe.
- Por exemplo:
  - Pinheiro é uma (IS-A) Árvore.
  - Um RelógioDigital é um (IS-A) Relógio.



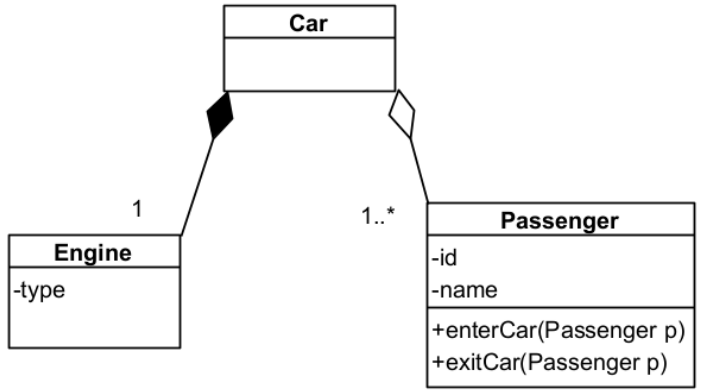
Herança (IS-A)



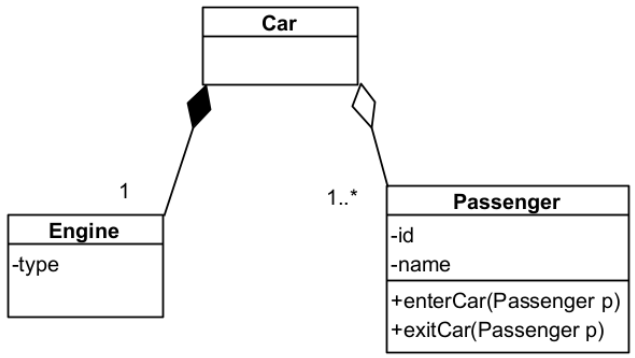

```
public class Person {
    private String name;

    // Constructor to initialize the Person class
    public Person(String name) {
        this.name = name;
    }

    public String getName() {        return name;    }
    public void setName(String name) {        this.name = name;    }

}

public class Student extends Person {
    private static int idCounter = 0; private int studentId;

    public Student(String name) {
        super(name);  // Call the superclass (Person) constructor
        this.studentId = ++idCounter;  // Increment and assign the unique ID
    }
    public int getStudentId() {        return studentId;    }
}
```

Agregação e composição (HAS-A)

- HAS-A indica que uma classe é contem por objetos de outra classe.
- Composição – parte integral
  - O carro (Car) tem um motor (Engine)
  - O motor (Engine) não faz sentido for do contexto do carro (Car)
- Agregação – pode ter / estar relacionado com
  - O carro (Car) pode ter passageiros (Passenger)
  - Os passageiros não dependem do carro para existir
- **Composição é mais forte que a agregação**



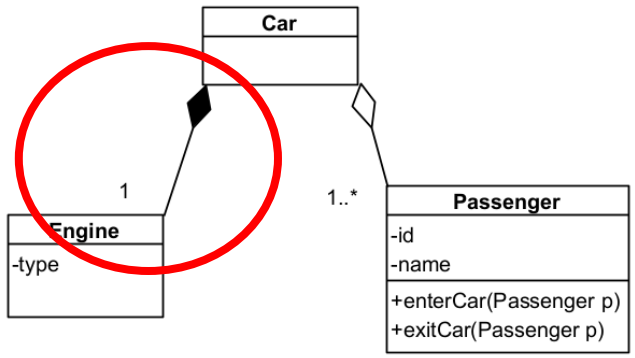
Has-A using collections / arrays




```
class Engine {

    private String type;

    public Engine(String type) {
        this.type = type;
    }

    public String getType() {
        return type;
    }
}
```

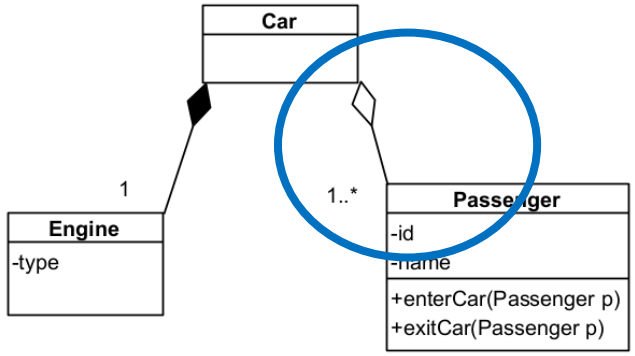
```
class Passenger {
    // Passenger properties
    private String name;
    private int id; // Unique identifier for each passenger


    public Passenger(String name, int id) {
        this.name = name;
        this.id = id;
    }

    public String getName() {        return name;    }
    public int getId() {        return id;    }

    // Override equals and hashCode to use id for equality
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Passenger passenger = (Passenger) o;
        return id == passenger.id;
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}
```

Has-A - Composição




```
class Car {

    private Engine engine;
    private List<Passenger> passengers;

    // Car constructor
    public Car(Engine engine) {
        this.engine = engine;
        this.passengers = new ArrayList<>();
    }
  (…)
}
```

Has-A Agregação usando collections




```
class Car {

    private Engine engine;
    private List<Passenger> passengers;

    // Car constructor
    public Car(Engine engine) {
        this.engine = engine;
        this.passengers = new ArrayList<>();
    }
  (…)
```

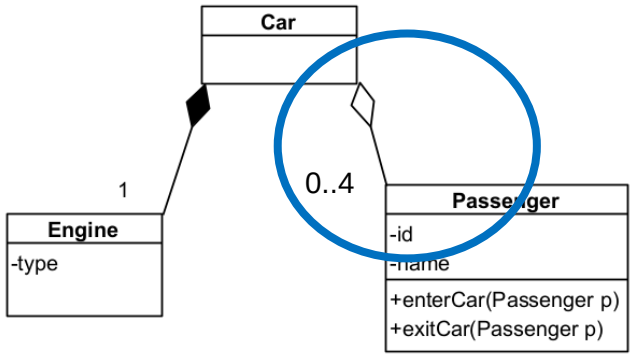
```
public void enterCar(Passenger passenger) {
        if (!passengers.contains(passenger)) {
            passengers.add(passenger);
        } else {
            System.out.println("A passenger with ID " + passenger.getId() + " is already in the car.");
        }
 }

public void exitCar(int passengerId) {
        Passenger tempPassenger = new Passenger("", passengerId); // Temporary passenger to leverage equals/hashCode
        if (passengers.remove(tempPassenger)) {
            System.out.println("Passenger with ID " + passengerId + " has exited the car.");
        } else {
            System.out.println("No passenger with ID " + passengerId + " found in the car.");
        }
}
```

Has-A gerida por quem tem

```
// Main class to run the program
public class Main {
    public static void main(String[] args) {
        // Create an Engine object
        Engine myEngine = new Engine("V8");

        // Create a Car object with the Engine
        Car myCar = new Car(myEngine);

        // Add passengers to the car
        myCar.enterCar(new Passenger("Alice", 101));
        myCar.enterCar(new Passenger("Bob", 102));
        myCar.displayCarDetails();

        // Try to add Alice again with the same ID
        myCar.enterCar(new Passenger("Alice", 101));

        // Bob exits the car
        myCar.exitCar(102);

        // Display the car details after changes
        myCar.displayCarDetails();
    }
}
```

Has-A gerida por quem tem

```
class Car {
    private Engine engine;
    private List<Passenger> passengers;
    private static final int MAX_PASSENGERS = 4;

    (…)
    // Car constructor
    public Car(Engine engine) {
        this.engine = engine;
        this.passengers = new ArrayList<>();
    }

    // Method to add a passenger to the car
    public void enterCar(Passenger passenger) {
        if (passengers.size() >= MAX_PASSENGERS) {        System.out.println("The car is full.");
            return;
        }
        if (!passengers.contains(passenger)) {
            passengers.add(passenger);
            System.out.println(passenger.getName() + " has entered the car.");
        } else {    System.out.println("A passenger is already in the car.");
        }
    }

(…)
}
```



Número limitado de passageiros

Reutilização das classes

Reutilização de classes

- Sempre que necessitamos de uma classe, podemos:
  - Recorrer a uma classe já existente que cumpre os requisitos
  - Escrever uma nova classe a partir "do zero"
  - Reutilizar uma classe existente usando agregação / composição
  - Reutilizar uma classe existente através de herança

Identificação de Herança

- Sinais típicos de que duas classes têm um relacionamento de herança
  - Possuem aspetos comuns (dados, comportamento)
  - Possuem aspetos distintos
  - Uma é uma especialização da outra

- Exemplos:
  - Gato é um Mamífero
  - Circulo é uma Figura
  - Água é uma Bebida

Questões?

- Quais as relações entre:
  - Trabalhador, Motorista, Vendedor, Administrativo e Contabilista
  - Triângulo, Retângulo e Losango
  - Professor, Aluno e Funcionário
  - Autocarro, Viatura, Roda, Motor, Pneu, Jante

Questões?

- Represente os seguintes elementos (classes) bem como as suas relações (herança e composição)
  - Livro
  - Artigo
  - Jornal
  - Publicação
  - Autor
  - Periódico
  - Editora
  - Revista

Herança - Conceitos

- A herança é uma das principais características de POO
  - A classe *CDeriv* herda, ou é derivada, de *CBase* quando *CDeriv* representa um sub-conjunto de *Cbase*

- A herança representa-se na forma:

```
class CDeriv extends CBase { /* ... */ }
```

- *Cderiv* tem acesso aos dados e métodos de *CBase*
  - que não sejam privados em *Cbase*

- Uma classe base pode ter múltiplas classes derivadas mas uma classe derivada não pode ter múltiplas classes base
  - Em Java não é possível a herança múltipla

Herança - Exemplo

```
class Person {
        private String name;
        public Person(String n) { name = n; }
        public String name() { return name; }
        public String toString() { return "PERSON";}
}
class Student extends Person {
        private int nmec;
        public Student(String s, int n) { super(s); nmec=n; }
        public int num()   { return nmec; }
        public String toString() { return "STUDENT"; }
}
public class Test {
        public static void main(String[] args)  {
           Person p = new Person("Joaquim");
           Student stu = new Student("Andreia", 55678);
           System.out.println(p + " : " + p.name());
           System.out.println(stu + " : " + stu.name() + ", " + stu.num());
        }
}
```

Base

Derivada

PERSON : Joaquim

STUDENT : Andreia, 55678

Herança - Exemplo

```
class Art {
        Art() {
                System.out.println("Art constructor");
        }
}
class Drawing extends Art {
        Drawing() {
                System.out.println("Drawing constr.");
        }
}
public class Cartoon extends Drawing {
        Cartoon() {
                System.out.println("Cartoon constr.");
        }
        public static void main(String[] args) {
                Cartoon x = new Cartoon();
        }
}
```

Art constructor

Drawing constr.

Cartoon constr.

A construção é feita a partir da classe base

Referência super

- Quando temos uma classe derivada que estende a outra classe base, a classe base também se designa por "super classe".
- A referência **super** permite aceder às características, métodos, e construtores da "super classe" que estamos a estender.
- É uma referência à parte do objeto construída a partir da "super classe", não pode ser usada nos elementos estáticos da classe.
- As restrições/capacidades do **super** são muita parecidas à da referência **this**.

Construtores com parâmetros

- Em construtores com parâmetros o construtor da classe base é a primeira instrução a aparecer num construtor da classe derivada.

```
class Game {
        int num;
        Game(int code) { ... }
        // ...
}

class BoardGame extends Game {
  // ...
  BoardGame(int code, int numPlayers) {
         super(code);
    // ...
  }
}
```

Herança de Métodos

- Ao herdar métodos podemos:
  - mantê-los inalterados,
  - acrescentar-lhe funcionalidades novas ou
  - redefini-los

Herança de Métodos - herdar

```
class Person {
        private String name;
        public Person(String n) { name = n; }
        public String name() { return name; }
        public String toString() { return "PERSON";}
}
class Student extends Person {
        private int nmec;
        public Student(String s, int n) { super(s); nmec=n; }
        public int num()        { return nmec; }
}
public class Test {
        public static void main(String[] args)  {
          Student stu = new Student("Andreia", 55678);
          System.out.println(stu + " : " +
           stu.name() + ", " + stu.num());
        }
}
```

Herança de Métodos - redefinir

```
class Person {
        private String name;
        public Person(String n) { name = n; }
        public String name() { return name; }
        public String toString() { return "PERSON";}
}

class Student extends Person {
        private int nmec;
        public Student(String s, int n) { super(s); nmec=n; }
        public int num()        { return nmec; }
        public String toString() { return "STUDENT"; }
}
```

Herança de Métodos - estender

```
class Person {
        private String name;
        public Person(String n) { name = n; }
        public String name() { return name; }
        public String toString() { return "PERSON";}
}

class Student extends Person {
        private int nmec;
        public Student(String s, int n) { super(s); nmec=n; }
        public int num()        { return nmec; }
        public String toString()
          { return super.toString() + " STUDENT"; }
}
```

Herança e controlo de acesso

- Não podemos reduzir a visibilidade de métodos herdados numa classe derivada
  - Métodos declarados como public na classe base devem ser public nas subclasses
  - Métodos declarados como protected na classe base devem ser protected ou public nas subclasses. Não podem ser private
  - Métodos declarados sem controlo de acesso (default) não podem ser private em subclasses
  - Métodos declarados como private não são herdados

Final

- O classificador final indica "não pode ser mudado"
- A sua utilização pode ser feita sobre:
  - Dados - constantes

    ```
    final int i1 = 9;
    ```
  - Métodos - não redefiníveis

    ```
    final int swap(int a, int b) { //:
    }
    ```
  - Classes - não herdadas

    ```
    final class Rato { //...
    }
    ```
- "final" fixa como constantes atributos de tipos primitivos mas não fixa objetos nem vetores
  - nestes casos o que é constante é simplesmente a referência para o objeto

```
class Value {  int i = 1; }
public class FinalData {
  // Can be compile-time constants
  private final int i1 = 9;
  private static final int VAL_TWO = 99;
  // Typical public constant:
  public static final int VAL_THREE = 39;

  public final int i4 = (int)(Math.random()*20);
  public static final int i5 = (int)(Math.random()*20);

  private Value v1 = new Value();
  private final Value v2 = new Value();
  private final int[] a = { 1, 2, 3, 4, 5, 6 }; // Arrays

  public static void main(String[] args) {
    FinalData fd1 = new FinalData();
    //! fd1.i1++; // Error: can't change value
    fd1.v2.i++; // Object isn't constant!
    fd1.v1 = new Value(); // OK -- not final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Object isn't constant!
    //! fd1.v2 = new Value(); // Can't change ref
    //! fd1.a = new int[3];
  }
}
```

https://tinyurl.com/3j57tb7v

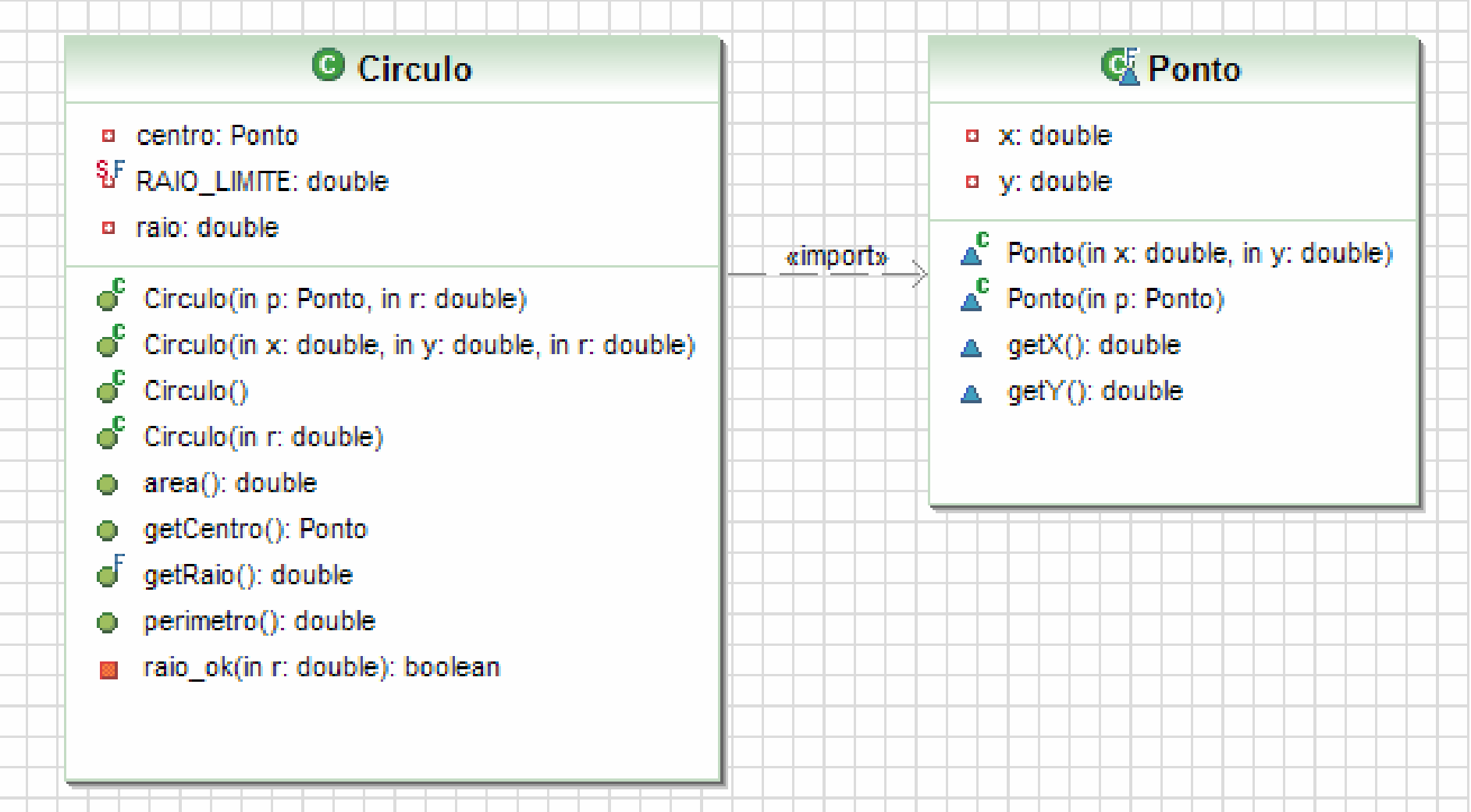
Exemplo – classe Ponto

```
public final class Ponto {
        private double x;
        private double y;

        public Ponto(double x, double y) { this.x=x; this.y=y; }
        public final double getX() { return(x); }
        public final double getY() { return(y); }
}
```

Exemplo – classe Circulo

```
public class Circulo {
    private Ponto centro;
    private double raio;

    public static final double RAIO_LIMITE = 100.0;

    public Circulo(Ponto p, double r) {
       centro = p;
       if (raio_ok(r)) raio = r; else raio = RAIO_LIMITE;
    }

    public double area() { return Math.PI*raio*raio;   }
    public double perimetro() { return 2*Math.PI*raio; }
    public final double getRaio() { return raio; }
    public final Ponto getCentro() { return centro; }

    private boolean raio_ok(double r) { return(r<=RAIO_LIMITE); }

}
```

Representação UML



Herança - Boas Práticas

- Programar para a interface e não para a implementação
- Procurar aspetos comuns a várias classes e promovê-los a uma classe base
- Minimizar os relacionamentos entre objetos e organizar as classes relacionadas dentro de um mesmo package
- Usar herança criteriosamente – sempre que possível favorecer a composição

Sumário - Porquê herança?

- Muitos objetos reais apresentam esta característica

- Permite criar classes mais simples com funcionalidades mais estanques e melhor definidas
  - Devemos evitar classes com interfaces muito "extensas"
- Permite reutilizar e estender interfaces e código

- Permite tirar partido do polimorfismo (próxima aula)