```
//DUCK TYPING:

//O Duck Typing é um princípio do POO onde prezamos por simplificar a execução de um objeto que possuí as mesmas características que outro objeto - geralmente isso acontece muito entre objetos que possuem a mesma classe mãe.

//O princípio do Duck Type quer dizer o seguinte "se anda como pato, faz barulho como pato e tem a aparência de pato, então é um pato", trazendo isso para o contexto o POO, se objetos compartilham as mesmas características, por que não criar interfaces que compartilham essas características invés de gerar uma interface diferente para cada objeto? Se "anda como pato, então é pato".

//Vejamos 2 exemplos:
class Animal{ //Temos um exemplo típico, uma classe animal que irá ter suas características herdadas em outras classes...
    constructor(nome){
        this._nome = nome
    }

    falar(){}
}

class Aguia extends Animal{ //Perceba que teremos 3 classes filhas de Animal, cada uma terá um método á sua maneira...
    constructor(nome){
        super(nome)
    }

    falar(){
        return 'Ahah!'
    }
}

class Cachorro extends Animal{
    constructor(nome){
        super(nome)
    }

    falar(){
        return 'Auau!'
    }
```

```
}

class Cavalo extends Animal{
    constructor(nome){
        super(nome)
    }

    falar(){
        return 'Rrrrrrrrrr!'
    }
}

class VerificacaoSemDuckType { //Aqui temos um sistema de verificação que não utiliza Duck Type, fazendo comparações
    static fazerBarulho(nome){ //entre os tipos de todos os objetos separadamente, veja como o código fica grande...
        if(VerificacaoSemDuckType.existeFalarEmAguia(nome)){
            return nome.falar()
        }
        else if(VerificacaoSemDuckType.existeFalarEmCachorro(nome)){
            return nome.falar()
        }
        else if(VerificacaoSemDuckType.existeFalarEmCavalo(nome)){
            return nome.falar()
        }
    }

    static existeFalarEmAguia(nome){ //E precisamos também gerar uma função comparadora para cada objeto...
        return "falar" in nome && nome.falar instanceof Function
    }
    static existeFalarEmCachorro(nome){
        return "falar" in nome && nome.falar instanceof Function
    }
    static existeFalarEmCavalo(nome){
        return "falar" in nome && nome.falar instanceof Function
    }
}
```

```
class VerificacaoComDuckType { //Agora veja a diferença com o um sistema de verificação que usa Duck Type, ele se
    static fazerBarulho(nome){ //aproveita do fato de que todos os objetos têm o mesmo método e faz a verificação
        if(VerificacaoComDuckType.existeFalar(nome)){ //direto sobre o método, em vez de comparar objeto á objeto...
            return nome.falar()
        }
    }

    static existeFalar(nome){ //Uma única funcão de comparação precisou ser criada...
        return "falar" in nome && nome.falar instanceof Function
    }
}

let aguia = new Aguia('Águia') //Note que ambas as interfaces comparadoras funcionam, mas qual delas exigiu menos
let aguiaFalando = VerificacaoSemDuckType.fazerBarulho(aguia) //esforço para fazer?
console.log(aguiaFalando)
/*RESULTADO NO CONSOLE:
Ahah!
*/

let falandoAguia = VerificacaoComDuckType.fazerBarulho(aguia)
console.log(falandoAguia)
/*RESULTADO NO CONSOLE:
Ahah!
*/

let black = new Cachorro('Black')
let caoFalando = VerificacaoSemDuckType.fazerBarulho(black)
console.log(caoFalando)
/*RESULTADO NO CONSOLE:
Auau!
*/

let falandoCao = VerificacaoComDuckType.fazerBarulho(black)
console.log(falandoCao)
/*RESULTADO NO CONSOLE:
Auau!
```

```
*/

let spirit = new Cavalo('Spirit')
let cavaloFalando = VerificacaoSemDuckType.fazerBarulho(spirit)
console.log(cavaloFalando)
/*RESULTADO NO CONSOLE:
Rrrrrrrrrr!
*/

let falandoCavalo = VerificacaoComDuckType.fazerBarulho(spirit)
console.log(falandoCavalo)
/*RESULTADO NO CONSOLE:
Rrrrrrrrrr!
*/
```