SENAI-SP

Kotlin

Classes

Classes

Kotlin é uma linguagem orientada a objetos com classes e herança baseada em mixim.

Todo objeto é uma instância de uma classe e todas as classes defendem de Object.

A herança baseada em mixim significa que, embora toda classe, exceto Object, tenha exatamente uma superclasse, um corpo de classe pode ser reutilizado em várias hierarquia de classe.

```
class Point (
  val x: Double? = null, // Declara x, inicialmente nula.
  val y: Double? = null, // Declara y, inicialmente nula.
  val z: Double = 0.0 // Declara z, inicialmente 0.
)
```

Classes

Todas as variáveis de instância geram um método *getter* implícito.Variáveis de instância não *finais* também geram um método *setter* implícito.

```
class Point (
  val x: Double? = null,
  val y: Double? = null
)

fun main() {
  val point = Point()
  point.x = 4.0 // Usa o método setter de x.
  println(point.x == 4.0) // Usa o método getter de x.
  println(point.y == null) //Usa o método getter de y.
}
```

Classes

As variáveis e métodos de classe são identificadas por companion object envolvendo sua definição. A diferença entre variáveis e métodos de instância entre de classe e que a última pertence a classe e não é acessado com **this**.

```
import "kotlin.math.sqrt"

class Ponto (val x: Double, val y: Double) {
  companion object {
    fun distanciaEntre(a: Ponto, b: Ponto) : Double {
      val dx = a.x - b.x
      val dy = a.y - b.y
      return sqrt(dx * dx + dy * dy)
    }
  }
}
```

```
fun main() {
  var a = Ponto(2.0, 2.0)
  var b = Ponto(4.0, 4.0)
  // imprime 2.8284271247461903
  print(Ponto.distanciaEntre(a, b))
}
```

Classes

Um Objeto pode ser criado utilizando o Construtor de uma Classe.

O nome do construtor pode ser:

```
val p1 = Point(2.0, 2.0)
```

ou:

```
val p2 = new Point()
```

Classes

Nas classes onde os atributos forem definidos com "val" todos os objetos criados são constantes, assim os valores dos atributos nunca poderão ser alterados.

```
// Construtor de objetos que nunca mudam
class Ponto(val x: Double = 0.0, val y: Double = 0.0)

fun main() {
  val ponto1 = Ponto()
  val ponto2 = Ponto(1.0, 2.0)
  . . .
}
```

Classes

SENAI-SP

Se não for declarado qualquer construtor, um construtor padrão será providenciado automaticamente. O construtor padrão não contém qualquer argumento e invoca o mesmo tipo de construtor de sua superclass.

```
class Ponto {
  var x = 0.0
  var y = 0.0
}

fun main() {
  val ponto = Ponto(); // O construtor padrão
  . . .
}
```

Classes

SENAI-SP

Utilize o construtor nomeado para implementar múltiplas variações de construtores para uma classe, provendo funcionalidades extras.

```
class Ponto {
  var x: Double
  var y: Double

  init {  // Inicializador
    x = 0.0
    y = 0.0
  }

  constructor()  // Construtor Padrão

  constructor(x: Double, y: Double) {  // Construtor Alternativo
    this.x = x
    this.y = y
  }
}
```

Classes

No exemplo abaixo é demostrado a chamada de o construtor da subclasse.

```
open class Pessoa {
  var nome: String = ""
  constructor(nome: String) {
    this.nome = nome
  }
}

class Empregado : Pessoa {
  var matricula: String = ""
  constructor(nome: String, matricula: String) : super(nome) {
    this.matricula = matricula
  }
}
```

Classes

SENAI-SP

No exemplo abaixo é demostrado a chamada de o construtor da subclasse na forma simplificada.

```
open class Pessoa(val nome: String)

class Empregado(nome: String, val matricula: String) : Pessoa(nome)
```

Classes

Algumas vezes o único propósito do construtor é redirecionar a execução para outro construtor da mesma classe.

```
class Ponto(val x: Int, val y: Int) {
  // Redirecionamento para o outro construtor
  constructor(x: Int) : this(x, 0)
}
```

Classes

Utilize a palavra reservada **object** para implementar construtores que nem sempre retorna uma nova instância desta class.

```
// Implementação do Pattern Singleton
object ClienteDao {

  fun fazAlgo() {
    . . .
  }
}
```

Classes

SENAI-SP

Utilize a palavra reservada **abstract** para implementar classes abstratas que não podem ser utilizadas para criar objetos.

```
// Implementação de uma classe abstrata
abstract class Calculador {
  // Método abstrato, cuja implementação não existe
  abstract fun atualizaValor()
}

// Extendendo a classe abstrata Calculador
class MeuCalculo : Calculador() {
  override fun atualizaValor() {
    . . .
  }
}
```

Interfaces

As interfaces no Kotlin podem conter declarações de métodos abstratos, bem como implementações de métodos. O que os diferencia das classes abstratas é que as interfaces não podem armazenar estado.

```
interface Ordenador {
  val crescente: Boolean

  fun ordena()
}

class MeuOrdenador : Ordenador {
  override val crescente = true

  override fun ordena() {
    . . .
  }
}
```

Modificadores

SENAI-SP

Os modificadores de visibilidade são aplicados a classes, objetos, interfaces, construtores, funções, propriedades e seus setters, os getters sempre têm a mesma visibilidade que a propriedade. A visibilidade padrão, usada se não houver modificador explícito, é pública. São quatro modificadores de visibilidade no Kotlin:

private - visível somente na classe

protected - visível na classe e nas suas subclasses

internal - visível aos clientes deste módulo

public - visível a todos os clientes

Extensões

Extensão é a capacidade de estender uma classe com novas funcionalidades sem precisar herdar uma classe

```
fun String.prompt() : String = JOptionPane.showInputDialog(this)

fun String.show(vararg args: Any) =
  JOptionPane.showMessageDialog(null,
    args.fold(this) {acum, obj -> "$acum$obj"})

fun Int.isPar() = this % 2 == 0

fun main() {
  val nome = "Informe seu Nome".show()
  "Bem Vindo, $nome".show()
  if( 3.isPar() )
    . . .
}
```