DCC024 Linguagens de Programação

2022.1

Orientação a Objetos

Haniel Barbosa

Orientação a Objetos (OO)

Little bundles of **data** that know how to **do operations** on themselves.

- Objetos são valores de um tipo abstrato de dados
- Objetos se comunicam entre si através de suas operações
  - Chamadas de métodos vistas como *troca de mensagens*
- Uma forma comum de implementar objetos é via *instâncias* de *classes*

Principais elementos de programação OO

- Encapsulamento
  - Parte dos dados e operações de um objeto apenas não acessíveis ao objeto
  - Comum separar partes públicas e privadas do estado e operações

Principais elementos de programação OO

- Encapsulamento
  - Parte dos dados e operações de um objeto apenas não acessíveis ao objeto
  - Comum separar partes públicas e privadas do estado e operações
- Herança e subtipagem
  - Definição de um tipo abstrato a partir de outro
  - *Liskov's substitution principle*
    - Se $S <: T$, então objetos de tipo T podem ser substituídos por objetos de tipo S.

Principais elementos de programação OO

- Encapsulamento
  - Parte dos dados e operações de um objeto apenas não acessíveis ao objeto
  - Comum separar partes públicas e privadas do estado e operações
- Herança e subtipagem
  - Definição de um tipo abstrato a partir de outro
  - *Liskov's substitution principle*
    - Se $S <: T$, então objetos de tipo T podem ser substituídos por objetos de tipo S.
- Polimorfismo
  - Sobrecarga (*overloading*): Redefinição de operações *para* novos tipos
  - Sobrescrita (*overriding*): Redefinição de operações *em* novos tipos
  - *Method Dispatch*
    - Determinação de qual versão de uma operação invocar baseado nos parâmetros

Exemplo de conjunto com bitsets em Python

```
INT_BITS = 32

def getIndex(element):
  index = int(element / INT_BITS)
  offset = element % INT_BITS
  bit = 1 << offset
  return (index, bit)

class Set:
  def __init__(self, capacity):
    self.vector = [0] * (1 + int(capacity / INT_BITS))
    self.capacity = capacity

  def add(self, element):
    (index, bit) = getIndex(element)
    self.vector[index] |= bit

  def delete(self, element):
    (index, bit) = getIndex(element)
    self.vector[index] &= ~bit

  def contains(self, element):
    (index, bit) = getIndex(element)
    return (self.vector[index] & bit) > 0
```

Exemplo de conjunto com bitsets em Python

```
s = Set(60)
s.add(4)
s.contains(4)
s.delete(4)
s.contains(4)

s.vector[0] = 16
s.contains(4)
```

Exemplo de conjunto com bitsets em Python

```
s = Set(60)
s.add(4)
s.contains(4)
s.delete(4)
s.contains(4)

s.vector[0] = 16
s.contains(4)

s.add(70)

def errorAdd(self, element):
  if (element > self.capacity):
    raise IndexError(str(element) + " is out of range.")
  else:
    (index, bit) = getIndex(element)
    self.vector[index] |= bit
    print(element, "added successfully!")

Set.add = errorAdd
```

Exemplo de conjunto com bitsets em Python

```
s = Set(60)
s.add(4)
s.contains(4)
s.delete(4)
s.contains(4)

s.vector[0] = 16
s.contains(4)

s.add(70)

def errorAdd(self, element):
  if (element > self.capacity):
    raise IndexError(str(element) + " is out of range.")
  else:
    (index, bit) = getIndex(element)
    self.vector[index] |= bit
    print(element, "added successfully!")

Set.add = errorAdd

s = Set(0)
s.add(40)
s.add(70)
```

Comparação entre diferentes linguagens

	Encapsulamento	Extensão
C	Não	Não
SML	Sim	Não
Python	Não	Sim
C++/Java/...	Sim	Sim

Exemplo de conjunto com bitsets em Python

```
class ErrorSet(Set):
  def checkIndex(self, element):
    if (element > self.capacity):
      raise IndexError(str(element) + " is out of range.")

  def add(self, element):
    self.checkIndex(element)
    Set.add(self, element)
    print(element, "successfully added.")

  def delete(self, element):
    self.checkIndex(element)
    Set.delete(self, element)
    print(element, "successfully removed.")

  def contains(self, element):
    if element > self.capacity:
      return false
    else:
      return Set.contains(self, element)
```

Classe para visitação em Python

```
class Visitor:
  "A parameterized list visitor."
  def __init__(self, cb):
    self.cb = cb
  def __str__(self):
    return "Visitor with callback: {0}".format(self.cb)
  def visit(self, n):
    for i in range(0, len(n)):
      n[i] = self.cb.update(n[i])
    return n

class CallbackBase:
  "The basic callback"
  def __init__(self):
    self.f = lambda x: x+1
  def __str__(self):
    return "basic callback"
  def shouldUpdate(self, i): return True
  def update(self, i):
    return self.f(i) if self.shouldUpdate(i) else i

l = [0, 1, 2, 3]; cb = CallbackBase()
v = Visitor(cb)
v.visit(l)
```

Classe para visitação em Python

```
class CallbackEven(CallbackBase):
  def __str__(self):
    return "even callback"
  def shouldUpdate(self, i):
    return i % 2 == 0

v0 = Visitor(CallbackBase())
v1 = Visitor(CallbackEven())
v0.visit([0,1,2,3])
v1.visit([0,1,2,3])
```

Classe para visitação em Python

```
class CallbackEven(CallbackBase):
  def __str__(self):
    return "even callback"
  def shouldUpdate(self, i):
    return i % 2 == 0

v0 = Visitor(CallbackBase())
v1 = Visitor(CallbackEven())
v0.visit([0,1,2,3])
v1.visit([0,1,2,3])


class CallbackDouble(CallbackBase):
  def __init__(self):
    self.f = lambda x: 2*x
  def __str__(self):
    return "double callback"

v = Visitor(CallbackDouble())
v.visit([0,1,2,3])
```

Classe para visitação em Python

```
class CallbackComb(CallbackDouble, CallbackEven):
  def __str__(self):
    return "the combining callback"

v = Visitor(CallbackComb())
v.visit([0,1,2,3])
```

Classe para visitação em Python

```
class CallbackComb(CallbackDouble, CallbackEven):
  def __str__(self):
    return "the combining callback"

v = Visitor(CallbackComb())
v.visit([0,1,2,3])

class CallbackOtherComb(CallbackDouble, CallbackEven):
  def update(self, i):
    return self.f(self.f(i)) if self.shouldUpdate(i) else i

v = Visitor(CallbackOtherComb())
v.visit([0,1,2,3])
print(v)
```

Ligação estática e dinâmica em C++

```
class A
{
 public:
  void f() { std::cout << "A!\n"; }
};

class B
{
 public:
  void f() { std::cout << "B!\n"; }
};

class C : public A, public B
{
};

int main()
{
  C* pc = new C;
  pc->f();
}
```

Ligação estática e dinâmica em C++

```
class A
{
 public:
  void f() { std::cout << "A!\n"; }
};

class B
{
 public:
  void f() { std::cout << "B!\n"; }
};

class C : public A, public B
{
};

int main()
{
  C* pc = new C;
  pc->A::f();
  pc->B::f();
}
```

Ligação estática e dinâmica em C++

```
class A
{
 public:
  void f() { std::cout << "A!\n"; }
};

class B
{
 public:
  void f() { std::cout << "B!\n"; }
};

class C : public A, public B
{
  void f() { std::cout << "C!\n"; }
};

int main()
{
  C* pc = new C;
  pc->f();
}
```

Ligação estática e dinâmica em C++

```
void aux(A* p)
{
  p->f();
}

int main()
{
  C* pc = new C;
  pc->f();

  aux(pc);
}
```

Ligação estática e dinâmica em C++

```
class A
{
 public:
  virtual void vf() { std::cout << "A!\n"; }
};

class B
{
 public:
  virtual void vf() { std::cout << "B!\n"; }
};

class C : public A, public B
{
  void vf() override { std::cout << "C!\n"; }
};
```

```
void aux(A* p)
{
  p->vf();
}
```

```
int main()
{
  C* pc = new C;
  aux(pc);
}
```