

Estruturas de Linguagem

Programação Orientada a Objetos

Francisco Sant'Anna

francisco@ime.uerj.br

<http://github.com/fsantanna/EDL>

O Conceito de Abstração

An **abstraction** is a view or representation of an entity that includes only the most significant attributes.

- Classes de abstrações

- processos

```
sortInt(list, listLen)
```

- dados

```
float d;
```

- Uma “arma” contra complexidade

- facilita no gerenciamento de programas

Abstração de Dados (definida pelo programador)

- Tipos de dados definidos pelo programador
- Invólucro que inclui
 - representação do tipo de dado
 - implementação das operações sobre o tipo de dado
- Detalhes desnecessários são escondidos
 - “information hiding”
- Acesso
 - tipo abstrato + operações (e construtores)

Exemplo: Pilha

<code>create(stack)</code>	Creates and possibly initializes a stack object
<code>destroy(stack)</code>	Deallocates the storage for the stack
<code>empty(stack)</code>	A predicate (or Boolean) function that returns true if the specified stack is empty and false otherwise
<code>push(stack, element)</code>	Pushes the specified element on the specified stack
<code>pop(stack)</code>	Removes the top element from the specified stack
<code>top(stack)</code>	Returns a copy of the top element from the specified stack

```
. . .  
create(stk1);  
push(stk1, color1);  
push(stk1, color2);  
temp = top(stk1);  
. . .
```

```
Stack stk; /** Cre  
stk.push(42);  
stk.push(17);  
topOne = stk.top();  
stk.pop();  
. . .
```

- Vetor?
- Lista?

Considerações

- Como expor a interface sem expor a representação / implementação?
 - modificadores
 - private / protected
 - módulos
 - unidades de compilação
- Operações entre diferentes tipos abstratos?
 - “friend”
- Encapsulamento de nomes?
 - namespaces, packages, modules

Pilha em C++ e C

```
class Stack {  
    public:  
        Stack();  
        ~Stack();  
        int empty (void);  
        void push  (int);  
        void pop   (void);  
        int top    (void);  
    private:  
        int* stackPtr;  
        int  maxLen;  
        int  topSub;  
}
```

```
typedef struct Stack Stack;  
Stack* create (void);  
int     empty  (Stack*);  
void    push   (Stack*, int);  
void    pop    (Stack*);  
int     top    (Stack*);
```

define

“Fundamentos” de OO

- Encapsulamento
- Herança
- Despacho Dinâmico

Subtipagem

- Noção de “substituibilidade”
- Partes que operam sobre elementos de um “supertipo” (e.g. funções), podem também operar sobre elementos dos seus “subtipos”.
- Se S é subtipo de T , então S pode ser usado de maneira confiável em qualquer contexto em que T é usado
- Noção é *semântica*
 - não depende de suporte de linguagem
 - nem é imposta com suporte de linguagem

Subtipagem

- Subtipagem por *herança* vs *interface*

```
class Shape {  
    int x, y;  
    void draw (void);  
    void move (int dx, int dy) {  
        x+=dx; y+=dy;  
    }  
}
```

```
class Circle extends Shape {  
    void draw (void) {  
        ...  
    }  
}
```

```
Shape s = <...>;  
s.move(10,10);  
s.draw();
```

```
interface Shape {  
    int x, y;  
    void draw (void);  
    void move (int dx, int dy);  
}
```

```
class Circle implements Shape {  
    void draw (void) {  
        ...  
    }  
    void move (int dx, int dy) {  
        ...  
    }  
}
```

```
Shape s = <...>;  
s.move(10,10);  
s.draw();
```

Subtipagem

- Subtipagem por *herança* vs *interface*
 - *herança*: subtipagem de implementação
 - subtipo “herda” a implementação de métodos da superclasse
 - mistura subtipagem com reuso
 - complexo?
 - *interface*: subtipagem de interface
 - subtipo deve prover implementação completa
 - apenas subtipagem
 - repetitivo?

Subtipagem

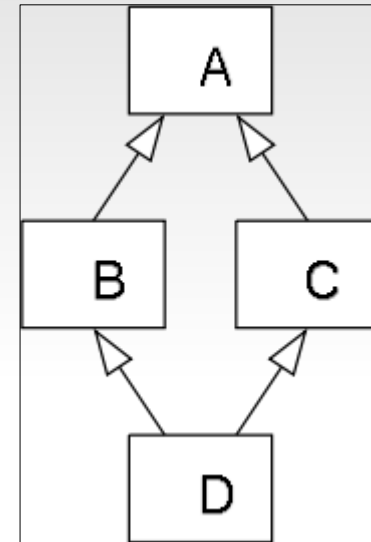
- Mixins, Traits, Extensions, Categories

```
interface Shape {  
    int x, y;  
    void draw (void);  
    void move (int dx, int dy);  
}  
  
mixin Moveable for Shape {  
    void move (int dx, int dy) {  
        x+=dx; y+=dy;  
    }  
}  
  
class Circle implements Shape  
    with Moveable {  
    void draw (void) {  
        ...  
    }  
}
```

- permite que interfaces tenham implementações padrão “por fora” da hierarquia de tipos
- também resolve o problema de herança múltipla

Heranças: problemas

- Herança múltipla
 - “the diamond problem”:



Heranças: problemas

- Herança vs Composição
 - *is a* vs *has a*

```
class Stack extends ArrayList
{
    private int stack_pointer = 0;
    public void push( Object article )
    {
        add( stack_pointer++, article );
    }
    public Object pop()
    {
        return remove( --stack_pointer );
    }
    public void push_many( Object[] articles )
    {
        for( int i = 0; i < articles.length; ++i )
            push( articles[i] );
    }
}
```

```
Stack a_stack = new Stack();
a_stack.push("1");
a_stack.push("2");
a_stack.clear();
```

Heranças: problemas

- Fragile Base Problem
 - solução: métodos sempre serem `final`

```
class Stack
{
    private int stack_pointer = -1;
    private Object[] stack = new Object[1000];
    public void push( Object article )
    {
        assert stack_pointer < stack.length;
        stack[ ++stack_pointer ] = article;
    }
    public Object pop()
    {
        assert stack_pointer >= 0;
        return stack[ stack_pointer-- ];
    }
    public void push_many( Object[] articles )
    {
        assert (stack_pointer + articles.length) < stack.length;
        System.arraycopy(articles, 0, stack, stack_pointer+1,
                           articles.length);
        stack_pointer += articles.length;
    }
}
```

```
class Monitorable_stack extends Stack
{
    private int high_water_mark = 0;
    private int current_size;
    public void push( Object article )
    {
        if( ++current_size > high_water_mark )
            high_water_mark = current_size;
        super.push(article);
    }

    public Object pop()
    {
        --current_size;
        return super.pop();
    }
    public int maximum_size_so_far()
    {
        return high_water_mark;
    }
}
```


Subtipagem

- Liskov Substitution Principle



Liskov Substitution Principle

Barbara Liskov wrote LSP in 1988:

What is wanted here is something like the substitution property: If for each object *o1* of type *T* there is an object *o2* of type *T'* such that *P* defined in terms of *T*, the behavior of *o1* is indistinguishable from the behavior of *o2*.

People are very confused below about what "behavior" means. Here is one reasonable definition (from the C++ FAQ, but the definition is language independent):

"derived class objects must be substitutable for the base class objects. That means objects of the derived class must behave in a manner consistent with the promises made in the base class' contract."

```
class Rectangle {  
    void stretch (float x, float y) {  
        <...>  
    }  
}  
  
class Square extends Rectangle {  
    void stretch (float x, float y) {  
        <??>  
    }  
}
```

Subtipagem

- Subclassing (linguagem) \neq Subtyping (semântica)
- Subclasse pode “herdar” Superclasse ou “implementar” Superclasse e não ser Subtipo

Subtipagem

- Rect é subtipo de Shape se, em qualquer contexto, substituir uma instância de Shape por uma instância de Rect mantém o comportamento do programa consistente.

```
interface Shape {  
    void draw (void);  
    void move (int dx, int dy);  
}  
class Rect implements Shape {  
    <...>  
    bool isSquare (void) { ... }  
}  
class Ellipse implements Shape {  
    <...>  
    bool isCircle (void) { ... }  
}
```

Variância

- Como uma relação entre tipos compostos A e B “varia” de acordo com relação entre tipos de suas subpartes?
- Co-variância
 - mantém relação normal de subtipagem
 - se as partes de A são subtipos das partes de B, então A é subtipo de B
- Contra-variância
 - inverte relação normal de subtipagem
 - se as partes de A são subtipos das partes de B, então B é subtipo de A
 - se as partes de B são supertipos das partes de A, então A é supertipo de B
- Invariância
 - quebra a relação de subtipagem

Variância

- Isso também vale para tipos compostos?
 - e.g., Rectangle[] é subtipo de Shape[] ?

```
void f_ss (Shape[] shapes) {  
    <...>  // usa como shapes  
}  
Rect[] rects = <...>;  
f_ss(rects);  
print(rects[10].isSquare());
```

```
void draw_ss (Shape[] shapes) {  
    foreach s in shapes {  
        s.draw();  
    }  
}  
Rect[] rects = <...>;  
draw_ss(rects);  
print(rects[10].isSquare());
```

```
void draw_ss (Shape[] shapes) {  
    shapes[10] = new Ellipse();  
    foreach s in shapes {  
        s.draw();  
    }  
}  
Rect[] rects = <...>;  
draw_ss(rects);  
print(rects[10].isSquare());
```

Variância

- Como uma relação entre tipos compostos A e B “varia” de acordo com relação entre tipos de suas subpartes?
- Co-variância
 - mantém relação normal de subtipagem
 - se as partes de A são subtipos das partes de B, então A é subtipo de B
- Contra-variância
 - inverte relação normal de subtipagem
 - se as partes de A são subtipos das partes de B, então A é supertipo de B
- Invariância
 - quebra a relação de subtipagem

Variância

- Arrays?
 - *invariante!*
 - read only?
 - *co-variante!*
 - write only?
 - *contra-variante!*
- Em Java, arrays são co-variantes
 - “roubadinha”

Rect é subtipo de **Shape**

e
void draw_all (**Rect**[] rects)
é subtipo de
void draw_all (**Shape**[] shapes);

Rect é subtipo de **Shape**

e
void ins_rect (**Rect**[] rs, Rect r)
é supertipo de
void ins_shape (**Shape**[] ss, Shape s)

```
void draw_ss (Shape[] shapes) {  
    foreach s in shapes {  
        s.draw();  
    }  
}  
Rect[] rects = <...>;  
draw_ss(rects);  
print(rects[10].isSquare());
```

```
void ins_rect (Rect[] rs, Rect r) {  
    rs.append(r);  
}  
Shapes[] ss = <...>;  
ins_rect(ss, new Rect());  
ss[10].draw();
```


Variância

- Funções?

- retornos?

- *co-variante!*

- parâmetros?

- *contra-variante!*

```
Rect f (Rect) { ... }  
Rect r1 = Rect();  
Rect r2 = f(r1);
```

```
Shape f (Rect) { ... }
```

```
Quad f (Rect) { ... }
```

- Os mesmos princípios devem ser aplicados ao herdar uma classe modificando seus métodos.

```
Rect f (Rect) { ... }  
Rect r1 = Rect();  
Rect r2 = f(r1);
```

```
Rect f (Shape) { ... }
```

```
Rect f (Quad) { ... }
```


Classes vs Protótipos

- Classe
 - reuso por template (modelo) **abstrato**
 - criação por construtores
- Protótipo
 - reuso por objeto **concreto**
 - criação por clonagem ou delegação

```
class Rect {  
    <...>  
}  
r1 = new Rect(...);  
r2 = new Rect(...);  
print(Rect.x, r1.x, r2.x)
```

```
Rect = {  
    <...>  
}  
r1 = clone(Rect);  
r2 = clone(r1);  
print(Rect.x, r1.x, r2.x)
```

Protótipos

- Implementação
 - Clonagem
 - novo objeto é cópia completa do seu protótipo
 - Delegação
 - novo objeto é inicialmente “vazio” e delega seu comportamento padrão ao seu protótipo

Protótipos em Lua

- Objetos com campos
- Clonagem
- Objetos com métodos
- Delegação
- “Classes” por delegação
- “Herança” por delegação
- Polimorfismo
- Encapsulamento?
- *Static typing* vs *Duck typing*

Dispatcho Dinâmico

- Amarração entre o nome do método e a implementação ocorre somente em tempo de execução.
- Tipo de *polimorfismo* dinâmico
 - no contexto de herança, também é chamado de *overriding*
 - não confundir com *overloading*, que é estático

```
Shape s = <...>;  
s.move(10,10);  
s.draw();
```

```
Shape shapes[];  
<...>  
for s in shapes do  
    s.draw();  
end
```