Modelação e Padrões de Desenho

Capítulo 1 – Basic Patterns

Fernando Miguel Carvalho
DEETC
Instituto Superior de Engenharia de Lisboa,
Centro de Cálculo
mcarvalho@cc.isel.ipl.pt

Polimorfismo e Dynamic Binding

Override – Redefinição de métodos

As subclasses podem redefinir (*overriding*) os métodos de instância (não estáticos), dando outra implementação.

A redefinição tem que ter:

- A mesma assinatura (nome e parâmetros);
- O mesmo tipo de retorno;
- Uma lista igual ou mais restrita de excepções;

Override != Overload

A **redefinição** (herança) e a **sobrecarga** (na mesma classe) são mecanismos totalmente distintos.

Notas:

- A resolução da chamada a **métodos em sobrecarga** é feita na **compilação**;
- A resolução da chamada a **métodos redefinidos** é feita em ***runtime*** (***dinamic binding***)
- Redefinir com mesma assinatura e tipo de retorno diferentes gera **erro de compilação**;
- Redefinir métodos estáticos ou campos nas subclasses apenas **esconde** (***hiding***) a definição da superclasse.

Override – Redefinição de métodos...

Protecção contra redefinição:

- Os métodos classificados com **`final`** não podem ser redefinidos nas subclasses;
- As chamadas são optimizadas pela JVM.

Chamar o método da superclasse:

- Na subclasse a implementação de um método na superclasse pode ser invocada com **`super.m().`**

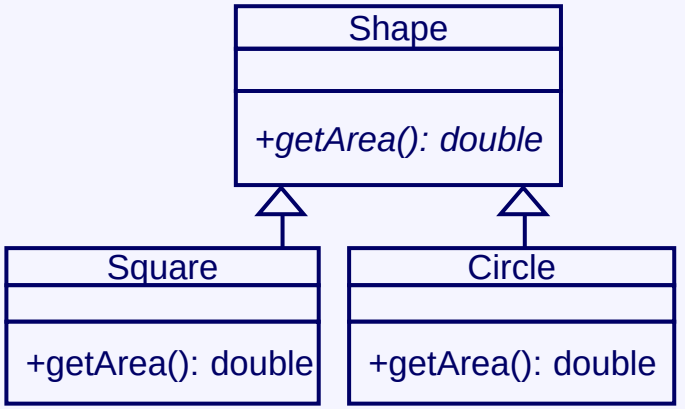
Restrição na herança:

- Se na subclasse não faz sentido a funcionalidade de determinado método, então deve ser redefinido para lançar excepção (ex: **`MethodNotSupported`**).

Chamada polimórfica

As chamadas aos métodos de instância (virtuais) são decididas em tempo de execução, dependendo do tipo de objecto referenciado (***dynamic binding***)

```
double totalArea(Shape[] sa) {
  double total = 0.0;
  for(int i=0; i < sa.lenght ; ++i)
    total += sa[i].getArea();
}
```



A chamada polimórfica **`ref.m()`** é processada assim:

Passo 1: **`cl`** ← classe do objecto referenciado por **`ref`**

Passo 2: Se **`m()`** implementado por **`cl`**
- ➔**então** chamar implementação de **`m()`** em **`cl`**
- ➔**senão** **`cl`** ← superclasse de **`cl`**, e repetir passo 2

Object

A classe **`Object`** define um conjunto de métodos dos quais se destacam:

- **`boolean equals(Object obj)`**
  - Permite saber se um objecto é semanticamente equivalente a outro.
- **`String toString()`**
  - Devolve uma representação do objecto em forma de *string.*
- **`int hashCode()`**
  - Serve de suporte à utilização de tabelas de *hash*.

Demo

Demo: Implementação da hierarquia Shapes (Shape.java, Square.java e Circle.java):

- Criação de classes bases de uma hierarquia (Shape);
- Criação de classes derivadas, como extensão (especialização) à classe base.

Interfaces


Now Playing
6 of 15
Feel Good, Inc.
Gorillaz
Demon Days
3:41
-1:20
MENU

Interfaces…

- Representam contratos (promessas) de serviços a disponibilizar por quem as implementa.
- Permitem definir:
  - Métodos de instância;
  - Campos de tipo constantes.
- Todos o membros têm acessibilidade pública
- Todos os métodos são abstractos

```
[ClassModifiers] interface InterfaceName
        [extendsInterface1, Interface2 ...]  {
   [InterfaceMemberDeclaration]
}
```

```
public interface Student {
  double calculateAverage();
}
```

```
public interface Employee {
  double calculateSalary();
}
```

```
public class WorkerStudent
   implements Student,
   implements Employee
{
  public double calculateAverage()
  { ... }
    public double calculateSalary()
  { ... }
}
```

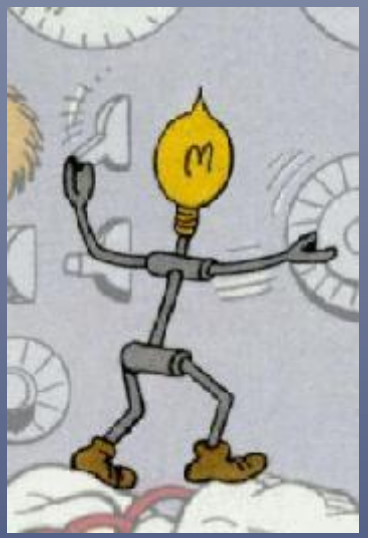
Interfaces e Subtipos

Também são definidas relações de subtipo

- Quando uma classe implementa uma interface
- Quando uma interface estende outra interface

Condição	Subtipo	Supertipo
`C1` extends `C2`	`C1`	`C2`
`I1` extends `I2`	`I1`	`I2`
`C` implements `I`	`C`	`I`
qualquer interface `I`	`I`	`Object`
qualquer tipo `T`	`T[]`	`Object`
`T1` subtipo de `T2`	`T1[]`	`T2[]`

Basic Patterns



Basic Patterns

Nome	Descrição
Interface **- Prestador de serviços** **- Serviço comum**	Pode ser usado no desenho de um conjunto de **classes prestadoras de serviços**, que forneçam um **serviço em comum**, de forma a que um objecto cliente possa usar qualquer uma das classes prestadoras de serviços sem que seja necessário alterar código do lado cliente.
Parent Abstract Class - ***framework***	Útil no desenho de uma ***framework*** com uma implementação consistente de funcionalidades comuns a um conjunto de classes relacionadas.
Private Methods - ***internal behavior***	Uma forma de desenhar numa classe **comportamento para uso interno**, não acessível do exterior.
Acessor Methods - ***object state*** - ***getter (getXXX ou isXXXX)*** ***- setter (setXXX)***	Uma forma de aceder ao **estado de um objecto** através de métodos específicos, garantindo encapsulamento e impedindo os clientes de manipularem directamente os campos de um objecto.
Constant Data Manager	Útil para desenhar um simples repositório centralizado de dados constantes da aplicação.
ImmutableObject	Usado para garantir que o estado de um objecto não pode ser alterado. Pode ser usado para evitar que o acesso concorrente a um objecto resulte em condições de *race*.
Monitor	Uma forma de desenhar uma aplicação de maneira a não produzir resultados imprevisíveis, quando mais que uma *thread* acede ao mesmo objecto.