Herança: classes base e derivadas
Redefinição e sobreposição de métodos

Composição e herança

Java permite reutilizar código através de construção de classes novas a partir de classes existentes.

As duas maneiras de reutilização de classes são: composição e herança.

Na composição a classe nova inclui objetos de outras classes. A relação respetiva pode ser definida com a palavra "tem" (*has-a*).

Exemplo: uma viatura tem rodas; uma pessoa tem nome.

Na herança a classe nova extende a funcionalidade de outra classe. A relação respetiva pode ser definida com a palavra "é" <uma espécie de> (*is-a*).

Exemplo: um sedan é uma viatura; um aluno é uma pessoa.

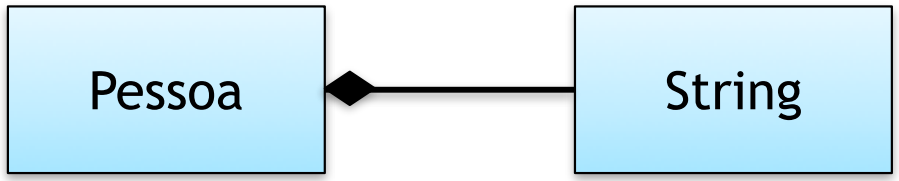
Composição

Na composição simplesmente criamos objetos de outras classes dentro da classe nova.

Deste modo conseguimos reutilizar funcionalidades asseguradas por outras classes.

Exemplo:

```
public class Pessoa
{
  private String nome; // nome da pessoa - composição
  private int idade;
  ...
}
```



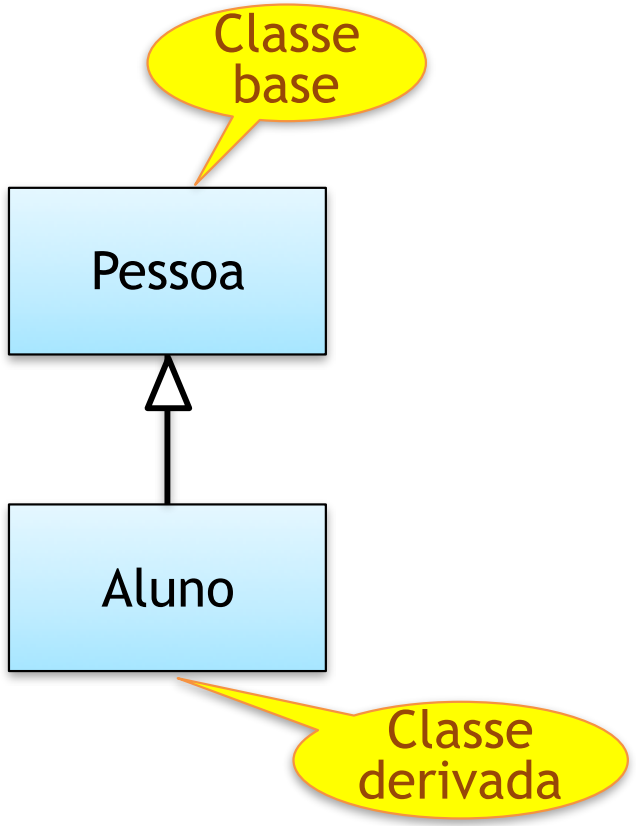
Herança

Herança permite criar um tipo novo que é uma espécie da classe existente.

Herdamos a interface da classe existente e todas as suas funcionalidades e adicionamos código novo para acrescentar novas funcionalidades.

Exemplo:

```
public class Aluno extends Pessoa
{
  private int nMec;
  ...
}
```



Herança - conceitos

A classe base é frequentemente chamada super-classe (*super class*) ou classe-mãe (*parent class*).

A classe derivada é frequentemente chamada subclasse (sub-class) ou classe-filha *(child class).*

Uma classe pode servir de base para qualquer quantidade de outras classes.

Entretanto uma classe só pode derivar de uma única classe (i.e. herança múltipla não é permitida).

Quais as relações entre:

```
Mamífero, Rato, Gato
Peugeot 307, Pneu, Volante, Roda, Jante, Viatura
```

Controlo de acesso

A classe derivada tem acesso a todos os membros da classe base que não sejam privados.

O especificador de acesso protected indica que um dado membro pode ser acedido por classe derivada ou por outras classes que estejam no mesmo **package** mas não pode ser acedido por outras classes.

É aconselhável manter os membros-dados privados e, caso classes derivadas precisam de acesso a estes membros, garantí-lo com métodos protegidos.

Herança de métodos

Ao herdar métodos podemos:

- mantê-los inalterados;
- acrescentar-lhes funcionalidades novas ou
- redefini-los (*override*)

Herança de métodos - herdar

```
class Pessoa
{
    private String nome;
    private int idade;

    public Pessoa(String nome, int age)
    { this.nome = nome; idade = age; }

    public String getName() { return nome; }

}
class Aluno extends Pessoa {

    private int nMec;

    public Aluno(String nome, int age, int num)
    //implementação do construtor

    public static void main(String[] args) {
            Aluno al = new Aluno("Paulo Ferreira", 20, 77888);
            System.out.println(al.getName());
    }
}
```

Herança de métodos - redefinir

A classe derivada pode modificar a implementação de métodos da classe base, i.e. redefení-los (*override*).

```
class Pessoa
{
    private String nome;
    private int idade;

    public Pessoa(String nome, int age)
    { this.nome = nome; idade = age; }

    public String getName() { return nome; }
    public String toString() { return "Pessoa"; }

}
class Aluno extends Pessoa {

    private int nMec;

    public Aluno(String nome, int age, int num)
    //implementação do construtor

    public String toString() { return "Aluno";}

    public static void main(String[] args) {
            Aluno al = new Aluno("Paulo Ferreira", 20, 77888);
            System.out.println(al);
    }
}
```

Herança de métodos - estender

Para se poder invocar a versão base do mesmo método, utiliza-se a palavra-chave super (que refere a super-classe).

Exemplo:

```
class Pessoa {
  private String nome;
  private int idade;

  public Pessoa(String nome, int age)
  { this.nome = nome; idade = age; }
  public String toString() { return "Pessoa"; }
}

public class Aluno extends Pessoa {
  private int nMec;

  public Aluno(String nome, int age, int num)
  //implementação do construtor

  public String toString()
  { return super.toString() + " " + nMec; }
}
```

Herança de métodos e controlo de acesso

Métodos declarados como **public** na classe base também devem ser **public** em todas as subclasses.

Métodos declarados como **protected** na classe base devem ser **protected** ou **public** nas subclasses. Não podem ser **private**.

Métodos declarados sem controlo de acesso (**package**) podem manter ou ser **private** em subclasses.

Métodos declarados como **private** não são herdados pelo que não se aplicam as regras de visibilidade em subclasses.

Inicialização da classe base

Quando se cria um objeto da classe derivada, este contém um subobjeto da classe base.

É importante assegurar inicialização correta deste subobjeto.

A inicialização é feita em construtores da classe derivada que invocam automaticamente o construtor por omissão da classe base.

Caso pretenda invocar algum outro construtor da classe base (não o por omissão) deve chamá-lo explicitamente com a palavra-chave super e fornecer argumentos necessários.

A invocação explícita do construtor da classe base deve ocorrer obigatoriamente na primeira linha do construtor da classe derivada.

Exemplo de inicialização

```
public class Pessoa
{
    private String nome;
    private int idade;

    public Pessoa(String nome, int age)
    { this.nome = nome; idade = age; }
    ...
}

public class Aluno extends Pessoa {
    private int nMec = 57891;

    public Aluno(String nome, int age, int num)
    { super(nome, age); nMec = num; }
    ...

    public static void main(String[] args) {
            Aluno al = new Aluno("Paulo Ferreira", 20, 77888);
            System.out.println(al);
    }
}
```

O que acontece se eliminarmos a invocação explícita do construtor da classe Pessoa?

Ordem de execução de construtores

```
// o construtor de cada classe imprime o seu nome
public class Turma {
    private int size;
    private Aluno[] students;
    private Pessoa prof;

    public Turma (int s, Pessoa p) {
            System.out.println("A criar uma turma");
            students = new Aluno[size = s];
            prof = p;
    }

    public static void main(String[] args) {
            Pessoa p = new Pessoa("Arnaldo Martins", 50);
            Turma t = new Turma(5, p);
            t.students[0] = new Aluno
                                ("Ana Pereira", 19, 76543);
    }
}
```

```
A criar uma pessoa
A criar uma turma
A criar uma pessoa
A criar um aluno
```

Destruição/finalização

Se for necessária a finalização, o código deve ser colocado num bloco **try** {...} **finally** {...}. O bloco **finally** deve invocar os seus métodos de finalização adequados.

Estes métodos na classe derivada devem, na última linha, invocar o método de finalização respetivo da classe base.

Exemplo:

```
public class TankBattery extends Tank {
   public TankBattery(double cap) { super(cap); }

   public void cleanup() { //do appropriate cleanup here
       System.out.println("Charge battery");
       super.cleanup();
   }

   public static void main(String[] args) {
       TankBattery tb = new TankBattery(60);
       try {
           tb.spend(20);
           System.out.println(tb);
           tb.fill(5.6);
           System.out.println(tb);
       }
       finally { tb.cleanup(); }
   }
}
```

@Override

A anotação @Override pode ser usada para indicar ao compilador que a seguir se pretende redefinir (*override)* um método da classe base. Se, por engano, tentar sobrecarregar em vez de redefinir, o compilador vai reportar o erro.

Qial é a diferença entre "redefinir" e "sobrecarregar"?

A redefinição indica que um método existente na classe base deve ser implementado de maneira diferente na classe derivada. Mas continua a ter exatamente a mesma assinatura.

A sobrecarga (sobreposição) indica que se pretende adicionar mais um método na classe derivada que terá o mesmo nome que o método da classe base mas recebe a lista de argumentos diferente.

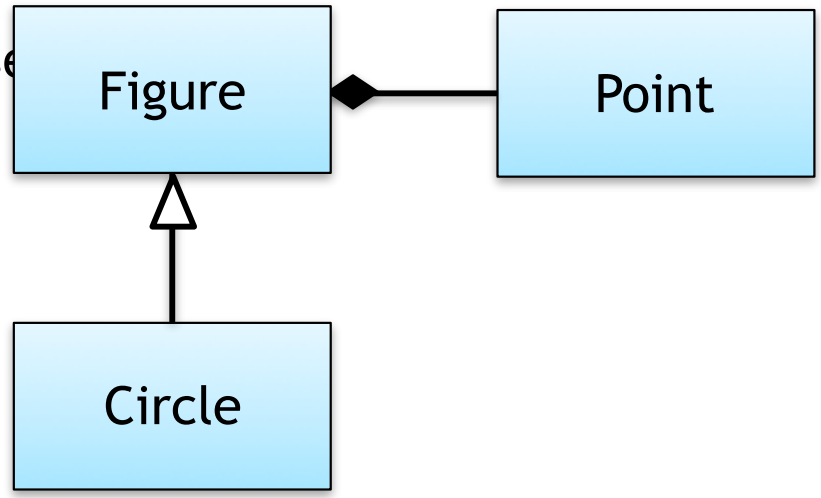
Exemplo com @Override

Exemplo:

```
class Point {
   double x, y;
   public Point(double x, double y){ this.x = x; this.y = y; }
}

class Figure {
   private String color;
   private Point center;
   public Figure (String co, Point ce) { color = co; center = ce; }
   public void move() { System.out.println("Move a figure"); }
}

class Circle extends Figure{
   private double radius
   public Circle(String co, Point ce, double r) { super(co, ce); radius = r; }

   @Override
   public void move() { System.out.println("Move a circle"); }

   @Override //erro
   public void move(Point newPos)
   { System.out.println("Move a circle to a new position"); }
}

public class Drawing {
   public static void main(String[] args) {
       Point p = new Point(10.0, 10.0);
       Circle c = new Circle("amarelo", p, 2.2);
       c.move();
       c.move(new Point(0.0, 0.0));
   }

}
```

Upcasting

O processo de conversão duma referência da classe derivada para uma referência da classe base chama-se ***upcasting***.



Esta operação é sempre segura porque convertemos do tipo mais específico para o tipo menos específico.

Exemplo:

```
Figure f = new Circle("amarelo", new Point(1.0, 1.0), 2.2);
```

Palavra-chave final

A palavra-chave final pode ser aplicada a dados, métodos e classes.

Quando aplicada a dados serve para definir constantes que recebem os seus valores ou na altura de compilação de código *(compile-time constants*) ou em *run-time*.

Constantes definidas na altura de compilação devem ser dum tipo primitivo.

Quando final é aplicada a uma referência indica que a referência será constante mas o objeto para o qual ela aponta – não!

Um membro-dado pode ser declarado com final e static => haverá apenas uma instância deste dado na memória que não poderá ser alterada.

Exemplo de membros-dados constantes

Exemplo:

```
class Value {
   int i;
   public Value(int i) { this.i = i; }
}

public class FinalData {
   // Compile-time constants:
   private final int valueOne = 10;
   private static final int VALUE_TWO = 99;

   // Cannot be compile-time constants:
   private final int valueThree = (int) (Math.random() * 10);
   private static final Value OBJ = new Value(33);

   // Arrays:
   private final int[] a = { 1, 2, 3, 4, 5, 6 };

   public FinalData() { val = 4; }


   public static void main(String[] args) {
        FinalData fd = new FinalData();

        fd.valueOne++; // erro
        FinalData.VALUE_TWO++; // erro
        fd.valueThree++; // erro

        fd.OBJ = new Value(55); // erro
        fd.OBJ.i++; // objeto não é constante

        for(int i = 0; i < fd.a.length; i++)
             fd.a[i]++; // objeto não é constante
        fd.a = new int[3]; // erro
   }
}
```

Finais brancos

Finais brancos são atributos declarados como constantes que não receberam valores iniciais.

Finais brancos devem ser inicializados antes de usados.

Inicialização de dados finais deve ser realizada ou no momento de definição ou em cada construtor!

Exemplo:

```
public class FinalData {

  private final int val;

  public FinalData()
  { val = 4; }
  ...
}
```

Argumentos finais

Os argumentos de funções também podem ser declarados como finais, proibindo deste modo que sejam alterados.

Exemplo:

```
class Gizmo {
  public void spin() {}
}

public class FinalArguments {
  void with(final Gizmo g) {
     g = new Gizmo(); // Illegal -- g is final
  }
  void without(Gizmo g) {
     g = new Gizmo(); // OK -- g not final
     g.spin() ;
  }

  void f(final int i) { i++; } // Can't change

  int g(final int i) { return i + 1; }

  public static void main(String[] args) {
     FinalArguments bf = new FinalArguments();
     bf.without(null);
     bf.with(null);
  }
}
```

Métodos finais

final pode ser aplicado a métodos para indicar que não podem ser redefinidos em classes derivadas.

Os métodos privados por omissão são **final** porque de qualquer modo é impossível redefini-los já que estão escondidos da classe derivada.

Exemplo:

```
class WithFinals {
   private void f() { System.out.println("WithFinals.f()"); }
   public final void g() { System.out.println("WithFinals.g()"); }
}

class OverridingPrivate extends WithFinals {
   public void f() { System.out.println("OverridingPrivate.f()") ; }
   public void g() { System.out.println("OverridingPrivate.g()"); } //erro
}

public class FinalOverriding {
   public static void main(String[] args) {
       OverridingPrivate op1 = new OverridingPrivate() ;
       op1.f();
       op1.g();
       WithFinals op2 = op1;
       op2.f(); //erro
       op2.g();
   }
}
```

```
OverridingPrivate.f()
WithFinals.g()
WithFinals.g()
```

Classes finais

Uma classe pode ser declarada como final.

Isto significa que será proibido derivar desta classe.

Todos os métodos numa classe final são final por omissão.

Exemplo:

```
class SmallBrain {}

final class Dinosaur {
   int i = 7;
   int j = 1;
   SmallBrain x = new SmallBrain();
   void f() {}
}

class Further extends Dinosaur {} // erro

public class Jurasic {
   public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
}
```

Ordem de inicialização

```
class Insect {
  private static int x1 = printInit("static Insect.x1 initialized");

  private int i = 9;
  protected int j;
  Insect() { System.out.println("i = " + i + " , j = " + j ); j = 39; }

  static int printInit(String s) { System.out.println(s); return 47; }
}

public class Beetle extends Insect {
  private int k = printInit("Beetle.k initialized");
  private static int x2 = printInit("static Beetle.x2 initialized");
  public Beetle() {
      System.out.println("k = " + k) ;
      System.out.println("j = " + j);
  }

  public static void main(String[] args) {
      System.out.println("Beetle constructor");
      Beetle b = new Beetle();
  }
}
```

```
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9 , j = 0
Beetle.k initialized
k = 47
j = 39
```

Métodos comuns a todas as classes

Todas as classe em Java derivam da super-classe **java.lang.Object**.

Logo herdam os métodos seguintes definidos na classe **Object**:

- toString()
- equals()
- hashCode()
- finalize()
- clone()
- getClass()
- wait()
- notify()
- notifyAll()

final

toString()

O método **toString**() deve ser sempre redefinido para ter um comportamento de acordo com o objeto.

Exemplo:

```
Pessoa p = new Pessoa();
p.setAge(17);
p.setName("Paulo Ferreira");
System.out.println(p); //Pessoa@55f96302
```

☹

```
@Override
public String toString(){
  return nome + " " + idade;
}
```

```
System.out.println(p); //Paulo Ferreira 17
```

☺

equals()

A expressão c1 == c2 verifica se as referências c1 e c2 apontam para a mesmo objeto. Caso c1 e c2 sejam variáveis automáticas a expressão anterior compara valores.

O métodos equals() testa se dois objetos são iguais. Por omissão, verifica se as referências apontam para o mesmo objeto.

```
Ponto p1 = new Ponto(1, 1);
Ponto p2 = new Ponto(1, 1);
System.out.println(p1 == p2);              // false
System.out.println(p1.equals(p2)); // false
```

equals() deve ser redefinido para que os objetos possam ser comparados.

Regras de redefinição de equals()

Para o argumento nulo (**null**) o método deve devolver **false**.

Devem ser respeitadas propriedades seguintes:

- reflexiva: `x.equals(x) == true`
- simétrica: `x.equals(y) == y.equals(x)`
- transitiva: `((x.equals(y) == true) && (y.equals(z) == true)) == (x.equals(z) == true)`
- consistente: não modificar os objetos comparados

Deve-se redefenir e não sobrecarregar!

Sempre que redefinir o método equals deve-se também redefinir o método hashCode.

Redefinição de equals()

```
@Override
public boolean equals(Object obj)
{
    if (obj == null)
            return false;
    if (this == obj)
            return true;
    if (getClass() != obj.getClass())
            return false;
    Circle other = (Circle) obj;
    if (centro == null) {
            if (other.centro != null)
                    return false;
    }
    else if (!centro.equals(other.centro))
            return false;
    if (raio != other.raio)
            return false;
    if (cor == null) {
            if (other.cor != null)
                    return false;
    } else if (!cor.equals(other.cor))
            return false;
    return true;
}
```

O argumento deve ser obrigatoriamente uma referência para Object

Indica a classe do objeto obj

```
public class Circle {
    private String cor;
    private Point centro;
    private double raio;
    …
}
```

equals e herança

```
class BaseClass {
    public BaseClass(int i ) { x = i; }
    private int x;

    @Override
    public boolean equals(Object obj) {
       if (this == obj) return true;
       if (obj == null) return false;
       if (getClass() != obj.getClass()) return false;
       if (x != ((BaseClass) obj).x) return false;
       return true;
    }
}

class DerivedClass extends BaseClass {
    public DerivedClass(int i, int j ) { super(i); y = j; }
    private int y;
    public boolean equals(Object rhs) {
    // Não é necessário testar a classe. Feito em base
        return super.equals(rhs) && y == ((DerivedClass) rhs).y;  }
    static public void main(String[] argv)
    {
       BaseClass b = new BaseClass(2);
       DerivedClass d = new DerivedClass(2,3);
       System.out.println(b.equals(b));
       System.out.println(d.equals(d));
       System.out.println(b.equals(d));
       System.out.println(d.equals(b));
    }
}
```

```
true
true
false
false
```

hashCode

Sempre que redefinir o método equals deve-se também redefinir o método hashCode.

O objectivo do *hash* é ajudar a identificar qualquer objeto através de um número inteiro. É usado em *hash tables*.

As regras de implementação são:

- Durante a execução duma aplicação, para o mesmo objeto, hashCode deve devolver o mesmo valor.
- Se dois objetos são iguais de acordo com o método equals, então hashCode deve produzir o mesmo inteiro para estes objetos.
- Se dois objetos não são iguais de acordo com o método equals, então hashCode não é obrigado de produzir inteiros diferentes para estes objetos (pode influenciar o desempenho de tabelas de *hashing*).
- Para computar o código *hash* deve-se utilizar os mesmos atributos que são comparados em equals.

Implementação de hashCode

A construção de uma boa função de *hash* não é trivial.

Exemplo:

```
@Override
public int hashCode()
{ return raio * centro.x() * centro.y();    }

Circulo c1 = new Circulo(10,15,27);
Circulo c2 = new Circulo(15,10,27);
```

```
4050
4050
```

Implementação de hashCode

O Eclipse pode gerar o método hashCode, bem como o método equals através de opções Source -> Generate hashCode() and equals() ...

Exemplo:

```
@Override
public int hashCode() {
  final int prime = 31;
  int result = 1;
  result = prime * result +
        ((centro == null) ? 0 : centro.hashCode());
  result = prime * result +
        ((cor == null) ? 0 : cor.hashCode());
  long temp;
  temp = Double.doubleToLongBits(raio);
  result = prime * result + (int) (temp ^ (temp >>> 32));
  return result;
}
```

Bibliografia

Bruce Eckel, Thinking in Java, 4th edition, Prentice-Hall, 2006

=> Capítulos “Access Control”, **“Reusing Classes”**