Herança

Construtores e Sobrecargas

Sobrecarga de métodos

- Dois métodos com o mesmo nome mas com tipo ou número de parâmetros diferentes, dizem-se sobrecarregados

```
public class Point {
  // ...

  /** Calcula a distância entre o ponto 'other' e 'this' */
  public double distance(Point other) {
    double dx = this.x - other.x;
    double dy = this.y - other.y;
    return Math.sqrt(dx * dx + dy * dy);
  }

 /** calculate the distance between this point and (x,y)*/
  public double distance(double x, double y) {
    double dx = this.x - x;
    double dy = this.y - y;
    return Math.sqrt(dx * dx + dy * dy);
  }
```

Sobrecarga de métodos: Exemplo

```
public class StringBuffer {
  StringBuffer append(String str) { ... }
  StringBuffer append(boolean b) { ... }
  StringBuffer append(char c) { ... }
  StringBuffer append(int i) { ... }
  StringBuffer append(long l) { ... }
  StringBuffer append(float f) { ... }
  StringBuffer append(double d) { ... }
  // ...
  public String substring(int i, int j) {
    // base method:
    return substring [i .. j-1]
  }
  public String substring(int i) {
    // provide default argument
    return substring(i, length - 1);
} }
```

Tipos de herança

- A linguagem Java suporta dois tipos de herança
  - de implementação
    - **`extends`**
  - de interface
    - **`implements`**

[*ClassModifiers*] class *ClassName*
[extends *SuperClass*]
[implements $Interface_1$, $Interface_2$ ...] {
[*ClassMemberDeclaration*]
}

Mais sobre herança

- As classes derivadas (que "estendem" outra classe) são também designadas de subclasses
- As classes base são também designadas *super classe*
- Se a classe E estende a classe B, então qualquer objecto de E pode ser usado como um objecto de B
- Não é permitida herança múltipla de implementação
- Todos os membros públicos e protegidos de uma super classe são acessíveis às suas classes derivadas
- Todos os membros protegidos são também acessíveis dentro do mesmo *package*

`Object`: Base da hierarquia

- A relação de herança entre classes forma uma hierarquia com a classe **`Object`** como base
- Todas as classes, exceto **`Object`**, têm uma única classe base
- Se nenhuma for especificada a classe **`Object`** é assumida como classe base

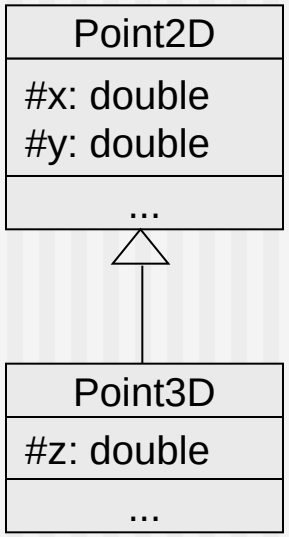
```
public class A extends Object {
  // ...
}
```

eq.

```
public class A {
  // ...
}
```

Construtores

- A iniciação de uma classe derivada consiste em:
  - Iniciar os campos da classe base
  - Iniciar os campos declarados na classe derivada



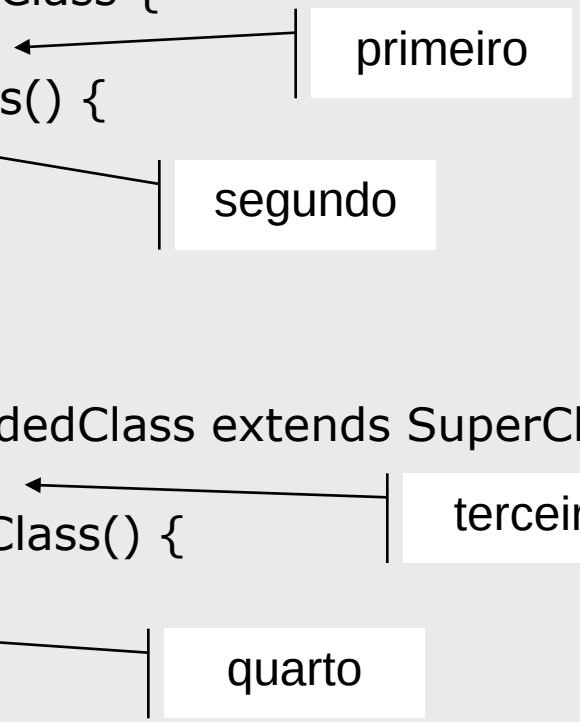

```
public class Point3D extends Point2D {
  protected double z;

  public Point3D(double x, double y, double z) {
    super(x, y);
    this.z = z;
  }

  public Point3D() {
    z = 0;
  }
}
```

Se usada, a instrução super tem de ser a primeira do construtor

Primeiro é chamado o construtor sem parâmetros da *superclass*

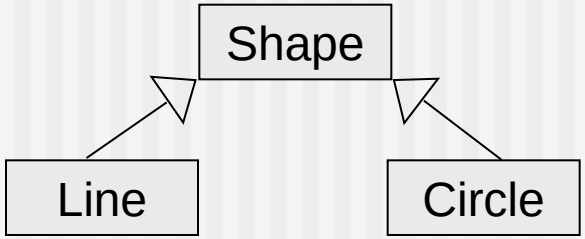
Construtores

```
public class SuperClass {
  int x = ...;
  public SuperClass() {
    x = ...;
  }
  //...
}

public class ExtendedClass extends SuperClass {
  int y = ...;
  public ExtendedClass() {
    super();
    y = ...;
  }
  // ...
}
```



Subtipos

- Uma subclasse *estende* as capacidades da sua superclass
- Uma subclasse é uma especialização da sua classe base
  - Todas as linhas são formas geométricas mas nem todas as formas geométricas são linhas



Conversões

- *Upcast* e *Downcast*
  - A promoção é feita implicitamente ao contrário da despromoção

Afectação Polimórfica

```
Circle c = new Circle();
Shape s = c;            // conversão implícita
Circle c1 = s;          // erro de compilação
Circle c2 = (Circle) s; // conversão explícita
```

- Operador **`instanceof`**
  - Testa se o tipo do objecto referenciado pelo primeiro argumento é igual ou convertível no segundo argumento

```
if (s intanceof Circle) {
  Circle c = (Circle) s;
  // ...
} else {  /* .. */ }
```

Conversões: Exemplo

```
class Student { ... }
class Undergraduate extends Student { ... }
class Graduate extends Student { ... }

Student student1, student2;
student1 = new Undergraduate(); // ok
student2 = new Graduate();      // ok

Graduate student3;
student3 = student2; // compilation error

student3 = (Graduate) student2; // explicit cast, ok

student3 = (Graduate) student1; // compilation ok
                                //run-time exception
```

Conversões: Exemplo (cont.)

- Aproximação pessimista

```
if (student instanceof Graduate) {
  Graduate graStudent = (Graduate) student1;
} else {
  // student1 não é um 'Graduate'
}
```

- Aproximação otimista

```
try {
  //...
  Graduate graStudent = (Graduate) student1;
  //...
} catch(ClassCastException e) {
  // student1 não é um 'Graduate'
}
```

Conversões: Porquê *downcast* ?

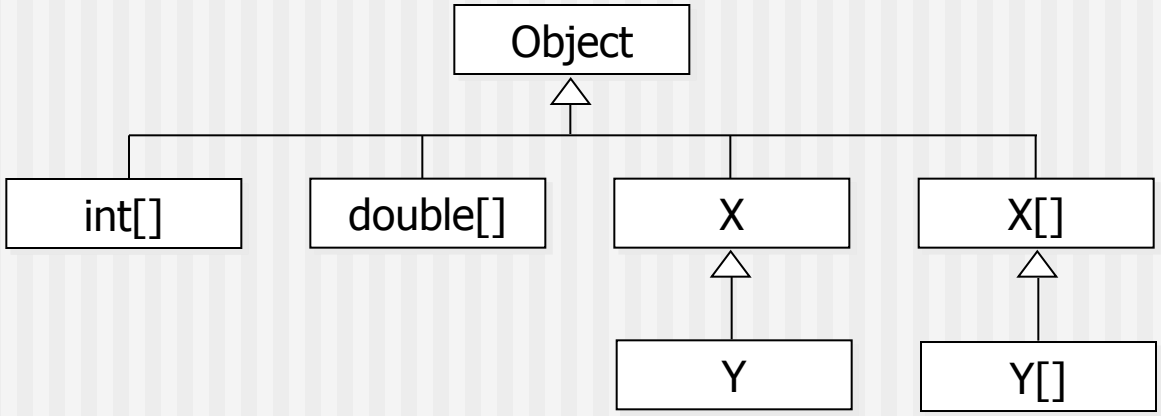
```
class Graduate extends Student {
  //...
  // Este método não existe em 'Student'
  public void getResearhTopic() {
  }
}
```

```
Student student1 = new Graduate();
//...
A.m1(student1);
```

```
class A {
  static public void m1(Student s) {
    if (student instanceof Graduate) {
      Graduate grad = (Graduate) student;
      grad.getResearchTopic(); // Ok
      //...
    }
} }
```

Arrays revisitados

- Os *arrays* são objectos e por isso o seu tipo é um subtipo de **`Object`**
- Se Y é um subtipo de X então Y[] também é um subtipo de X[]

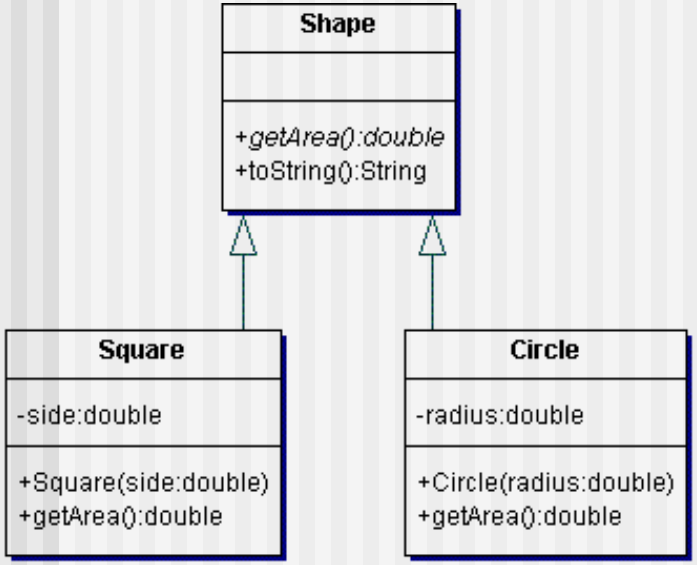


Arrays revisitados

```
Student sa1[];
Graduate sa2[] = new Graduate[40];
//...
sa1 = sa2; // afectação polimórfica
Student student4 = sa1[0];
Student student5 = sa2[0];

Graduate student6 = sa1[0]; // erro de compilação
Graduate student6 = (Graduate) sa1[0]; // Ok.
```

Redefinição de métodos

- Em Java todos os métodos de instância são *polimórficos*




```
public class ShapeApp {
  //...
  public void someMethod(Shape s) {
    //...
    double a = s.getArea();
  }
}
```

- O método **`getArea`** a ser chamado depende do tipo do objecto referenciado

Redefinição de métodos

- É possível a chamada a métodos redefinidos
- A palavra chave `super` permite a chamada dos métodos (originais) da classe base

```
public class Point2D {
  //...
  public String toString() {
    return "x=" + x + "; y=" + y;
  }
}
```

```
public class Point3D extends Point2D {
  //...
  public String toString() {
    return super.toString() + "; z=" + z;
  }
}
```

Sobrecarga, redefinição, esconder

- **Sobrecarga** (*Overloading*)
  - Mais do que um método com a mesma assinatura
- **Redefinição** (*Overriding*)
  - Substituir a implementação de um método da classe base por um do subtipo
- **Esconder** (*Hiding*)
  - Campos e métodos estáticos não podem ser redefinidos, apenas escondidos
  - Campos e métodos estáticos escondidos podem ser acecidos através da referência para a classe base (`super`)
  - Um método estático só pode ser escondido por outro método estático
  - Um campo estático pode ser escondido por um de instância

`Object`: revisitado

- 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*

Interfaces

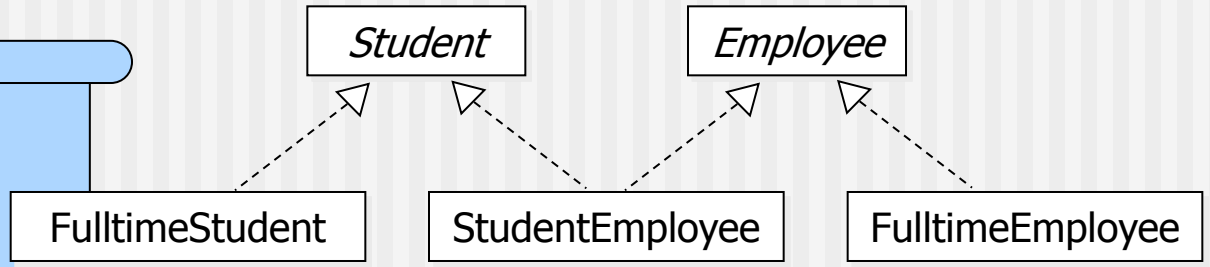
- Representam contratos (promessas) de serviços a dispon
  implementa.
- Permitem definir:
  - Métodos de instâr
  - Campos de tipo c
- Todos o membros
- Todos os métodos

```
[ClassModifie
        [extendsI
   [InterfaceM
}
```

```
public interface MyInterface {
  // método abstracto
  void aMethod(int i);
}

public class MyClass implements MyInterface
{
  public void aMethod(int i) {
    // implementação
  }
  //...
}
```

Interfaces

- Uma classe pode implementar várias interfaces
  - Podendo por isso assumir vários papéis em diferentes contextos




```
public interface Student
{
   float getAverage();
   // outros métodos
}

public interface Employee
{
  float getSalary();
  // outros métodos
}
```

```
Employee[] students = new Employee[...]
students[0] = new FulltimeEmployee();
Students[1] = new StudentEmployee();
//...
for (int i=0; i<students.length; ++i) {
  ... students[i].getSalary() ...
}
```

Interfaces: colisões de nomes

- O que acontece quando uma classe implementa várias interfaces e, por exemplo, dois métodos dessas interfaces têm o mesmo nome?
  - Se não têm a mesma assinatura, considera-se uma sobrecarga
  - Se têm a mesma assinatura e o mesmo tipo de retorno são considerados como um só método (fusão).
  - Se têm a mesma assinatura e o tipo de retorno é diferente, será gerado um erro de compilação
  - Se têm a mesma assinatura e o mesmo tipo de retorno mas lançam excepções de tipos diferentes, são considerados o mesmo método e o `throws` da implementação é a união das duas listas de `throws`

Interfaces: colisões de nomes

```
public interface X {
  public void method1(int i);
  public void method2(int i);
  public void method3(int i);
  public void method4(int i) throws Exception1;
}

public interface Y {
  public void method1(double i);
  public void method2(int i);
  public int method3(int i);
  public void method4(int i) throws Exception2;
}

public class MyClass implements X, Y {
  public void method1(int i) { ... } // redefine method1 de X
  public void method1(double i) { ... } // redefine method1 de Y
  public void method2(int i) { ... } // redefine method2 de X e Y
  public void method4(int i) // redefine method4 de X e Y
    throws Exception1, Exception2
    { ... }
}
```

Herança Múltipla *vs* Simples

- A linguagem C++ suporta totalmente herança múltipla
- A linguagem Java suporta apenas uma forma limitada deste tipo de herança
  - Através da extensão e implementação de interfaces
- No exemplo anterior o método **`getAverage`** pode ter uma implementação diferente caso se trate de um **`FulltimeStudent`** ou de um **`StudentEmployee`**
  - ... a mesma situação com o `getSalary`
- Com o tipo de herança múltipla que existe em C++, para além de uma instância poder desempenhar diferentes papeis, pode também herdar implementações de várias classes

Herança Múltipla *vs* Simples

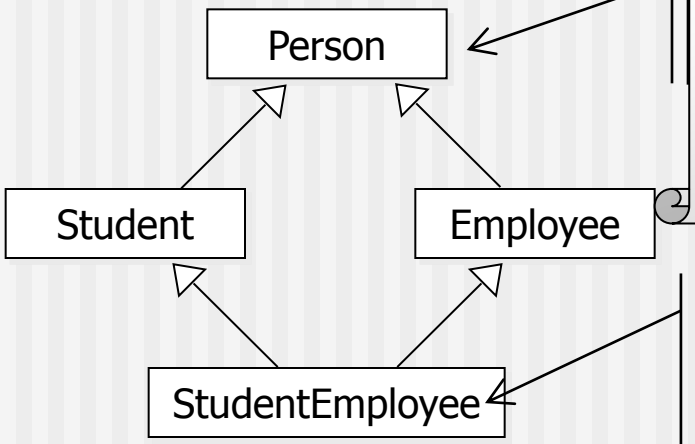
- Em C++ poderíamos escrever o seguinte código:

```
public class Student {
  public float getAverage() { ... }
  protected float average;
}

public class Employee {
  public float getSalary() { ... }
  protected float salary;
}

public class FulltimeStudent extends Student {
  // é herdada a implementação de getAverage
  // ... outros membros
}

public class FulltimeEmployee extends Employee {
  // é herdada a implementação de getSalary
  // ... outros membros
}
```

```
// Ilegal em JAVA
public class StudentEmployee
             extends Student,
                     Employee
{
  // implementação de
  // getAverage e getSalary
  // é herdada
}
```

Herança Múltipla *vs* Simples

- O modelo de herança múltipla da linguagem C++ é mais difícil de implementar, menos eficiente e pode introduzir ambiguidades nas relações
- Consideremos uma classe base de estudante e de empregado:

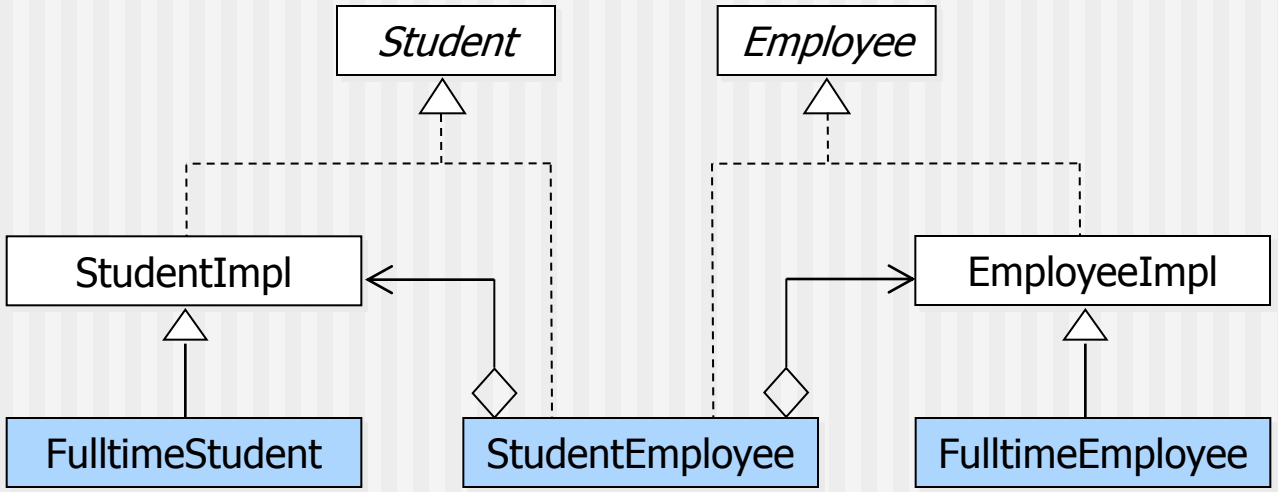



```
public class Person {
  public String getName() { ... }
  protected String name;
}
```

Quantos nomes?
Quando acesso name qual é que é usado?
Posso acessar às duas cópias?

Herança Múltipla *vs* Simples

- Uma forma de herança múltipla pode ser conseguida através de *delegação*



- Um objeto do tipo StudentEmployee delega as suas ações nos métodos getAverage e getSalary de StudentImpl e EmployeeImpl, respectivamente.

Herança Múltipla *vs* Simples

```
public class StudentEmployee implements Student, Employee
{
  protected StudentImpl studentImpl;
  protected EmployeeImpl employeeImpl;

  public StudentEmployee() {
    studentImpl = new StudentImpl();
    employeeImpl = new EmployeeImpl();
  }
  public float getAverage() {
    return studentImpl.getAverage(); // delegação
  }
  public float getSalary() {
    return employeeImpl.getSalary(); // delegação
  }
}
```