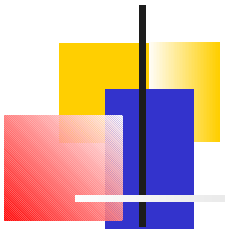


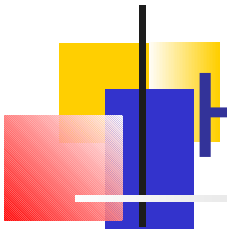
Aula

Herança

Renata Pontin de Mattos Fortes

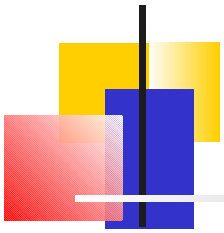
renata@icmc.usp.br

2006



Herança

- técnica de projeto OO fundamental, usada para criar e organizar classes reutilizáveis



Roteiro

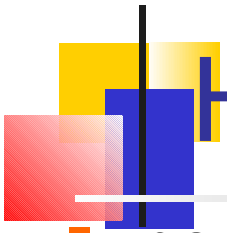
Criando Subclasses

Sobrescrevendo Métodos

Hierarquias de classes

Herança e Visibilidade

Projetando por Herança

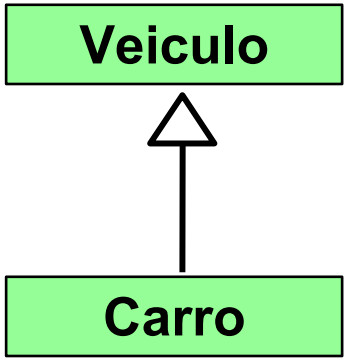


Herança

- *possibilita que um desenvolvedor derive uma nova classe a partir de uma existente*
- a classe existente é chamada classe PAI, ou *superclasse*, ou classe *base*
- a classe derivada é chamada classe FILHA ou subclasse
- como o nome sugere, a classe filha HERDA características da classe pai
- ou seja, herda os métodos e dados definidos pela classe pai.

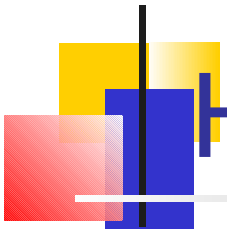
Herança

- os relacionamentos de herança são mostrados no diagrama de classes UML usando uma seta sólida com uma ponta triangular vazia, apontando para a classe pai



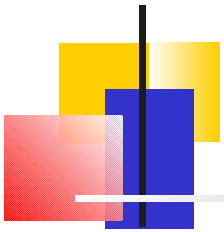
a propriedade de herança cria um relacionamento *is-a* (é um), significando que o filho é uma versão mais específica do pai.

a classe derivada pode acrescentar dados membros adicionais, MAS não pode remover dados membros



Herança

- um programador pode adaptar uma classe derivada conforme necessário, adicionando novas variáveis ou métodos, ou modificando os herdados
- *o **reuso de software é um benefício*** fundamental da **herança**
- usando componentes de software existentes para criar os novos, obtemos as vantagens sobre todos os esforços realizados no projeto, implementação, e testes do software existente



Derivando Subclasses

- em Java, usamos a palavra reservada **extends** para estabelecer o relacionamento de herança

```
class Carro extends Veiculo
{
    // conteudo da classe
}
```

Derivação

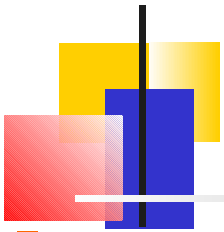
```
class Person {
    int age = 37;
    int getAge( ) { return age; } // usa Person::age
}
class OldPerson extends Person {
    int age = 99;
    int setAge( ) { age = 50; } // usa OldPerson::age

    public static void main( String [] args ) {
        OldPerson p = new OldPerson( );
        p.setAge( );
        System.out.println( p.getAge( ) );
    }
}
```

```
class Person  {
   public Person( String n, int ag, String ad, String p )    {
      name = n; age = ag; address = ad; phone = p;
   }
   public String toString( )    {
      return getName( ) + " " + getAge( ) + " " + getPhoneNumber( );
   }
   public final String getName( )    {
       return name;
   }
   public final int getAge( )    {
       return age;
   }
   public final String getAddress( )    {
       return address;
   }
   public final String getPhoneNumber( )    {
        return phone;
   }
   public final void setAddress( String newAddress )    {
        address = newAddress;
   }
   public final void setPhoneNumber( String newPhone )    {
        phone = newPhone;
   }
   private String name;
   private int age;
   private String address;
   private String phone;
}
```

```
 class Student extends Person{
    public Student( String n, int ag, String ad, String p, double nro )    {
         super( n, ag, ad, p );
         Nota = nro;
   }
   public String toString( )    {
         return super.toString( ) + " " + getNota();
   }
   public double getGPA( )    {
       return nota;
   }
   private double nota;
 }

class Employee extends Person{
   public Employee( String n, int ag, String ad, String p, double s )   {
         super( n, ag, ad, p );
         salary = s;
   }
   public String toString( )    {
         return super.toString( ) + " $" + getSalary( );
   }
   public double getSalary( )    {
         return salary;
   }
   public void raise( double percentRaise )    {
       salary *= ( 1 + percentRaise );
  }
  private double salary;
}
```

```
class PersonDemo{
    public static void printAll( Person[ ] arr )    {
         for( int i = 0; i < arr.length; i++ ) {
             if( arr[ i ] != null )  {
                 System.out.print( "[" + i + "] " + arr[ i ] );
                 System.out.println( );
             }
          }
    }
    public static void main( String [ ] args )    {
        Person [ ] p = new Person[ 4 ];
        p[0] = new Person( "Jose", 25, "Sao Carlos", "777-1212" );
        p[1] = new Student( "Beto", 27, "Sao Paulo", "666-1212", 8.0 );
        p[3] = new Employee( "Rui", 29, "Barueri", "555-1212", 10000.0 );
        if( p[3] instanceof Employee )
          ((Employee) p[3]).raise( .04 );
        printAll( p );
    }
}
```

polimorfismo



modificador protected

- os modificadores de visibilidade afetam o modo que os membros da classe podem ser usados numa classe filha.
- as variáveis e os métodos declarados com visibilidade private não podem ser referenciados pelo nome numa classe filha
- podem ser referenciados na classe filha se forem declarados com visibilidade public – mas as variáveis púplicas violam o princípio de encapsulamento
- existe um terceiro modificador de visibilidade que auxilia nas situações de herança: `protected`

modificador protected

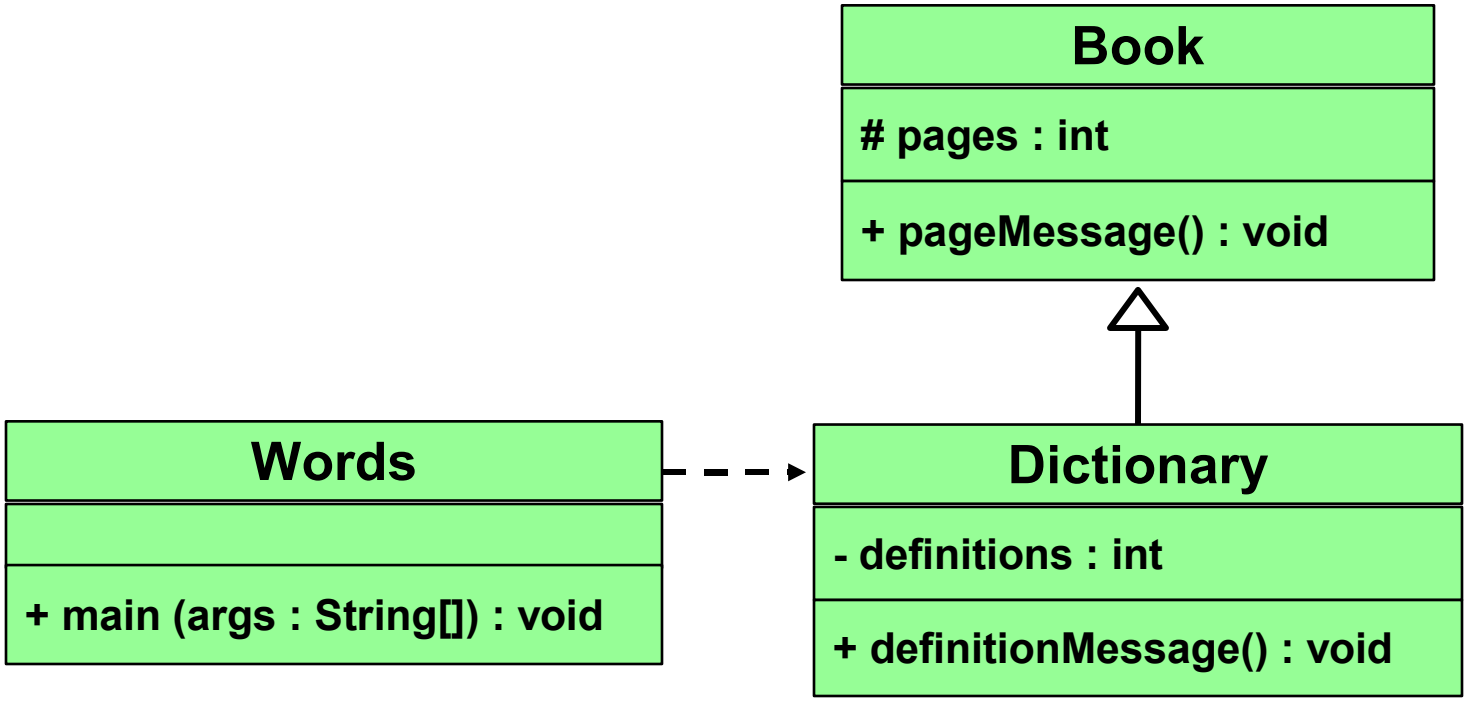
- o modificador **`protected`** permite que uma classe filha referencie uma variável ou método da classe pai, diretamente na classe filha

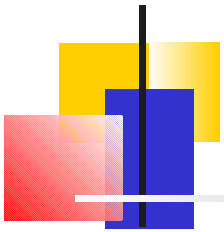
- oferece mais encapsulamento do que a visibilidade public, mas não é tão impermeável como a visibilidade private

- uma variável protected é visível por qualquer classe no mesmo package da classe pai

- as variáveis e métodos **`protected`** podem ser mostrados nos diagramas UML com o simbolo **#**

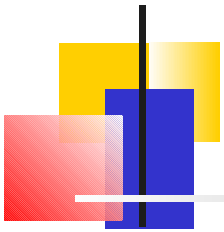
Diagrama de classes para Words





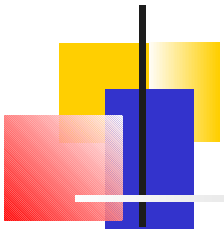
a referência super

- os construtores não são herdados, mesmo tendo visibilidade public
- a referência **`super`** pode ser usada para referenciar a classe pai, e geralmente é usada para invocar o construtor do pai



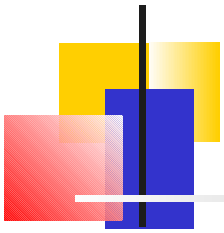
a referência super

- um construtor de filho é responsável por chamar o construtor do pai
- a primeira linha de um construtor do filho deve usar a referência **`super`** para chamar o construtor do pai
- a referência **`super`** também pode ser usada para referenciar outras variáveis e métodos definidos na classe pai



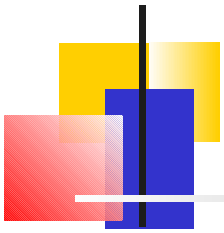
Herança Múltipla

- Java suporta herança simples, significando que uma classe derivada pode ter somente uma classe pai
- *a herança múltipla permite que uma* classe seja derivada de duas ou mais classes, herdando os membros de todos os pais
- as colisões, como o mesmo nome de variáveis em dois pais, têm que ser resolvidas
- Java não suporta herança múltipla
- ma maioria dos casos, o uso de interfaces nos dá aspectos de herança múltipla sem esse problema (overhead)



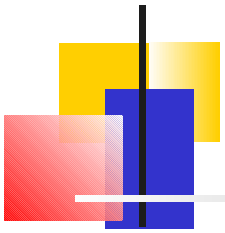
sobrescrevendo métodos

- uma classe filha pode sobrescrever *a* ***definição de um método herdado*** *em favor de si própria*
- o novo método deve ter a mesma assinatura que o método pai, mas pode ter um corpo diferente
- o tipo do objeto executando o método determina qual versão de método é invocado



Sobrescrita

- um método na classe pai pode ser invocado explicitamente usando a referência **`super`**
- se um método for declarado com o modificador **`final`**, ele não pode ser sobrescrito
- o conceito de sobrescrita pode ser aplicado para dados e é chamado de variáveis sombra
- variáveis sombras devem ser evitadas pois tendem a causar código confuso desnecessariamente



Métodos de Classes

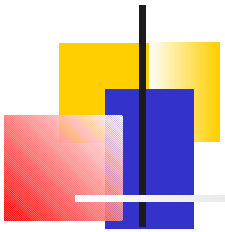
- a definição de um método em uma classe é realizada no corpo da classe como um bloco na forma:

```
[modificador] tipo nomeDoMetodo(argumentos) {
      corpo do metodo
}
```

- *nomeDoMetodo* – deve ser identificador válido

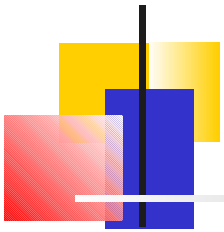
  O nomeDoMetodo e o tipo dos argumentos constituem a **assinatura** do método

- *tipo* indica o tipo de retorno do método (primitivos da linguagem, um tipo de objeto - nome de classe ou interface, ou void (por *default*).
- *modificador* (opcional).



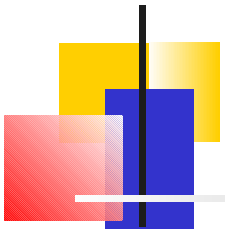
Método abstrato

- definição de um método = assinatura e implementação (o seu corpo).
- Algumas vezes, é possível afirmar que uma classe deve ter um método com determinada especificação mas nada pode se afirmar sobre seu comportamento. Nesses casos, define-se a classe com método abstrato.
- A classe que tenha pelo menos um método abstrato não pode ser instanciada e também deve ser declarada como abstrata.
- A definição desse método deverá ser completada em uma classe derivada dessa que contém o método abstrato, usando o mecanismo de redefinição de métodos.



Redefinição de métodos

- redefinição de métodos em classes derivadas
- sobrescrita ou *overriding*
- ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição (ou seja, um novo corpo) em uma classe derivada.
- O mecanismo de redefinição, juntamente com o conceito de *late binding* são essenciais para **polimorfismo.**



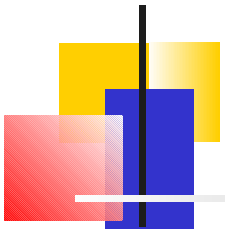
Late binding - ligação tardia

- o método a ser invocado definido durante a compilação do programa -> o mecanismo de **ligação prematura** *(early binding)* é utilizado.

Para o **polimorfismo**, a linguagem de POO deve suportar o *late binding (dynamic binding* ou *run-time binding)*, onde a definição do método que será efetivamente invocado só ocorre durante a execução do programa.

Em Java, todas as determinações de métodos a executar ocorrem através de *late binding* exceto em dois casos:

- métodos declarados como **final** não podem ser redefinidos e portanto não são passíveis de chamada polimórfica da parte de seus descendentes; e
- métodos declarados como **private** são implicitamente finais.



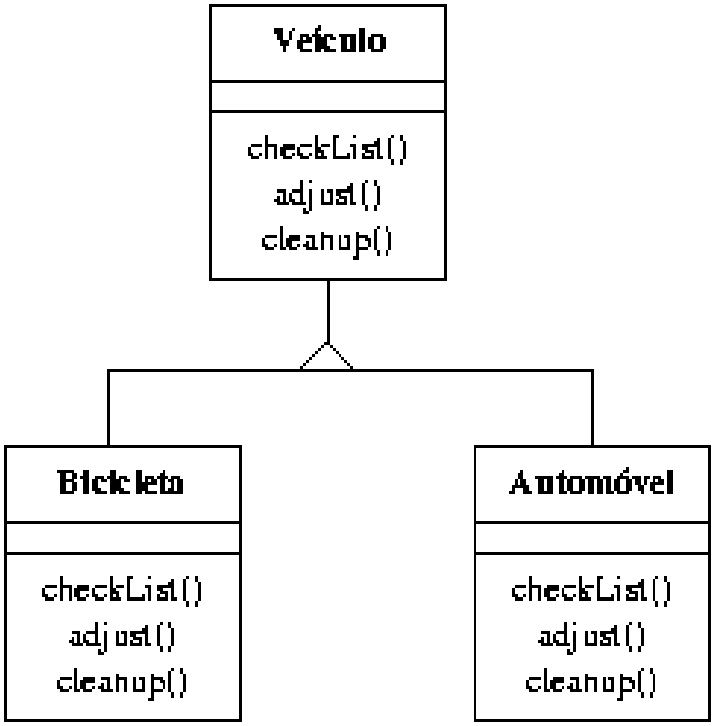
Polimorfismo

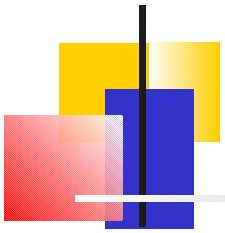
- princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação (assinatura) mas comportamentos distintos, especializados para cada classe derivada, usando para tanto uma referência a um objeto do tipo da superclasse.

- A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de *late binding*

- No polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, sendo utilizado o mecanismo de redefinição de métodos.

Polimorfismo em Java

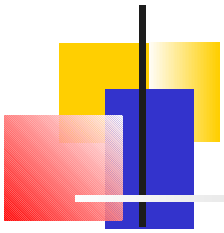




Polimorfismo em Java

As classes têm três métodos, definidos para veículos de forma geral e redefinidos mais especificamente para automóveis e bicicletas:

- `checkList( )`, para verificar o que precisa ser analisado no veículo;
- `adjust( )`, para realizar os reparos e a manutenção necessária; e
- `cleanup( )`, para realizar procedimentos de limpeza do veículo.



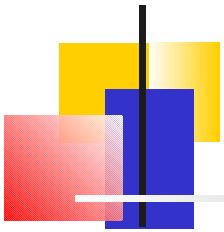
Polimorfismo em Java

A aplicação Oficina define um objeto que recebe objetos da classe Veículo. Para cada veículo recebido, a oficina executa na seqüência os três métodos da classe Veículo.

No entanto, não há como saber no momento da programação se a Oficina estará recebendo um automóvel ou uma bicicleta -- assim, o momento de decisão sobre qual método será aplicado só ocorrerá durante a execução do programa.

```java
import java.util.*;
class Veiculo {
    public Veiculo() {
System.out.print("Veiculo ");
    }
    public void checkList() {
System.out.println("Veiculo.checkList");
    }
    public void adjust() {
System.out.println("Veiculo.adjust");
    }
    public void cleanup() {
System.out.println("Veiculo.cleanup");
    }
}

class Automovel extends Veiculo {
    public Automovel() {
System.out.println("Automovel");
    }
    public void checkList() {
System.out.println("Automovel.checkList");
    }
    public void adjust() {
System.out.println("Automovel.adjust");
    }
    public void cleanup() {
System.out.println("Automovel.cleanup");
    }
}

class Bicicleta extends Veiculo {
    public Bicicleta() {
System.out.println("Bicicleta");
    }
    public void checkList() {
System.out.println("Bicicleta.checkList");
```

```java
public class Oficina {
    Random r = new Random();

    public Veiculo proximo() {
      Veiculo v;
      int code = r.nextInt();
      if (code%2 == 0)
          v = new Automovel();
      else
          v = new Bicicleta();

      return v;
    }

    public void manter(Veiculo v) {
      v.checkList();
      v.adjust();
      v.cleanup();
    }

    public static void main(String[] args) {
     Oficina o = new Oficina();
     Veiculo v;

     for (int i=0; i<4; ++i) {
        v = o.proximo();
        o.manter(v);
      }
    }
}
```



Sobrecarga

- O mecanismo de **redefinição** não deve ser confundido com o mecanismo de sobrecarga de métodos.
- um método aplicado a um objeto é selecionado para execução através da sua assinatura e da verificação a qual classe o objeto pertence.
- mecanismo de sobrecarga *(overloading)* - dois métodos de uma mesma classe podem ter o mesmo nome, desde que suas listas de parâmetros sejam diferentes, constituindo assim uma assinatura diferente.
- essa situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos tipos de argumentos do método.
- Exemplo de sobrecarga em Java - os métodos abs(), max() e min() da classe Math, que têm implementações alternativas para quatro tipos de argumentos distintos.

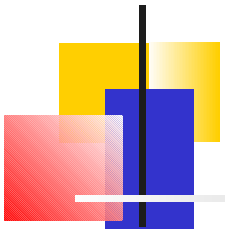
Polimorfismo em construtores

A invocação de métodos com *late binding* possibilita chamar construtores cujo comportamento poderia ser diferenciado polimorficamente.

Exemplo

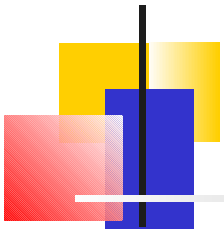
Resultado da execução??

```
abstract class Base {
    abstract void m();
    public Base() {
        System.out.println("Base: inicio construcao");
        m();
        System.out.println("Base: fim construcao");
    }
}

public class Derivada extends Base {
    int valor = 1;
    void m() {
        System.out.println("Derivada.m: " + valor);
    }
    public Derivada(int v) {
        System.out.println("Derivada: inicio construcao");
        valor = v;
        System.out.println("Derivada: fim construcao");
    }

    public static void main(String[] args) {
        new Derivada(10);
    }
}
```



Exemplo

- O resultado dessa execução pode ser explicado pela seqüência de ações que é obedecida para a construção de um objeto a partir do momento no qual seu construtor é invocado.
2. O espaço para o objeto é alocado e seu conteúdo é inicializado (bitwise) com zeros.
3. O construtor da classe base é invocado.
4. Os membros da classe são inicializados para o objeto, seguindo a ordem em que foram declarados na classe.
5. O restante do corpo do construtor é executado.



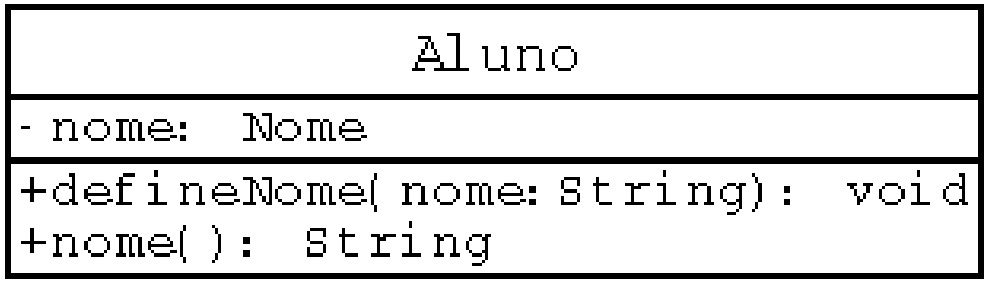
Exemplo

- O comportamento apresentado no exemplo difere daquele intuitivamente esperado por um programador que esteja analisando o código superficialmente.
- Em programas maiores -> situações de erro de difícil detecção.
- Recomendação: *Não invoque métodos no corpo de construtores a menos que isto seja seguro*.
- Métodos seguros para invocação a partir de construtores são aqueles que não podem ser redefinidos .

Especificação de uma classe

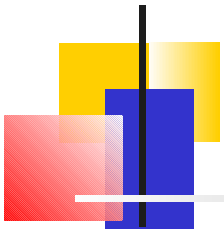
- A representação de classes em diagramas UML contempla três tipos básicos de informação:
- o nome da classe,
- os seus atributos e
- os seus métodos.
- Graficamente, um retângulo com três compartimentos internos representa esses grupos de informação.

Especificação de classe em UML



- **Atributos =** conjunto de propriedades da classe. Para cada propriedade, especifica-se:
  - nome: um identificador para o atributo.
  - tipo: o tipo do atributo (inteiro, real, caráter, outra classe, etc.)
  - valor_default: opcional, um valor inicial para o atributo.
  - visibilidade: opcional, o quão acessível é um atributo de um objeto a partir de outros objetos. Valores possíveis são:

    - (privativo), nenhuma visibilidade externa;
    + (público), visibilidade externa total; e
    # (protegido), visibilidade externa limitada.

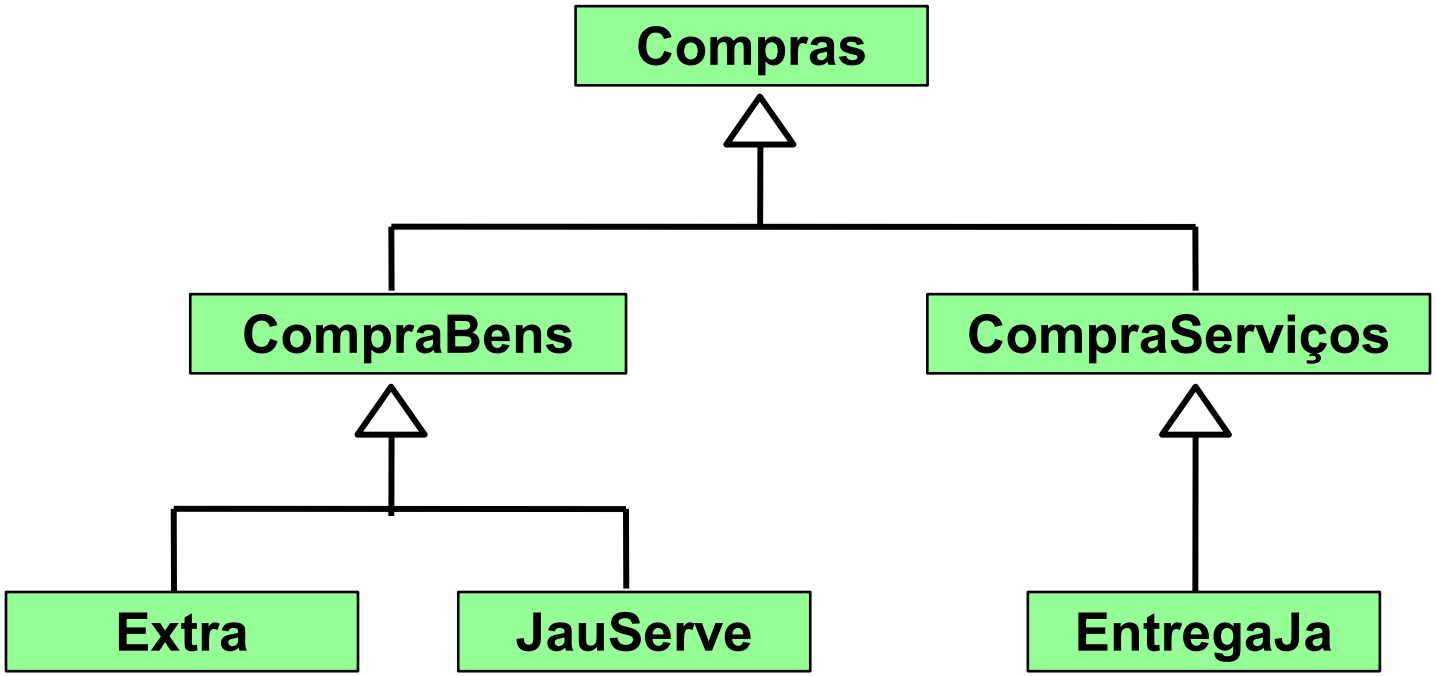


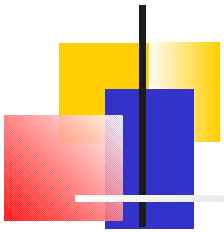
sobrecarga vs. sobrescrita

- **sobrecarga** lida com múltiplos métods com o mesmo nome na mesma classe, mas com diferentes assinaturas
- **sobrescrita** lida com dois métodos, um numa classe pai e outro numa classe filha, que têm a mesma assinatura
- sobrecarga possibilita definir uma operação similar de diferentes formas para diferentes parâmetros
- sobrescrita possibilita definir uma operação similar de diferentes formas para tipos de objetos diferentes

Hierarquia de Classes

- uma classe filha pode ser pai de uma outra classe filha formando uma hierarquia de classes





Hierarquia de Classes

- duas classes filhas da mesma classe pai são chamadas irmãs.
- as características comuns devem ser colocadas como mais altas na hierarquia
- um membro herdado é passado continuamente abaixo
- assim, uma classe filha herda de todos os seus ancestrais
- não existe uma só hierarquia que seja apropriada para todas as situações

a classe Object

- a classe chamada **`Object`** é definida no package **`java.lang`** da biblioteca padrão de Java
- todas as classes são derivadas dessa classe **`Object`**
- se uma classe não é explicitamente definida como filha de uma classe existente, é assumido que é filha da classe **`Object`**
- portanto, a classe **`Object`** é a classe raíz de todas as hierarquias de classes

a classe Object

- a classe **`Object`** contém poucos métodos úteis, que são herdados por todas as classes
- Ex: o método **`toString`** é definido na classe **`Object`**
- toda vez que definimos o método **`toString`**, estamos sobrescrevendo uma definição herdada
- o método **`toString`** na classe **`Object`** é definido para retornar uma string que contém o nome da classe objeto junto com alguma outra informação

a classe Object

- o método `equals` da classe `Object` retorna true se duas referências são aliases
- podemos sobrescrever `equals` em qualquer classe para definir a igualdade de alguma forma apropriada
- a classe `String` define o método `equals` para retornar true se dois objetos `String` contêm os mesmos caracteres
- os projetistas da classe `String` sobrescreveram o método `equals` herdado de `Object` numa versão mais adequada

Métodos e classes abstratos

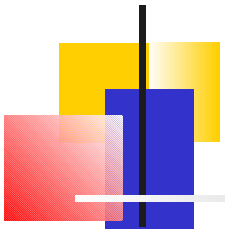
- **Um *método abstrato* é um método que não pode ser racionalmemte definido para uma classe, mas faz sentido para as extensões da classe.**

- Um método abstrato é um ´lugar´
- Qualquer **classe com um método abstrato é uma *classe abstrata.***
- Uma classe abstrata não pode ser instanciada
- Uma subclasse de uma classe abstrata é abstrata a menos que redefina todos os métodos abstratos.

Métodos e classes abstratos

```
abstract public class Shape {
    abstract public double area( );
    final public boolean lessThan( Shape rhs ) {
        return area( ) < rhs.area( );
    }
    final public double getArea( ) {
        return area( );
    }
}
```

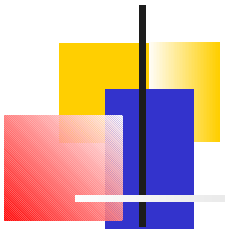
- a classe derivada deve implementar **`area`**, **e pode não redefinir `lessThan` ou `getArea`.**



classes Abstratas

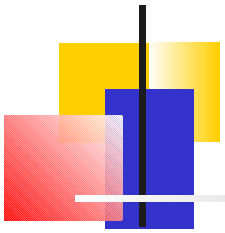
- uma classe *abstrata é um 'lugar' numa hierarquia de classes que representa* um conceito genérico
- **uma classe abstrata não pode ser instanciada !!**
- usamos modificador **`abstract`** no cabeçalho da classe para declara-la como abstrata:

```
public abstract class Product
{
    // contents
}
```



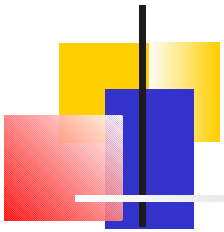
classes Abstratas

- uma classe abstrata geralmente contém métodos abstratos, ou seja, sem definição do que implementam (como uma interface)
- o modificador `abstract` deve ser aplicado a cada método abstrato
- além disso, uma classe abstrata pode conter métodos não-abstratos com definições completas
- uma classe declarada como abstrata não tem que conter métodos abstratos



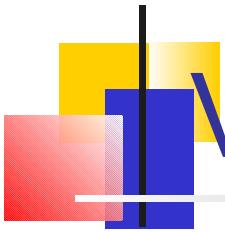
classes Abstratas

- a classe filha de uma classe abstrata deve sobrescrever os métodos abstratos da classe pai, ou ela será considerada abstrata
- um método abstrato não pode ser definido como **`final`** ou **`static`**
- o uso de classes abstratas é um elemento de projeto de software importante – possibilita estabelecer elementos comuns numa hierarquia, que são bastante genéricos pra instanciar



Visibilidade

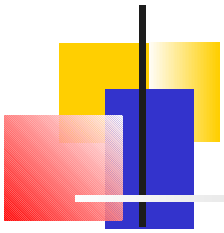
- é importante compreender uma questão relacionada com herança e visibilidade
- todas as variáveis e métodos de uma classe pai, mesmo os membros, são herdados pelas classes filhas
- como já mencionado, os membros private não podem ser referenciados pelo nome nas classes filhas
- porém, os membros private herdados pelas classes filhas existem e podem ser referenciados indiretamente



Visibilidade

- como a classe pai pode referenciar os membros private, a classe filha pode referencia-los indiretamente, usando os métodos da sua classe pai

- a referência **`super`** pode ser usada para referenciar a classe pai, mesmo que não exista nenhum objeto da classe pai



Código fonte Java

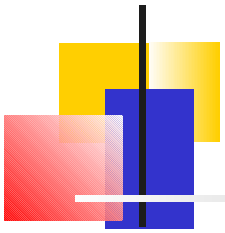
- constitui uma unidade de compilação, pode incluir comentários, declaração relacionadas a pacotes e pelo menos uma definição de classe ou de interface.
- Declarações relacionadas a pacotes:

```
package nome.do.pacote;
```

Opcional, mas se presente deve ser o primeiro comando do arquivo fonte. Indica que as definições que se seguem fazem parte do pacote especificado.

```
import nome.do.pacote.Classe;
import nome.do.pacote.*;
```

Indica que a classe especificada ou, no segundo caso, quaisquer classes do pacote especificado serão utilizadas no código fonte.



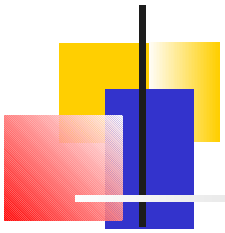
Pacote

Em grandes aplicações é preciso organizar as classes de maneira a

- evitar problemas com nomes duplicados de classes, e
- localizar o código da classe de forma eficiente.

Em Java, uma solução é a organização de classes e interfaces em pacotes.

Assim, uma classe Xyz que pertence a um pacote nome.do.pacote tem o nome completo **`nome.do.pacote.Xyz`** e o compilador Java espera encontrar **o arquivo `Xyz.class` em um subdiretório `nome/do/pacote`.** Este, por sua vez, deve estar localizado sob um dos diretórios especificados na variável de ambiente CLASSPATH.



fim.