BCC221
Programação Orientada a Objetos

Prof. Marco Antonio M. Carvalho

2014/2

decom
departamento
de computação

UFOP

Endereços Importantes

- Site da disciplina:
  http://www.decom.ufop.br/marco/
- *Moodle*:
  www.decom.ufop.br/moodle
- Lista de e-mails:
  bcc221-decom@googlegroups.com
- Para solicitar acesso:
  http://groups.google.com/group/bcc221-decom

Avisos

Avisos

Na aula Passada

- Métodos *static*
- Classe *Math*
- Promoção de Argumentos
- Sobrecarga de Métodos
- Composição
- Enumerações
  - Enumerações e Classes
- *static import*
- Criando Pacotes
- Acesso de Pacote

Na aula de hoje

- Herança
- Especificadores de Acesso
- Classe *Object*
- Exemplo
- Construtores em Subclasses
- Compilação
- Redefinição de Métodos
- Métodos e Classes *final*
- Engenharia de Software com Herança

Herança

- A herança é uma das características primárias da orientação a objetos
  - Uma forma de reuso de *software* pela qual uma classe nova é criada e absorve os membros de classes já existentes, aprimorando-os;
  - Diminui o tempo de implementação;
  - Aumenta a confiabilidade e qualidade do *software*
    - Desde que sejam utilizados bons componentes.

Herança

- Uma classe já existente e que é herdada é chamada de **superclasse**
  - A nova classe que herdará os membros é chamada de **subclasse**.
- Normalmente, uma subclasse adiciona os seus próprios atributos e métodos ao comportamento da superclasse
  - Logo, é uma forma especializada da superclasse;
  - Uma subclasse também pode vir a ser uma superclasse.
- A **superclasse direta** é a superclasse da qual a subclasse herda explicitamente
  - As outras são consideradas **superclasses indiretas**.

Herança

- Na **herança única**, uma subclasse herda somente de uma superclasse direta
  - Java não permite a realização de **herança múltipla**, em que uma subclasse pode herdar de mais de uma superclasse direta;
  - No entanto, é possível utilizar **interfaces** para desfrutar de alguns dos benefícios da herança múltipla evitando alguns dos problemas relacionados.

Herança

- Relembrando...
  - Herança define um relacionamento **é um**
    - Um objeto da subclasse **é um** objeto da superclasse;
    - O contrário **não é verdadeiro**.
  - problema com a herança é que a subclasse pode herdar métodos que não precisa ou que não deveria ter
    - Ainda, o método pode ser necessário, mas inadequado;
    - A classe pode sobrescrever (*override*) um método herdado para adequá-lo.

Especificadores de Acesso

Especificadores de Acesso

- ***public***:
  - Os membros *public* de uma classe são acessíveis em qualquer parte de um programa em que haja uma referência a um objeto da classe ou das subclasses.
- ***private***:
  - Membros *private* são acessíveis apenas dentro da própria classe.
- ***protected***:
  - Membros protected podem ser acessados por por membros da própria classe, de subclasses e de classes do mesmo pacote
    - *protected* também tem acesso de pacote.

Especificadores de Acesso

- Todos os membros *public* e *protected* de uma superclasse mantêm seus especificadores de acesso quando se tornam membros de uma subclasse
  - Subclasses se referem a estes membros simplesmente pelo nome;
- Quando uma subclasse sobrescreve um método da superclasse, o método original da superclasse ainda pode ser acessado quando antecedido pela palavra ***super*** seguida de .

super.metodo();

Classe *Object*

Classe *Object*

- A hierarquia das classes em Java é iniciada pela classe ***Object***
  - Todas as outras classes herdam (ou **estendem**) direta ou indiretamente a partir dela
    - Mesmo que não seja definido explicitamente.
  - Define um construtor e 11 métodos
    - Alguns devem ser sobrescritos pelas subclasses para melhor funcionamento.
  - Não possui atributos.

Classe *Object*

Métodos da classe Object	
clone()	*getclass()*
equals()	*hashCode()*
finalize()	*notify(), notifyAll()*
toString()	*wait() – 3 versões*

Classe *Object*

- ***clone***()
  - Método *protected*;
  - Retorna uma referência para *Object*
    - Exige um *cast* para o objeto original.
  - Realiza a cópia do objeto a partir do qual foi invocado;
  - Classes devem sobrescrevê-lo como um método público
    - Devem também implementar a interface *Cloneable*.
  - A implementação padrão realiza uma **cópia rasa**
    - Uma implementação sobrescrita normalmente realiza uma **cópia profunda**.

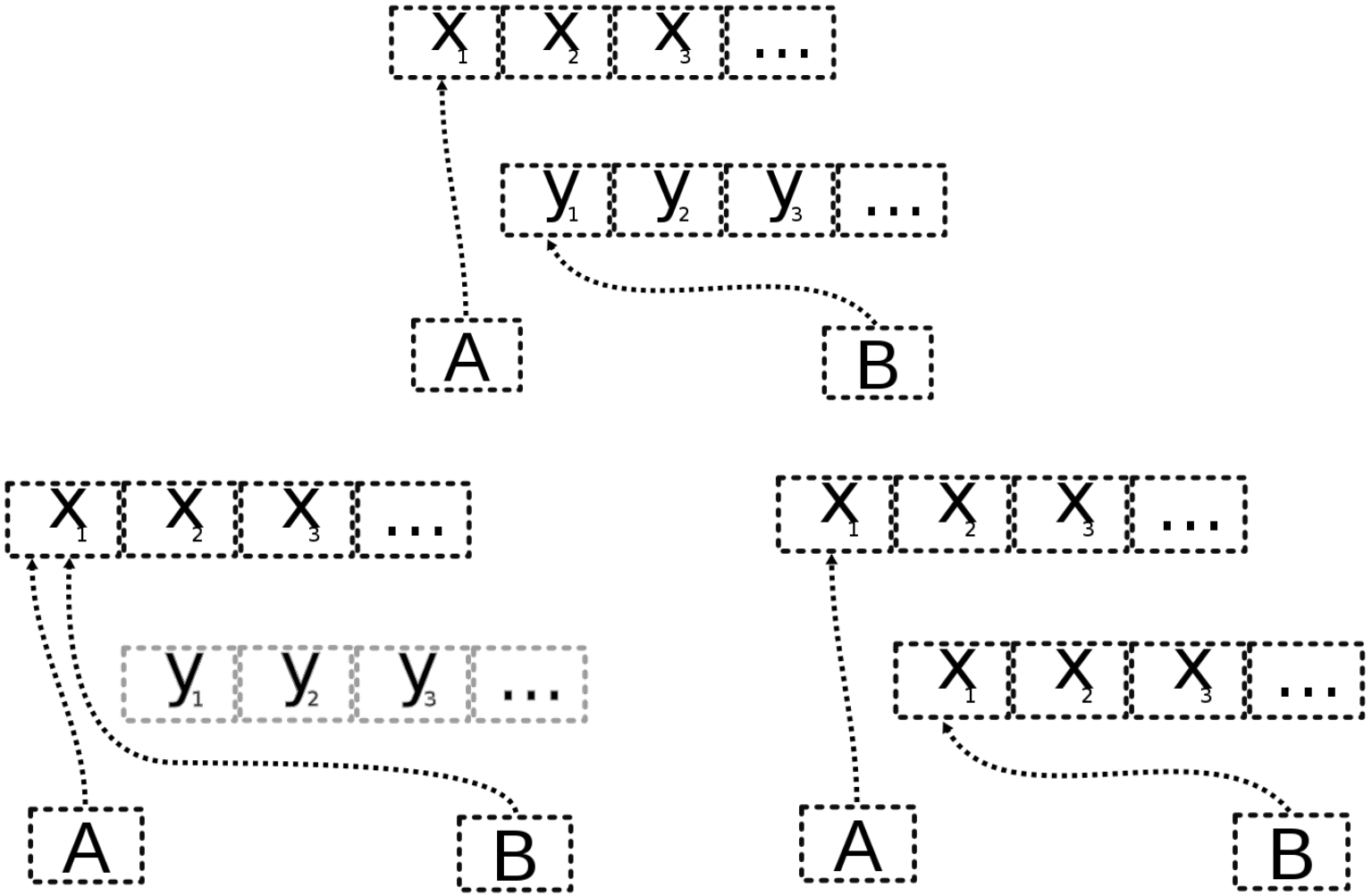
http://www.java-tips.org/java-se-tips/java.lang/how-to-implement-cloneable-interface.html

Cópia Rasa vs. Cópia Profunda

- A cópia rasa (*shallow copy*) realiza o mínimo de duplicação possível
  - É uma cópia de referência, não dos elementos
    - Não existe para classes que possuem apenas tipos primitivos.
  - Dois objetos compartilham os mesmos membros.
- A cópia profunda (*deep copy*) realiza o máximo de duplicação possível
  - Cria um novo objeto completo e independente;
  - Mesmo conteúdo do objeto clonado, membro a membro.

http://www.java2s.com/Code/Java/Class/ShallowCopyTest.htm

Cópia Rasa vs. Cópia Profunda



Classe *Object*

- ***equals***()
  - Compara dois objetos quanto a igualdade e retorna *true* caso sejam iguais, *false* caso contrário;
  - Quando dois objetos de uma classe em particular precisarem ser comparados, este método deve ser sobrescrito
    - Para considerar todos os membros.
  - Toda implementação deve retornar
    - *true* para comparação de um único objeto a.*equals*(a);
    - *false* se o argumento é null;
    - *true* sse a.*equals*(b) e b.*equals*(a) retornarem *true*;
    - Se a.*equals*(b) e a.*equals*(c) retornarem *true* então b.*equals*(c) retorna *true*.

http://www.javapractices.com/topic/TopicAction.do?Id=17

Classe *Object*

- ***finalize***()
  - Invocado pelo coletor de lixo automático para realizar a terminação de um objeto prestes a ser coletado;
  - Não há garantia de que o objeto será coletado, portanto, não há garantia de que este método será executado;
  - Não possui parâmetros e retorna *void*;
  - A implementação padrão não realiza nenhuma operação.

Classe *Object*

- ***getClass***()
  - Todos objetos em Java conhecem o seu tipo em tempo de execução;
  - Este método retorna um objeto da classe *Class* (pacote *java.lang*) que contém informações sobre o objeto, como o nome da classe (obtido pelo método *getName*()).

http://download.oracle.com/javase/8/docs/api/

Classe *Object*

- ***hashCode***()
  - Retorna um código *hash* (*int*) para o objeto a partir do qual foi invocado;
  - Utilizado por tabelas *hash*, definidas em *java.util.hashTable*;
  - Deve ser sobrescrito se o método *equals()* for sobrescrito.

Classe *Object*

- ***wait***()
  - Relacionada a *multithreading*;
  - Faz com que uma *thread* aguarde uma invocação para resumir sua execução, ou aguarde um determinado intervalo de tempo.

Classe *Object*

- ***notify***(), ***notifyAll***()
  - Relacionada a *multithreading*;
  - Respectivamente, "*acordam*" uma ou todas as *threads* que aguardam para resumir sua execução.

Classe *Object*

- ***toString***()
  - Retorna a representação do objeto que o invocou em formato de *string;*
  - A implementação padrão retorna os nomes do pacote e da classe, seguidos pela representação em hexadecimal do valor retornado pelo método ***hashCode***();
  - É recomendado que todas as subclasses sobrescrevam este método;
  - Pode ser utilizado em substituição de métodos ***print***().

Exemplo

Exemplo

- Consideremos novamente o exemplo de uma empresa que possui dois tipos de empregados
  - Comissionados (superclasse)
    - Recebem uma comissão sobre vendas.
  - Assalariados Comissionados (subclasse)
    - Recebem salário fixo e comissão sobre vendas.

ComissionEmployee.java


```
public class CommissionEmployee
{
      private String firstName;
      private String lastName;
      private String socialSecurityNumber;
      private double grossSales;
      private double commissionRate;

      public CommissionEmployee(String first, String last, String ssn, double sales, double rate)
      {
            //uma chamada  implicita ao construtor Object ocorre aqui
            firstName = first;
            lastName = last;
            socialSecurityNumber = ssn;
            setGrossSales( sales );
            setCommissionRate( rate );
      }
```

ComissionEmployee.java


```
// setter
public void setFirstName( String first )
{
      firstName = first;
}

//getter
public String getFirstName()
{
      return firstName;
}

//setter
public void setLastName( String last )
{
      lastName = last;
}

//getter
public String getLastName()
{
      return lastName;
}
```

ComissionEmployee.java


```
//setter
public void setSocialSecurityNumber( String ssn )
{
      socialSecurityNumber = ssn; // should validate
}
//getter
public String getSocialSecurityNumber()
{
      return socialSecurityNumber;
}
//setter
public void setGrossSales( double sales )
{
      grossSales = ( sales < 0.0 ) ? 0.0 : sales;
}
//getter
public double getGrossSales()
{
      return grossSales;
}
```

ComissionEmployee.java


```
//setter
public void setCommissionRate( double rate )
{
      commissionRate = ( rate > 0.0 && rate < 1.0 ) ? rate : 0.0;
}
//getter
public double getCommissionRate()
{
      return commissionRate;
}
//calcula o salário
public double earnings()
{
      return getCommissionRate() * getGrossSales();
}
```

ComissionEmployee.java


```
    //sobrescreve o método toString da classe Object
    public String toString()
    {
        return String.format( "%s: %s %s\n%s: %s\n%s: %.2f\n%s: %.2f",
        "commission employee", getFirstName(), getLastName(), "social security
        number", getSocialSecurityNumber(), "gross sales", getGrossSales(),
        "commission rate", getCommissionRate());
    }
}
```

Herança

- Construtores não são herdados
  - Logo, a classe *CommissionEmployee* não herda o construtor da classe *Object*;
  - Porém, o construtor da classe *CommissionEmployee* invoca implicitamente o construtor da classe *Object*;
  - A primeira tarefa de qualquer construtor é invocar o construtor da superclasse direta
    - Implicita ou explicitamente.
  - Se não houver uma chamada explícita, o compilador invoca o construtor padrão
    - Sem argumentos;
    - Não efetua nenhuma operação.

Herança

- O método ***toString***(), herdado da classe *Object* é sobrescrito na classe de exemplo
  - Retorna uma String que representa um objeto;
  - Este método é chamado implicitamente quando tentamos imprimir um objeto com **%s** no *printf*, por exemplo.
- O exemplo ainda utiliza o método *format* da classe *String*
  - Retorna uma *String* montada a partir de parâmetros.

BasePlusCommissionEmployee.java


```
//declaração de herança
public class BasePlusCommissionEmployee extends CommissionEmployee
{
      private double baseSalary;

      public BasePlusCommissionEmployee(String first, String last, String ssn,
                                        double sales, double rate, double salary)
      {
            //chama o construtor da superclasse
            super( first, last, ssn, sales, rate );
            setBaseSalary( salary );
      }

      //setter
      public void setBaseSalary( double salary )
      {
            baseSalary = ( salary < 0.0 ) ? 0.0 : salary;
      }
```

BasePlusCommissionEmployee.java


```
    //getter
    public double getBaseSalary()
    {
        return baseSalary;
    }

    //metodo sobrescrito
    public double earnings()
    {
        //invoca o metodo earnings da superclasse
        return getBaseSalary() + super.earnings();
    }

    //sobrescreve o método toString da classe Object
    public String toString()
    {
        return String.format("%s %s\n%s: %.2f", "base-salaried",
                             super.toString(), "base salary", getBaseSalary());
    }
}
```

Herança

- A herança é definida pela palavra reservada ***extends***;
- A subclasse invoca o construtor da superclasse explicitamente através da instrução

```
super( first, last, ssn, sales, rate );
```

- Esta deve ser a primeira ação em um construtor.

Herança

- Se um método realiza as operações necessárias em outro método, é preferível que ele seja chamado, ao invés de duplicarmos o código
  - Reduz a manutenção no código;
  - Boa prática de Engenharia de *Software*.
- No exemplo, invocamos o método *earnings*() da superclasse, já que ele é sobrescrito na subclasse

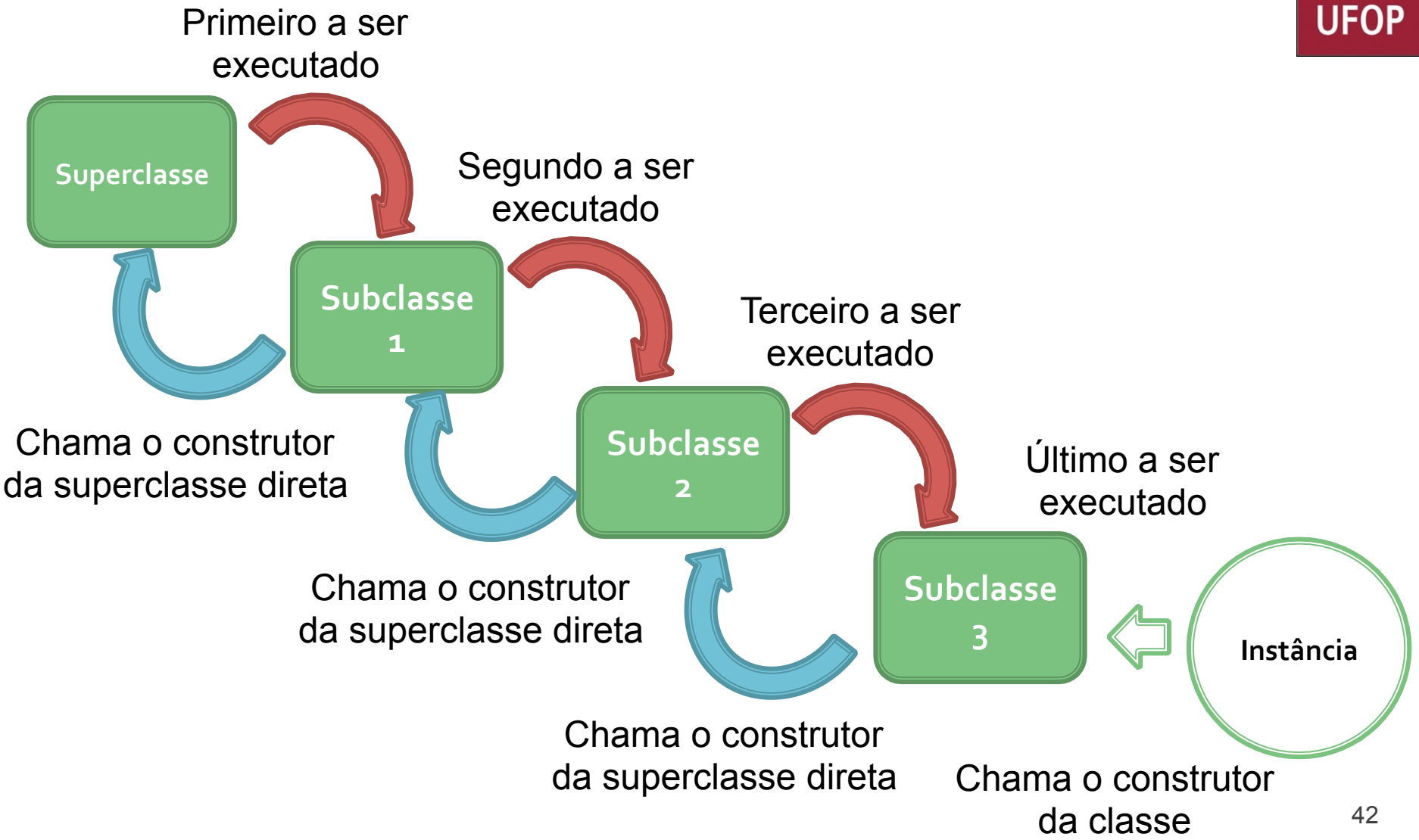
super.earnings();

Construtores em Subclasses

Construtores em Subclasses

- Como vimos no exemplo, instanciar uma classe derivada inicia uma cadeia de chamadas a construtores
  - O construtor da subclasse chama o construtor da superclasse antes de executar suas próprias ações;
  - O construtor da superclasse é chamada direta ou indiretamente (construtor *default*).
- Considerando um hierarquia de classes mais extensa:
  - O primeiro construtor **chamado** é o da última subclasse
    - E é o último a ser **executado**.
  - O último construtor **chamado** é o da classe *Object*
    - E é o primeiro a ser **executado**.

Construtores em Subclasses



Compilação

Compilação

- Compilamos cada classe separadamente
  - Uma maneira é compilar todas as classes de um diretório

javac *.java

Redefinição de Métodos

Redefinição de Métodos

- Subclasses podem redefinir métodos das superclasses
  - A assinatura pode até mudar, embora o nome do método permaneça;
  - A precedência é do método redefinido na classe derivada
    - Na verdade, este **substitui** o método da classe base na classe derivada.
  - Por exemplo, é possível sobrescrever um método *public* como *private*.

Redefinição de Métodos

- É comum que métodos redefinidos chamem o método original dentro de sua redefinição e acrescentem funcionalidades
  - Como no exemplo anterior, em que frases adicionais são impressas na redefinição do método *print()*.

Métodos e Classes *final*

Métodos e Classes *final*

- Uma variável ou atributo declarado com o modificador *final* é constante
  - Ou seja, depois de inicializada não pode ser **modificada**.
- Um método declarado com o modificador *final* não pode ser **sobrescrito**;
- Uma classe declarada com o modificador *final* não pode ser **estendida**
  - Embora possa ser utilizada em composições.

Engenharia de *Software* com Herança

Engenharia de *Software* com Herança

- Em uma hierarquia de herança, uma subclasse não necessita ter acesso ao código fonte da superclasse
  - Java exige apenas acesso ao arquivo ***.class*** da superclasse para que possamos compilar e executar uma subclasse.
- Esta característica é útil para *software* proprietário
  - Basta distribuí-lo em formato *bytecode*, não é necessário fornecer o código fonte;
  - No entanto, deve haver documentação precisa sobre o funcionamento da classe, para que outros programadores a compreendam.

Engenharia de *Software* com Herança

- Projetistas de sistemas orientados a objetos devem evitar a proliferação de classes
  - Cria problemas de administração;
  - Pode impedir o reuso de software
    - Uma biblioteca grande demais e confusa dificulta ao usuário encontrar a funcionalidade desejada.
- Uma alternativa é criar menos classes com maior funcionalidade substancial.

Perguntas?

Na próxima aula

- Polimorfismo

FIM