

HERANÇA, POLIMORFISMO E INTERFACES

Paradigmas de Programação

LEI - ISEP

Objectivos

- ❑ Conceito de herança
- ❑ Implementação de subclasses que herdam e redefinem (*override*) métodos da superclasse
- ❑ Compreender o conceito de polimorfismo
- ❑ Conhecer a superclasse comum Object e os seus métodos
- ❑ Utilização de *interface types*

Nesta lição, pretende-se aprender a noção de herança que expressa a relação entre classes especializadas e gerais

Conteúdos

- ❑ Hierarquias de herança
- ❑ Implementação de Subclasses
- ❑ *Overriding* de Métodos
- ❑ Polimorfismo
- ❑ **Object: A Superclasse Comum (aula TP)**
- ❑ *Interface Types*

3

Hierarquias de Herança

- ❑ Na programação orientada ao objeto, herança é uma relação entre:

- Uma *superclasse*: uma classe mais genérica



- Uma *subclasse*: uma classe mais especializada



- ❑ A subclasse ‘herda’ dados (variáveis) e comportamento (métodos) da superclasse

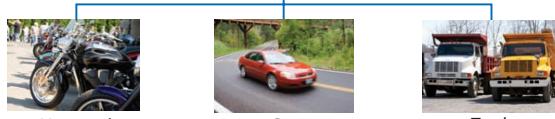
4

Uma Hierarquia Classe Veículo

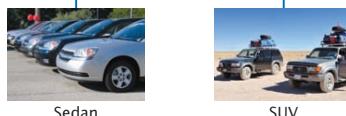
- Genérica



- Especializada



- Mais específica



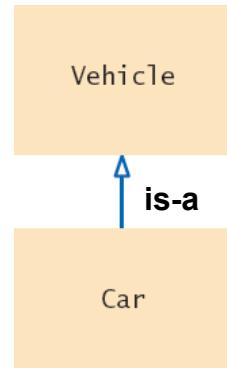
5

Princípio da Substituição

- Sendo a subclasse Car “*is-a*” Vehicle

- Car partilha características comuns com Vehicle
- É possível usar um objeto Car num programa que espera um objeto Vehicle

```
Car myCar = new Car(. . .);  
processVehicle(myCar);
```



A relação ‘*is-a*’ é representada num diagrama de classes por uma seta e significa que a subclasse pode comportar-se como um objeto da superclasse

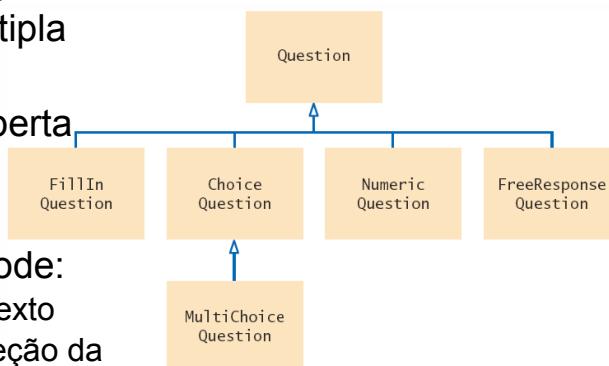
6

Hierarquia Questão de um Teste

- Existem diferentes tipos de questões num teste:

- 1) Preencher espaços
- 2) Escolha simples
- 3) Escolha múltipla
- 4) Numérica
- 5) Resposta Aberta

A 'raiz' da hierarquia é apresentada no topo



- Uma questão pode:
 - Mostrar o seu texto
 - Verificar a correção da resposta

7

Question.java (1)

```
1  /**
2   * A question with a text and an answer.
3  */
4  public class Question
5  {
6      private String text;
7      private String answer;
8
9
10 /**
11  * Constructs a question with empty question and answer.
12 */
13 public Question()
14 {
15     text = "";
16     answer = "";
17 }
18 /**
19  * Sets the question text.
20  * @param questionText the text of this question
21 */
22 public void setText(String questionText)
23 {
24     text = questionText;
25 }
```

A classe Question é a 'raiz' da hierarquia, também conhecida como a superclasse

- Suporta apenas Strings
- Não suporta:
 - Valores aproximados
 - Resposta de escolha múltipla

8

Question.java (2)

```
27  /**
28   * Sets the answer for this question.
29   * @param correctResponse the answer
30  */
31  public void setAnswer(String correctResponse)
32  {
33      answer = correctResponse;
34  }
35
36  /**
37   * Checks a given response for correctness.
38   * @param response the response to check
39   * @return true if the response was correct, false otherwise
40  */
41  public boolean checkAnswer(String response)
42  {
43      return response.equals(answer);
44  }
45
46  /**
47   * Displays this question.
48  */
49  public void display()
50  {
51      System.out.println(text);
52  }
53 }
```

9

QuestionDemo1.java

Program Run

```
1 import java.util.ArrayList;
2 import java.util.Scanner;
3
4 /**
5  * This program shows a simple quiz with one question.
6  */
7 public class QuestionDemo1
8 {
9     public static void main(String[] args)
10    {
11        Scanner in = new Scanner(System.in);
12
13        Question q = new Question();
14        q.setText("Who was the inventor of Java?");
15        q.setAnswer("James Gosling");
16
17        q.display();
18        System.out.print("Your answer: ");
19        String response = in.nextLine();
20        System.out.println(q.checkAnswer(response));
21    }
22 }
```

Who was the inventor of Java?
Your answer: James Gosling
true

Cria um objeto da
classe Question e usa
os métodos

10

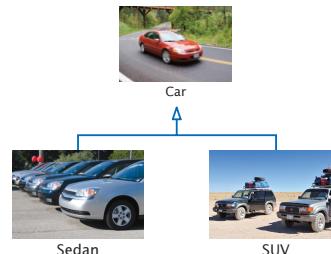
Boas Práticas

- ❑ Usar uma única classe para variação de valores, herança para variação de comportamento
 - Se dois veículos apenas diferem no consumo de combustível, usar uma variável de instância para a variação, não a herança

```
// Car instance variable  
double milesPerGallon;
```

- Se dois veículos têm comportamentos diferentes, usar herança

Não exagerar na utilização de herança



11

Implementação de Subclasses

- ❑ Consideremos a implementação da classe ChoiceQuestion para lidar com:

In which country was the inventor of Java born?

1. Australia
2. Canada
3. Denmark
4. United States

Vamos ver como definir uma subclasse e como uma subclasse herda automaticamente da sua superclasse

- ❑ De que forma ChoiceQuestion difere de Question?
 - Armazena escolhas (1,2,3 e 4) para além da questão
 - Terá que haver um método para adicionar escolhas múltiplas
 - O método display mostrará estas escolhas logo a seguir à questão, numeradas apropriadamente

12

Herdando de uma Superclasse

- ❑ Subclasses herdam da superclasse:
 - Todos os métodos públicos que a subclasse não reescreva (*override*)
 - Todas as variáveis de instância
- ❑ A Subclasse pode
 - Adicionar novas variáveis de instância
 - Adicionar novos métodos
 - Alterar a implementação de métodos herdados

Definir a subclasse especificando o que é diferente da superclasse



13

Reescrita (*Overriding*) de Métodos da Superclasse

- ❑ É possível reutilizar métodos da classe Question?
 - Os métodos herdados têm exatamente o mesmo comportamento
 - Se for necessário alterar a sua forma de funcionamento:
 - Escrever na subclasse um método mais especializado
 - Usar o nome do método da superclasse para o método que se pretende substituir (usar o mesmo nome)
 - Deve ter exatamente o mesmo conjunto de parâmetros
 - Conduz à reescrita (*override*) do método da superclasse
- ❑ O novo método será invocado com o mesmo nome quando é chamado num objeto da subclasse

Uma subclasse pode reescrever (*override*) um método da superclasse fornecendo uma nova implementação

14

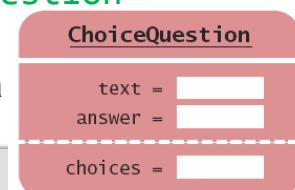
Planeamento da subclasse

- ❑ Usar a palavra reservada `extends` para definir a relação de herança a partir de `Question`
 - Herdar as variáveis `text` e `answer`
 - Adicionar nova variável de instância `choices`

```
public class ChoiceQuestion extends Question
{
    // This instance variable is added to the subclass
    private ArrayList<String> choices;

    // This method is added to the subclass
    public void addChoice(String choice, boolean correct) { . . . }

    // This method overrides a method from the superclass
    public void display() { . . . }
}
```



15

Declaração da Subclasse

- ❑ A subclasse herda da superclasse e estende ‘`extends`’ a funcionalidade da superclasse

Declare instance variables that are **added** to the subclass.

Declare methods that are **added** to the subclass.

Declare methods that the subclass **overrides**.

The reserved word `extends` denotes inheritance.

```
Subclass           Superclass
public class ChoiceQuestion extends Question
{
    private ArrayList<String> choices
    public void addChoice(String choice, boolean correct) { . . . }

    public void display() { . . . }
}
```

16

Implementação de addChoice

- ❑ O método recebe dois parâmetros
 - O texto para a escolha
 - Um booleano indicando se se trata de uma escolha correta
- ❑ Adiciona texto como uma escolha, adiciona o número da escolha ao texto e chama o método herdado `setAnswer`

```
public void addChoice(String choice, boolean correct)
{
    choices.add(choice);
    if (correct)
    {
        // Convert choices.size() to string
        String choiceString = "" + choices.size();
        setAnswer(choiceString);           setAnswer() é equivalente
    }                                     a this.setAnswer()
}
```

17

Erro Comum (1)



- ❑ Replicação de variáveis de instância da Superclasse
 - Uma subclasse não pode aceder às variáveis de instância da superclasse

```
public class Question
{
    private String text;
    private String answer;
    ...
}

public class ChoiceQuestion extends Question
{
    ...
    text = questionText;    // Compiler Error!
```

18

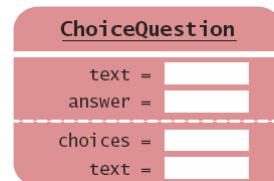
Erro Comum (1)



- Não tentar corrigir o erro de compilação com uma nova variável de instância que tenha o mesmo nome

```
public class ChoiceQuestion extends Question
{
    private String text; // Second copy
```

- O construtor inicializa a segunda variável `text` (subclasse)
- O método `display` acede à primeira (superclasse)



19

Erro Comum (2)



- Confundir *Super* e *Subclasses*
 - O uso da terminologia super e sub pode ser confuso
 - A Subclasse `ChoiceQuestion` é uma versão estendida 'extended' e mais elaborada do que `Question`
 - Trata-se de uma versão 'super' de `Question`?... NÃO
- A terminologia Super e Subclasse provém da teoria de conjuntos
 - `ChoiceQuestion` é um elemento de um **subconjunto** de elementos herdados a partir de `Question`
 - O conjunto de elementos `Question` é um **superconjunto** de elementos `ChoiceQuestion`

20

Reescrita (*Overriding*) de Métodos

- ❑ A classe **ChoiceQuestion** necessita de um método **display** que *overrides* o método **display** da classe **Question**
- ❑ Correspondem a duas implementações distintas
- ❑ Os dois métodos denominados **display** são:
 - **Question display**
 - Mostra a variável de instância String **text**
 - **ChoiceQuestion display**
 - Reescreve o método **Question display**
 - Mostra a variável de instância String **text**
 - Mostra a lista local de escolhas (**choices**)

21

Chamada de Métodos da Superclasse

In which country was the inventor of Java born?

1. Australia
2. Canada
3. Denmark
4. United States

- ❑ Consideremos o método **display** da classe **ChoiceQuestion**
 - Será necessário mostrar a questão E a lista de alternativas (**choices**)
- ❑ **text** é variável de instância privada da superclasse
 - ❑ Como obter acesso à variável para mostrar a questão?
 - ❑ Chamar o método **display** da superclasse **Question**!
 - ❑ A partir de uma subclasse, adicionamos um prefixo ao nome do método:
super.

```
public void display()
{
    // Display the question text
    super.display(); // OK
    // Display the answer choices
    . .
}
```

22

QuestionDemo2.java (1)

```
1 import java.util.Scanner;
2
3 /**
4  * This program shows a simple quiz with two choice questions.
5 */
6 public class QuestionDemo2
7 {
8     public static void main(String[] args)
9     {
10         ChoiceQuestion first = new ChoiceQuestion();
11         first.setText("What was the original name of the Java language?");
12         first.addChoice("*?", false);
13         first.addChoice("Duke", false);
14         first.addChoice("Oak", true);
15         first.addChoice("Gosling", false);
16
17         ChoiceQuestion second = new ChoiceQuestion();
18         second.setText("In which country was the inventor of Java born?");
19         second.addChoice("Australia", false);
20         second.addChoice("Canada", true);
21         second.addChoice("Denmark", false);
22         second.addChoice("United States", false);
23
24         presentQuestion(first);
25         presentQuestion(second);
26     }
}
```

Cria dois objetos da classe
ChoiceQuestion

Chama presentQuestion (próxima página)

23

QuestionDemo2.java (2)

```
28 /**
29  * Presents a question to the user and checks the response.
30  * @param q the question
31 */
32 public static void presentQuestion(ChoiceQuestion q)
33 {
34     q.display();
35     System.out.print("Your answer: ");
36     Scanner in = new Scanner(System.in);
37     String response = in.nextLine();
38     System.out.println(q.checkAnswer(response));
39 }
40 }
```

Usa o método display
de ChoiceQuestion
(subclasse)

24

ChoiceQuestion.java (1)

```
1 import java.util.ArrayList;
2
3 /**
4  * A question with multiple choices.
5 */
6 public class ChoiceQuestion extends Question
7 {
8     private ArrayList<String> choices;
9
10    /**
11     * Construtor
12     */
13    public ChoiceQuestion()
14    {
15        choices = new ArrayList<String>();
16    }
17
18    /**
19     * Adds an answer choice to this question.
20     * @param choice the choice to add
21     * @param correct true if this is the correct choice, false otherwise
22     */
23    public void addChoice(String choice, boolean correct)
24    {
25        choices.add(choice);
26        if (correct)
27        {
28            // Convert choices.size() to string
29            String choiceString = "" + choices.size();
30            setAnswer(choiceString);
31        }
32    }
```

Herda a partir da classe Question

Novo método addChoice

25

ChoiceQuestion.java (2)

```
33
34     public void display()
35     {
36         // Display the question text
37         super.display();
38         // Display the answer choices
39         for (int i = 0; i < choices.size(); i++)
40         {
41             int choiceNumber = i + 1;
42             System.out.println(choiceNumber + ": " + choices.get(i));
43         }
44     }
45 }
```

Reescrita do método display

Program Run

```
Who was the inventor of Java?  
Your answer: Bjarne Stroustrup  
false  
In which country was the inventor of Java born?  
1: Australia  
2: Canada  
3: Denmark  
4: United States  
Your answer: 2  
true
```

26

Erro Comum (3)



▫ Sobrecarga (*Overloading*) acidental

```
println(int x);
println(String s); // Overloaded
```

- O **overloading** ocorre quando dois métodos partilham o nome mas têm parâmetros diferentes
- **Overriding** ocorre quando a subclasse define um método com o mesmo nome e exatamente os mesmos parâmetros do método da superclasse
 - Método `display()` da classe `Question`
 - Método `display()` da classe `ChoiceQuestion`
- Se pretendemos **override**, mas alteramos os parâmetros, estaremos a **overloading** o método herdado e não a reescrevê-lo (**overriding**) – passaremos a ter 2 métodos
 - Ex: Método `display(PrintStream out)` de `ChoiceQuestion`

27

Erro Comum (4)



▫ Esquecer o uso de `super` aquando da invocação de um método da Superclasse

- Vamos assumir que `Manager` herda de `Employee`
 - `getSalary` é um método reescrito de `Employee`
 - `Manager.getSalary` inclui um bónus

```
public class Manager extends Employee
{
    ...
    public double getSalary()
    {
        double baseSalary = getSalary(); // Manager.getSalary
        // should be super.getSalary(); // Employee.getSalary
        return baseSalary + bonus;
    }
}
```

28

Chamada do Construtor da Superclasse

- Quando uma subclasse é instanciada, será invocado o construtor da superclasse sem argumentos
- Se preferirmos chamar um construtor mais específico, podemos invocá-lo substituindo o nome da superclasse pela palavra reservada `super` seguida por `()`:

```
public ChoiceQuestion(String questionText)
{
    super(questionText);
    choices = new ArrayList<String>();
}
```

- Deve ser a primeira instrução no construtor

29

Construtor com Inicialização da Superclasse

- Para inicializar variáveis de instância privadas na superclasse, podemos invocar um construtor específico

```
The superclass constructor is called first.  
The constructor body can contain additional statements.
```

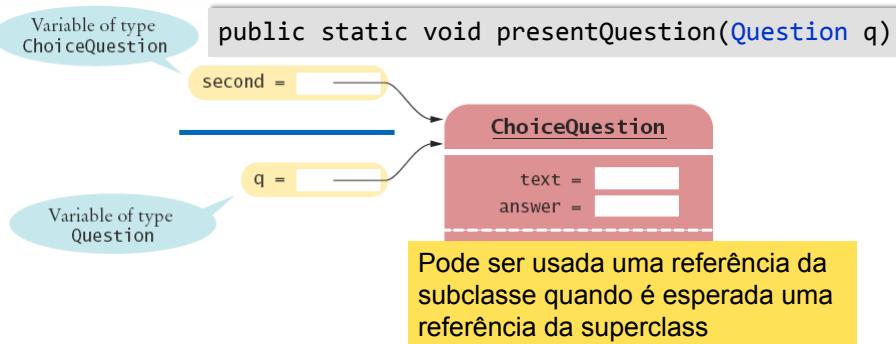
```
public ChoiceQuestion(String questionText)
{
    super(questionText);
    choices = new ArrayList<String>();
}
```

If you omit the superclass constructor call, the superclass constructor with no arguments is invoked.

30

Polimorfismo

- ❑ QuestionDemo2 passou dois objetos **ChoiceQuestion** ao método **presentQuestion**
 - É possível escrever um método **presentQuestion** que mostre questões dos tipos **Question** e **ChoiceQuestion**?
 - **De que maneira?**



31

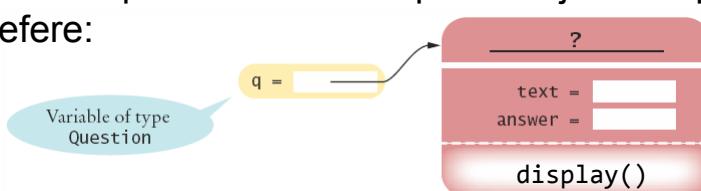
Que versão do método **display** é chamado?

- ❑ **presentQuestion** chama o método **display** independentemente do tipo do parâmetro:

```
public static void presentQuestion(Question q)
{
    q.display();
    ...
}
```

- ❑ Se é passado um objeto da classe **Question**:
 - **Question display**
- ❑ Se é passado um objeto da classe **ChoiceQuestion**:
 - **ChoiceQuestion display**

- ❑ A variável **q** não conhece o tipo do objeto ao qual se refere:



32

Benefícios do Polimorfismo

- ❑ Em Java, a chamada de métodos é *sempre determinada pelo tipo do objeto, não* pela variável que contém a referência do objeto
 - Este mecanismo é denominado *dynamic lookup*
 - O *dynamic lookup* permite tratar objetos de diferentes classes de maneira uniforme
- ❑ Esta característica é chamada de **polimorfismo**
- ❑ Ao pedirmos a vários objetos que executem uma tarefa, cada objeto executa-a à sua maneira
- ❑ O polimorfismo facilita a extensibilidade dos programas

33

QuestionDemo3.java (1)

```
1 import java.util.Scanner;
2 /**
3  * This program shows a simple quiz with two question types.
4 */
5 public class QuestionDemo3
6 {
7     public static void main(String[] args)
8     {
9         Question first = new Question();
10        first.setText("Who was the inventor of Java?");
11        first.setAnswer("James Gosling");
12
13        ChoiceQuestion second = new ChoiceQuestion();
14        second.setText("In which country was the inventor of Java born?");
15        second.addChoice("Australia", false);
16        second.addChoice("Canada", true);
17        second.addChoice("Denmark", false);
18        second.addChoice("United States", false);
19
20        presentQuestion(first);
21        presentQuestion(second);
22    }
23 }
```

Cria um objeto da classe Question

Cria um objeto da classe ChoiceQuestion, usa o novo método addChoice

Chama presentQuestion (próxima página) passando ambos os tipos de objetos

34

QuestionDemo3.java (2)

```
24
25  /**
26   * Presents a question to the user and checks the response.
27   * @param q the question
28  */
29  public static void presentQuestion(Question q)
30  {
31      q.display();
32      System.out.print("Your answer: ");
33      Scanner in = new Scanner(System.in);
34      String response = in.nextLine();
35      System.out.println(q.checkAnswer(response));
36  }
37 }
```

Recebe um parâmetro do tipo da superclasse

Usa o método display apropriado

35

Dynamic Lookup e o Parâmetro Implícito

- Consideremos que mudamos o método `presentQuestion` para o interior da classe `Question` e chamamo-lo da seguinte forma:

```
ChoiceQuestion cq = new ChoiceQuestion();
cq.setText("In which country was the inventor of Java born?");
. .
cq.presentQuestion(); void presentQuestion()
{
    display();
    System.out.print("Your answer: ");
    Scanner in = new Scanner(System.in);
    String response = in.nextLine();
    System.out.println(checkAnswer(response));
}
```

- Que métodos `display` e `checkAnswer` serão chamados?

36

Dynamic Lookup no Métodos

- ❑ Podemos acrescentar o parâmetro implícito ao código para tornar mais claro
 - Devido ao mecanismo de *dynamic lookup*, as versões dos métodos `display` e `checkAnswer` da subclasse `ChoiceQuestion` serão chamados automaticamente
 - Isto sucede apesar do método `presentQuestion` estar declarado na classe `Question`, a qual não tem conhecimento acerca da classe `ChoiceQuestion`

```
public class Question
{
    void presentQuestion()
    {
        this.display();
        System.out.print("Your answer: ");
        Scanner in = new Scanner(System.in);
        String response = in.nextLine();
        System.out.println(this.checkAnswer(response));
    }
}
```

37

Classes Abstratas (1)

- Se for desejável **forçar** a reescrever nas subclasses um método de uma classe base, podemos declarar o método como `abstract`
- Não é possível instanciar um objeto que tem métodos `abstract`
 - Portanto, a classe é considerada `abstract`

```
public abstract class Account
{
    public abstract void deductFees(); // no method implementation
    ...
    public class SavingsAccount extends Account // Not abstract
    {
        public void deductFees() // Provides an implementation
        {
            // method implementation. . .
        }
    }
}
```

- Se estendermos a classe `abstract`, é necessário implementar todos os métodos abstratos

38

Classes Abstratas (2)

- Membros de uma classe abstrata:
 - Uma classe abstrata pode ter atributos estáticos e métodos estáticos
 - Um membro estático é acedido através de uma referência à classe:
`AbstractClass.staticMethod()`
- Quando uma classe abstrata implementa uma interface:
 - Uma classe que implemente uma interface deve implementar todos os métodos da interface
 - No entanto, é possível que a classe não implemente todos os métodos da interface desde que a classe seja declarada como abstrata:

```
abstract class X implements Y {  
    // implementa alguns dos métodos de Y  
}  
  
class XX extends X {  
    // implementa os restantes métodos de Y  
}
```

Neste caso, a classe X deve ser abstrata porque não implementa completamente a classe Y, mas a classe XX já implementa Y

39

Referências Abstratas

- Uma classe que pode ser instanciada é denominada classe concreta
- Não é possível instanciar um objeto que tenha métodos **abstract**
 - Mas podemos declarar uma referência a um objeto cujo tipo seja uma classe **abstract**

```
Account anAccount;          // OK: Reference to abstract object  
anAccount = new Account(); // Error: Account is abstract  
anAccount = new SavingsAccount(); // Concrete class is OK  
anAccount = null;           // OK
```

- Isto permite o polimorfismo mesmo que baseado em classes **abstract**

Utilização de classes abstratas – quando se pretende forçar os programadores a criarem subclasses

40

Métodos e Classes Final

(aula TP)

- É possível **evitar** que outros programadores criem subclasses e reescrevam métodos usando `final`
- A classe String na biblioteca Java é um exemplo:

```
public final class String { . . . }
```

- Exemplo de um método que não pode ser reescrito:

```
public class SecureAccount extends BankAccount
{
    .
    .
    public final boolean checkPassword(String password)
    {
        .
        .
    }
}
```

41

Acesso Protegido (`protected`)

(aula TP)

- Quando implementámos o método `display` da classe `ChoiceQuestion`, o método `display` pretendia aceder à variável de instância `text` da superclasse, mas esta era `private`
- Optou-se por usar um método da superclasse para mostrar o conteúdo de `text`
- O Java dispõe de uma solução mais elegante
 - A superclasse pode declarar uma variável de instância como `protected` em vez de `private`
 - Dados `protected` num objeto podem ser acedidos pelos métodos da classe e das subclasses
 - Mas podem também ser acedidos por todas as outras classes do mesmo package!

```
public class Question
{
    protected String text;
    .
    .
}
```

Se se pretende garantir o acesso aos dados apenas aos métodos das subclasses, é preferível tornar os métodos de acesso protegidos

42

Passos para Aplicação de Herança

- ❑ Como exemplo, consideremos um banco que oferece aos seus clientes os seguintes tipos de contas:
 - 1) Uma conta poupança (*savings*) que remunera juros. A taxa é aplicada mensalmente e é baseada num saldo mínimo mensal
 - 2) Uma conta à ordem (*checking*) não remunerada, oferece três levantamentos gratuitos por mês e cobra \$1 de taxa por cada levantamento adicional
- ❑ O programa fará a gestão de um conjunto de contas de ambos os tipos
 - Deve ser estruturado de forma a que outros tipos de contas possam ser adicionadas sem afetar o processamento dos tipos existentes
- ❑ Menu: D)epósito L)evantamento M)ês e S)air
 - Para depósitos e levantamentos, pedir o número de conta e montante. Mostrar o saldo da conta após cada transação
 - No comando “Mês”, acumular os juros ou inicializar o contador de transações, dependendo do tipo de conta. Depois mostrar o saldo de todas as contas

43

Passos para Aplicação de Herança

- 1) Identificar as classes que são parte da hierarquia

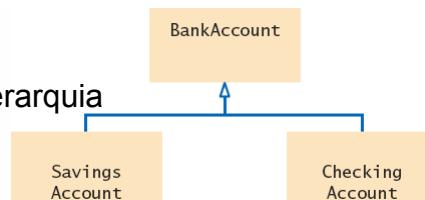
SavingsAccount

CheckingAccount

- 2) Organizar as classes numa hierarquia

Com base na superclasse

BankAccount



- 3) Determinar as responsabilidades comuns

- a. Escrever pseudocódigo para cada tarefa
- b. Encontrar tarefas comuns

44

Aplicação de Herança

For each user command

If it is a deposit or withdrawal

Deposit or withdraw the amount from the specified account.

Print the balance.

If it is month end processing

For each account

Call month end processing.

Print the balance.

Deposit money.
Withdraw money.
Get the balance.
Carry out month end processing.

45

Passos para Aplicação de Herança

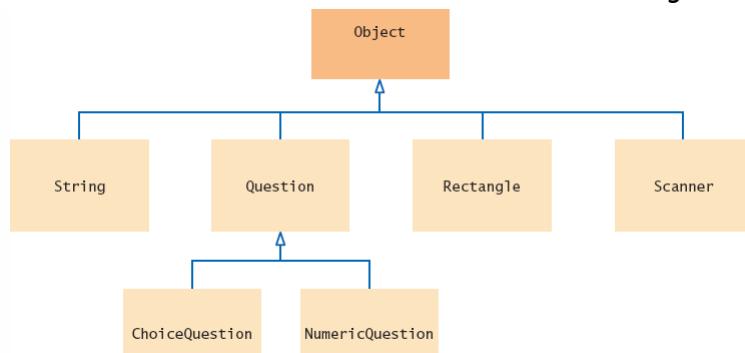
- 4) Decidir que métodos são reescritos nas subclasses
 - Para cada subclasse e responsabilidade comum, decidir quando o comportamento pode ser herdado e quando necessita ser reescrito
- 5) Declarar a interface pública de cada subclasse
 - Tipicamente, as subclasses têm responsabilidades que ultrapassam as da superclasse. Identificá-las, assim como os métodos que necessitam de ser reescritos
 - Será também necessário especificar como os objetos das subclasses devem ser construídos
- 6) Identificar as variáveis de instância
 - Identificar as variáveis de instância para cada classe. Colocar as variáveis de instância comuns a todas as classes na base da hierarquia
- 7) Implementar construtores e métodos
- 8) Construir objetos de diferentes subclasses e processá-los

46

Object: A Superclasse Raiz

(aula TP)

- Em Java, qualquer classe que é declarada sem a cláusula `extends` estende automaticamente a classe Object



Os métodos da classe Object são genéricos. Vamos ver como reescrever o método `toString`

47

Definição de um Método `toString`

(aula TP)

- O método `toString` devolve uma String contendo a representação para cada objeto
- A classe `Rectangle` (`java.awt`) tem um método `toString`
 - Podemos invocar o método `toString` diretamente

```
Rectangle box = new Rectangle(5, 10, 20, 30);
String s = box.toString();           // Call toString directly
// Sets s to "java.awt.Rectangle[x=5,y=10,width=20,height=30]"
```

- O método `toString` também pode ser invocado implicitamente quando se concatena uma String com um objeto:

```
System.out.println("box=" + box); // Call toString implicitly
```

- O compilador pode invocar o método `toString` porque sabe que qualquer objeto possui um método `toString`:
 - Qualquer classe estende a classe Object e pode reescrever o método `toString`

48

Reescrita do Método `toString`

(aula TP)

- Exemplo: Reescrever o método `toString` para a classe `BankAccount`

```
BankAccount momsSavings = new BankAccount(5000);
String s = momsSavings.toString();
// Sets s to something like "BankAccount@d24606bf"
```

- Imprime o nome da classe, seguido pelo código de hash usado para identificar objetos

- Podemos pretender incluir informação sobre o objeto

```
public class BankAccount
{
    public String toString()
    {
        // returns "BankAccount[balance=5000]"
        return "BankAccount[balance=" + balance + "]";
    }
}
```

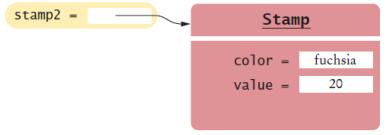
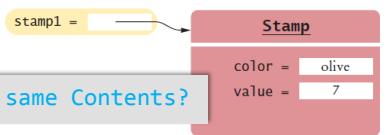
Reescreve o método `toString` para obter uma string que descreva o estado do objeto

49

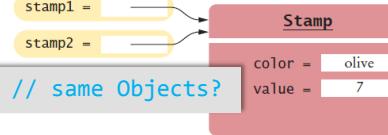
Reescrita do Método `equals` (aula TP)

- O método `equals` da classe `Object`, para quaisquer duas referências não nulas `x` e `y`, devolve `true` se e apenas se `x` e `y` se referem ao mesmo objeto se (`x == y` tem o valor `true`)

```
if (stamp1.equals(stamp2)) . . . // same Contents?
```



```
if (stamp1 == stamp2) . . . // same Objects?
```



50

Reescrita do Método `equals` (aula TP)

- A classe Object especifica o tipo do parâmetro como `Object`

```
public class Stamp {  
    private String color;  
    private int value;  
    . . .  
    public boolean equals(Object otherObject)  
    {  
        . . .  
    }  
    . . .  
    public boolean equals(Stamp otherObject)  
    {  
        Stamp other = (Stamp) otherObject;  
        return color.equals(other.color)  
            && value == other.value;  
    }  
}
```

O método `equals` da classe `Stamp` deve declarar o mesmo tipo de parâmetro do método `equals` de `Object` para rescrevê-lo

Fazer cast do parâmetro da classe `Stamp`

51

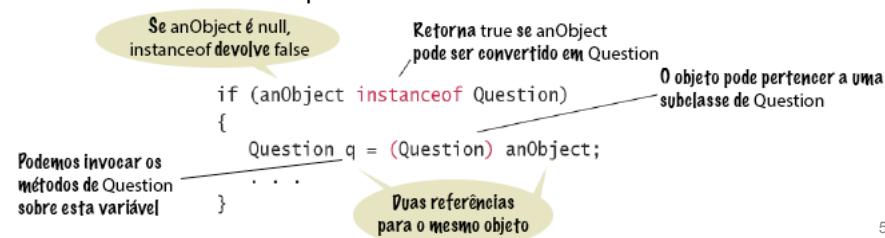
Operador `instanceof`

- É possível armazenar a referência de uma subclasse numa variável declarada como referência da superclasse
- A conversão oposta também é possível:
 - De uma referência da superclasse para uma referência da subclasse
 - Se temos uma variável do tipo `Object` e sabemos que ela armazena uma referência do tipo `Question`, podemos convertê-la: `Question q = (Question) obj;`
- Para garantir que se trata de um objeto do tipo `Question`, é possível verificá-lo com o operador `instanceof`: `if (obj instanceof Question) { Question q = (Question) obj; }`

52

Utilização de instanceof

- ❑ A utilização do operador `instanceOf` envolve também a conversão de tipos
 - Devolve true se é possível converter de forma segura um tipo de objeto noutro tipo
- ❑ A conversão permite o uso de métodos do novo objeto
 - Frequentemente usado para fazer uma referência mais específica
 - Converter a partir de um referência de Objet para um tipo de classe mais específica



53

Erro Comum (5)



- ❑ Não fazer testes de tipo

```
if (q instanceof ChoiceQuestion) // Evitar
{
    // Executar a tarefa sobre ChoiceQuestion
}
else if (q instanceof Question)
{
    // Executar a tarefa sobre Question
}
```

- Estratégia a evitar – se uma nova classe for adicionada, todos estes terão que ser revistos
 - P.ex. quando adicionamos a classe NumericQuestion
- Deixemos o polimorfismo selecionar o método correto:
 - Declarar um método `executarAtarefa` na superclasse
 - Reescrevê-lo nas subclasses

54

Herança e o Método `toString`

- Em vez de escrever o tipo do objeto no método `toString`
 - Usar `getClass` (herdado de `Object`) na superclasse

```
public class BankAccount { . . .
    public String toString()
    {
        return getClass().getName() + "[balance=" + balance + "]";
    }
}
```

- Depois usar herança, chamar `toString` da superclasse

```
public class SavingsAccount extends BankAccount
{
    . . .
    public String toString()
    {
        return super.toString() + "[interestRate=" + intRate + "]";
    } // returns SavingsAccount[balance= 10000][interestRate= 5]
}
```

Isto permite que a superclass mostre as variáveis de instância privadas

55

Herança e o Método `equals`

- E se chamarmos `stamp1.equals(x)` em que `x` não seja um objeto `Stamp`?
 - Usando o operador `instanceOf`, poderia ser possível que um `otherObject` pertença a uma subclasse de `Stamp`
- Usar o método `getClass` para comparar a classe do objeto com o objeto recebido no argumento

```
public boolean equals(Object otherObject)
{
    if (otherObject == null) { return false; }
    if (getClass() != otherObject.getClass()) { return false; }
    Stamp other = (Stamp) otherObject;
    return color.equals(other.color) && value == other.value;
}
```

Garante a comparação dos mesmos tipos

56

Interface Types

- ❑ Uma **interface** é um tipo especial de declaração que lista um conjunto de métodos e suas assinaturas
 - Uma classe que ‘*implements*’ uma **interface** deve implementar todos os métodos da **interface**
 - Assemelha-se a uma classe, mas existem diferenças:
 - Todos os métodos numa interface são abstratos:
Têm um nome, parâmetros e um tipo de retorno, mas não têm uma implementação
 - Todos os métodos numa interface são públicos automaticamente
 - Uma interface não pode ter variáveis de instância
 - Uma interface não tem métodos estáticos

```
public interface Measurable  
{  
    double getMeasure();  
}
```

Uma interface Java declara um conjunto de métodos e suas assinaturas

57

Interface: Sintaxe de Declaração

- ❑ Declaração de uma **interface** e de uma classe que **implementa** (*implements*) a **interface**.

```
Interface methods  
are always public.  
public interface Measurable  
{  
    double getMeasure();  
}  
  
Other  
BankAccount  
methods.  
  
public class BankAccount implements Measurable  
{  
    . . .  
    public double getMeasure()  
    {  
        return balance;  
    }  
}  
  
Interface methods  
have no implementation.  
A class can implement one  
or more interface types.  
Implementation for the method that  
was declared in the interface type.
```

58

Utilização das Interfaces

- Podemos usar a interface `Measurable` para implementar um método estático “universal” para cálculo de médias:

```
public interface Measurable
{
    double getMeasure();
}
```

```
public static double average(Measurable[] objs)
{
    if (objs.length == 0) return 0;
    double sum = 0;
    for (Measurable obj : objs)
    {
        sum = sum + obj.getMeasure();
    }
    return sum / objs.length;
}
```

59

Implementação de uma *Interface*

- Uma classe pode ser declarada para **implementar** uma interface
 - A classe deve implementar todos os métodos da interface

```
public class BankAccount implements Measurable
{
    public double getMeasure()
    {
        return balance;
    }
    ...
}
```

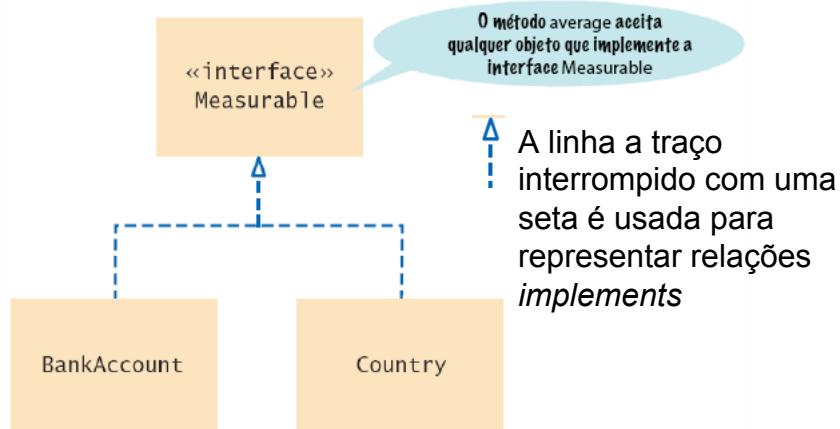
Usar a palavra reservada
`implements` na
declaração da classe

```
public class Country implements Measurable
{
    public double getMeasure()
    {
        return area;
    }
    ...
}
```

Os métodos da interface
devem ser declarados como
`public`

60

Diagrama de Implementação



61

MeasureableDemo.java (1)

```
1  /**
2   * This program demonstrates the measurable BankAccount and Country classes.
3  */
4  public class MeasureableDemo
5  {
6      public static void main(String[] args)
7      {
8          Measurable[] accounts = new Measurable[3];
9          accounts[0] = new BankAccount(0);
10         accounts[1] = new BankAccount(10000);
11         accounts[2] = new BankAccount(2000);
12
13         System.out.println("Average balance: "
14             + average(accounts));
15
16         Measurable[] countries = new Measurable[3];
17         countries[0] = new Country("Uruguay", 176220);
18         countries[1] = new Country("Thailand", 514000);
19         countries[2] = new Country("Belgium", 30510);
20
21         System.out.println("Average area: "
22             + average(countries));
23     }
}
```

62

MeasureableDemo.java (2)

```
25  /**
26   * Computes the average of the measures of the given objects.
27   * @param objs an array of Measurable objects
28   * @return the average of the measures
29  */
30  public static double average(Measurable[] objs)
31  {
32      if (objs.length == 0) { return 0; }
33      double sum = 0;
34      for (Measurable obj : objs)
35      {
36          sum = sum + obj.getMeasure();
37      }
38      return sum / objs.length;
39  }
40 }
```

Program Run

```
Average balance: 4000.0
Average area: 240243.33333333334
```

63

Interface nativa Comparable

- ❑ A biblioteca Java inclui um conjunto de interfaces, incluindo a interface Comparable
 - Requer a implementação de um método: `compareTo()`
 - É usada para comparar dois objetos
 - É implementada por muitos objetos na API Java
 - Podemos implementá-la nas nossas classes para usar certas ferramentas da API Java, como a ordenação
- ❑ É invocada sobre um objeto e é passado outro
 - Invocada sobre o objeto `a`, devolve os seguintes valores:
 - Negativo: `a` antecede `b`
 - Positivo: `a` sucede `b`
 - 0: `a` é igual a `b`

`a.compareTo(b);`

64

Parâmetro Comparable Type

- A interface Comparable usa um tipo especial de parâmetro que lhe permite lidar com qualquer tipo:

```
public interface Comparable<T>
{
    int compareTo(T other);
}
```

- O tipo <T> é um espaço reservado para um tipo de objeto
- A classe ArrayList usa a mesma técnica com o tipo envolto pelos símbolos < >

```
ArrayList<String> names = new ArrayList<String>();
```

65

Exemplo Comparable

- O método compareTo da classe BankAccount compara contas bancárias através do seu saldo (*balance*)
 - Recebe um parâmetro do tipo da sua própria classe (BankAccount)

```
public class BankAccount implements Comparable<BankAccount>
{
    ...
    public int compareTo(BankAccount other)
    {
        if (balance < other.getBalance()) { return -1; }
        if (balance > other.getBalance()) { return 1; }
        return 0;
    }
    ...
}
```

Os métodos da interface devem ser declarados como public

66

Utilização de compareTo para Ordenar

- ❑ O método `Arrays.sort` usa o método `compareTo` para ordenar os elementos do vetor
 - Uma vez que a classe `BankAccount` implementa a interface `Comparable`, podemos ordenar um vetor de contas bancárias com o método `Arrays.sort`:

```
BankAccount[] accounts = new BankAccount[3];
accounts[0] = new BankAccount(10000);
accounts[1] = new BankAccount(0);
accounts[2] = new BankAccount(2000);
Arrays.sort(accounts);
```

- O vetor está agora ordenado por ordem crescente de saldo

A implementação das interfaces da biblioteca Java permite-nos usar as potencialidades da biblioteca com as nossas classes

67

Erro Comum (6)



- ❑ Esquecer de declarar a implementação dos métodos como `public`
 - Os métodos numa interface não são declarados como `public`, porque são `public` por defeito
 - Contudo, os métodos numa classe não são `public` por defeito
 - O esquecimento da palavra reservada `public` na declaração de um método de uma interface é frequente:

```
public class BankAccount implements Measurable
{
    double getMeasure()    // Oops—should be public
    {
        return balance;
    }
    . . .
}
```

68

Declaração de Constantes na Interface

- As interfaces não podem ter variáveis de instância, mas é possível declarar constantes
- Quando se declara uma constante numa interface, podemos (e devemos) omitir as palavras reservadas `public static final`, porque todas as variáveis numa interface são automaticamente `public static final`

```
public interface SwingConstants
{
    int NORTH = 1;
    int NORTHEAST = 2;
    int EAST = 3;
    ...
}
```

69

Objetos Função(1)

- As interfaces funcionam bem SE os objetos que necessitam do serviço implementam a interface
- O propósito de um objeto função é executar um único método
 - Isto permite que uma classe que não implemente a interface possa usar os serviços da interface, criando um objeto função e usando o seu método que implementa a interface
- **Em primeiro lugar, vamos criar uma nova interface**
 - O método `measure` mede um objeto e devolve a sua medida. Usamos um parâmetro do tipo `Object`, o “mínimo denominador comum” de todas as classes Java, porque não queremos restringir o conjunto de classes que podem ser medidas

```
public interface Measurer
{
    double measure(Object anObject);
}
```

70

Objetos Função(2)

- Depois declaramos a classe que implementa a nova interface

```
public class StringMeasurer implements Measurer
{
    public double measure(Object obj)
    {
        String str = (String) obj; // Cast obj to String type
        return str.length();
    }
}

public interface Measurer
{
    double measure(Object anObject);
}
```

71

Objetos Função(3)

- Exemplo de utilização do Objeto Função
 - Instanciar um objeto da classe objeto função
 - Chamar o método que aceita um objeto deste tipo

```
String[] words = { "Mary", "had", "a", "little", "lamb" };
Measurer strMeas = new StringMeasurer();
double result = average(words, strMeas);

public static double average(Object[] objs, Measurer meas)
{
    if (objs.length == 0) { return 0; }
    double sum = 0;
    for (Object obj : objs)
    {
        sum = sum + meas.measure(obj);
    }
    return sum / objs.length;
}
```

72

Interface nativa Comparator

- ❑ Esta interface não tem que ser implementada pelas classes cujos objetos se pretendem comparar
- ❑ Uma outra classe pode implementar esta interface
- ❑ Usando esta interface, podemos implementar a ordenação baseada em diferentes atributos dos objetos a ordenar
 - Será necessário implementar uma classe responsável por ordenar de acordo com cada um dos critérios – estas classes implementam a interface Comparator e devem definir o método compare()

73

Interface nativa Comparator

```
int compare(Object o1, Object o2)
```

Este método compara os objetos o1 e o2 e devolve um inteiro:

- ❑ Positivo – o1 é maior que o2
- ❑ Zero – o1 é igual o2
- ❑ Negativo – o1 é menor que o2

74

Exemplo Comparator(1)

- Implementação da ordenação de contas bancárias, primeiro por id (int), depois por owner (String)

```
public class BankAccount {  
    private double balance;  
    private String owner;  
  
    public BankAccount(double bal, String name) {  
        balance = bal;  
        owner = name;  
    }  
    public double getBalance() {  
        return balance;  
    }  
    public String getOwner() {  
        return owner;  
    }  
    @Override  
    public String toString() {  
        return this.getClass().getName() + ":" +  
balance + ", " + owner;  
    }  
}
```

75

Exemplo Comparator(2)

- Implementação da classe BankSortbyIdComparator que implementa a interface Comparator, definindo o método compare() para comparação de objetos da classe BankAccount através do atributo owner

```
import java.util.Comparator;  
  
public class BankSortbyOwnerComparator implements  
    Comparator<BankAccount>{  
    @Override  
    public int compare(BankAccount account1, BankAccount account2) {  
        return account1.getOwner().compareTo(account2.getOwner());  
    }  
}
```

76

Exemplo Comparator(3)

Objeto da classe que implementa a interface Comparator

```
public class ComparatorMain {  
    public static void main(String[] args) {  
        BankAccount[] accounts = new BankAccount[3];  
        accounts[0] = new BankAccount(1000, "Jose");  
        accounts[1] = new BankAccount(2000, "Antonio");  
        accounts[2] = new BankAccount(500, "Manuel");  
  
        System.out.println("Before Sort : ");  
        for (BankAccount account : accounts) {  
            System.out.println(account);  
        }  
        Arrays.sort(accounts, new BankSortbyOwnerComparator());  
        System.out.println("After Sort by owner : ");  
        for (BankAccount account : accounts) {  
            System.out.println(account);  
        }  
        ...  
    }  
}
```

77

Exemplo Comparator(4)

Classe anónima, sem nome,
usada para criar um único objeto

```
public class ComparatorMain {  
    ...  
    Arrays.sort(accounts, new Comparator<BankAccount>() {  
        @Override  
        public int compare(BankAccount account1,  
                           BankAccount account2) {  
            return (account1.getBalance() <  
                   account2.getBalance() ) ? -1: (account1.getBalance() >  
                   account2.getBalance() ) ? 1:0 ;  
        }  
    };  
    System.out.println("After Sort by balance : ");  
    for (BankAccount account : accounts) {  
        System.out.println(account);  
    }  
}
```

78

Exemplo Comparator(5)

□ Saída do programa:

```
Before Sort :  
comparatormain.BankAccount:1000.0, Jose  
comparatormain.BankAccount:2000.0, Antonio  
comparatormain.BankAccount:500.0, Manuel  
After Sort by owner :  
comparatormain.BankAccount:2000.0, Antonio  
comparatormain.BankAccount:1000.0, Jose  
comparatormain.BankAccount:500.0, Manuel  
After Sort by balance :  
comparatormain.BankAccount:500.0, Manuel  
comparatormain.BankAccount:1000.0, Jose  
comparatormain.BankAccount:2000.0, Antonio
```

79

Comparator vs Comparable

	Comparable	Comparator
Lógica de ordenação	A lógica de ordenação deve estar na mesma classe cujos objetos são ordenados – ordenação natural	A lógica de ordenação encontra-se numa classe separada – permite criar diferentes ordenações baseadas em diferentes atributos
Implementação	A classe cujos objetos são ordenados deve implementar a interface	A classe cujos objetos são ordenados não implementa a interface – uma outra classe que implementa a interface é usada para definir o método de comparação
Método	<code>int compareTo(Object o1)</code> Este método compara o objeto <code>this</code> com <code>o1</code>	<code>int compare(Object o1, Object o2)</code> Este método compara os objetos <code>o1</code> e <code>o2</code>
Package	<code>java.lang.Comparable</code>	<code>java.util.Comparator</code>

80

Resumo: Herança

- ❑ Uma subclasse herda dados e comportamentos de uma superclasse
- ❑ É sempre possível usar um objeto da subclasse no lugar de um objeto da superclasse
- ❑ A subclasse herda todos os métodos que ela não reescreve
- ❑ Uma subclasse pode reescrever (override) um método da superclasse definindo uma nova implementação

81

Resumo: Reescrita de Métodos

- ❑ Um método reescrito pode estender ou substituir a funcionalidade do método da superclasse
- ❑ Utilizar a palavra reservada `super` para chamar um método da superclasse
- ❑ A não ser que seja especificado de outra forma, o construtor da subclasse chama o construtor da superclasse sem argumentos
- ❑ Para chamar um construtor da superclasse, usar a palavra reservada `super` no início do construtor da subclasse
- ❑ O construtor da subclasse pode passar argumentos para um construtor da superclasse, usando a palavra reservada `super`.

82

Resumo: Polimorfismo

- ❑ Uma referência a uma subclasse pode ser usada quando é esperada uma referência a uma superclasse
- ❑ Polimorfismo (“possuir múltiplas formas”) permite manipular objetos que partilham um conjunto de tarefas, mesmo que as tarefas sejam executadas de forma diferente
- ❑ Um método **abstract** é um método cuja implementação não é especificada
- ❑ Uma classe **abstract** é uma classe que não pode ser instanciada

83

Resumo: `toString` e `instanceof`

- ❑ Reescrever o método `toString` para gerar uma String que descreve o estado do objeto
- ❑ O método `equals` (herdado da classe `Object`) verifica se dois objetos são iguais – o método usa o operador `==` para determinar se os dois objetos são iguais
- ❑ Se sabemos que um objeto pertence a uma dada classe, fazer uma conversão de tipo (cast)
- ❑ O operador `instanceof` verifica se um objeto pertence a um tipo particular

84

Resumo: Interfaces

- ❑ O tipo `interface` contém os tipos de retorno, nomes e parâmetros de um conjunto de métodos
- ❑ Ao contrário de uma classe, uma `interface` não dispõe de implementação
- ❑ Utilizando um tipo interface como parâmetro de um método, este pode aceitar objetos de diferentes classes
- ❑ O termo `implements` indica quais são as interfaces que uma classe implementa
- ❑ Por exemplo, implementando a interface `Comparable` os objetos de uma classe podem ser comparados num método de ordenação

85