UPskill – JAVA + .NET

Programação Orientada a Objetos em Java – Herança e Polimorfismo

Adaptado de Donald W. Smith (TechNeTrain)

Objetivos

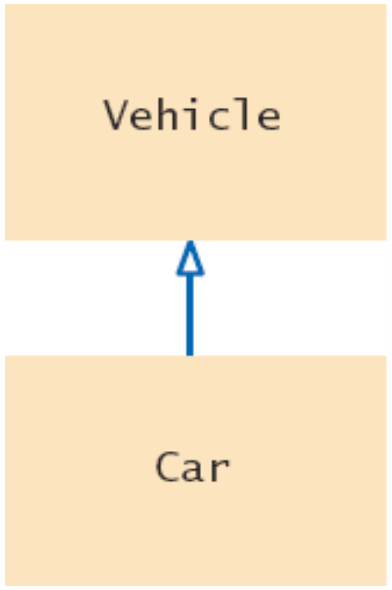
- Conceito de herança
- Implementação de subclasses que herdam e redefinem métodos da superclasse
- Compreender o conceito de polimorfismo
- Conhecer a superclasse comum `Object` e os seus métodos

Conteúdos

- Hierarquia de classes
- Implementação de subclasses
- Reescrita *(overriding)* de métodos
- Polimorfismo
- `Object`: A superclasse comum

Hierarquia de classes

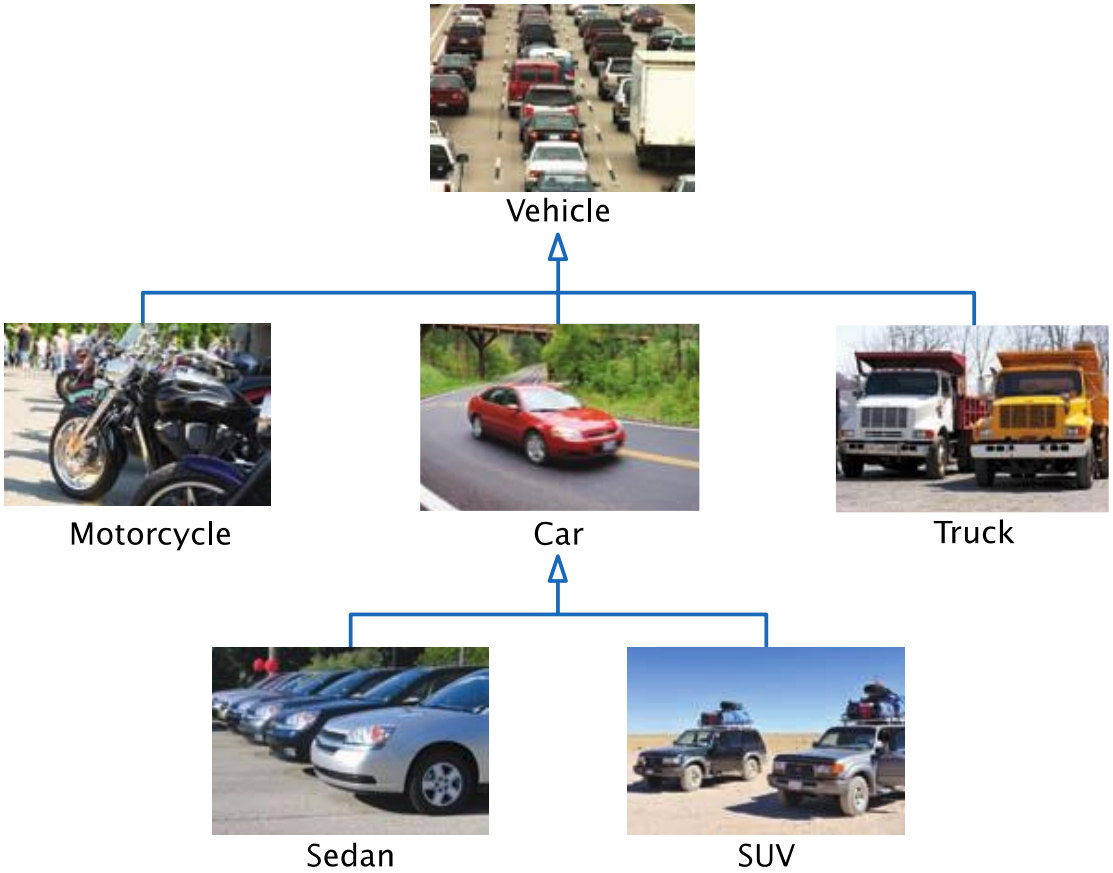
- Na Programação Orientada ao Objeto, herança é uma relação entre:
  - Uma *superclasse*
    (classe mais genérica)
  - Uma *subclasse*
    (classe mais especializada)



- A subclasse “herda” dados (variáveis) e comportamento (métodos) da superclasse

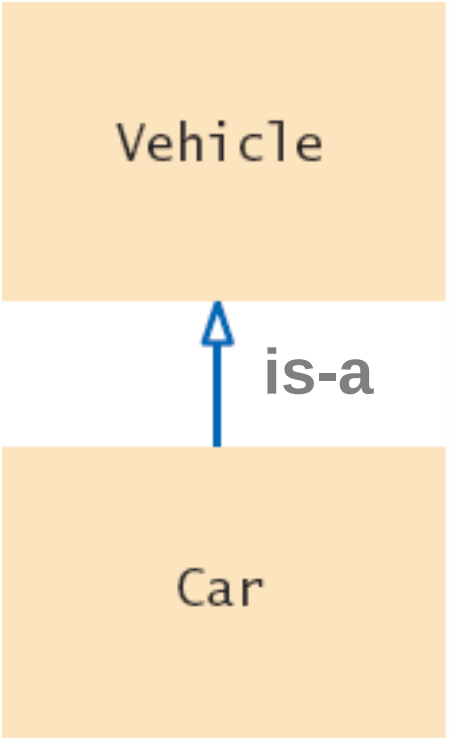
Uma Hierarquia da Classe Veículo

- Genérica
- Especializada
- Mais específica



Princípio da Substituição

- Como `Car` "***is-a***" `Vehicle`
  - `Car` partilha características comuns com `Vehicle`
  - É possível usar um objeto `Car` num programa que espera um objeto `Vehicle`



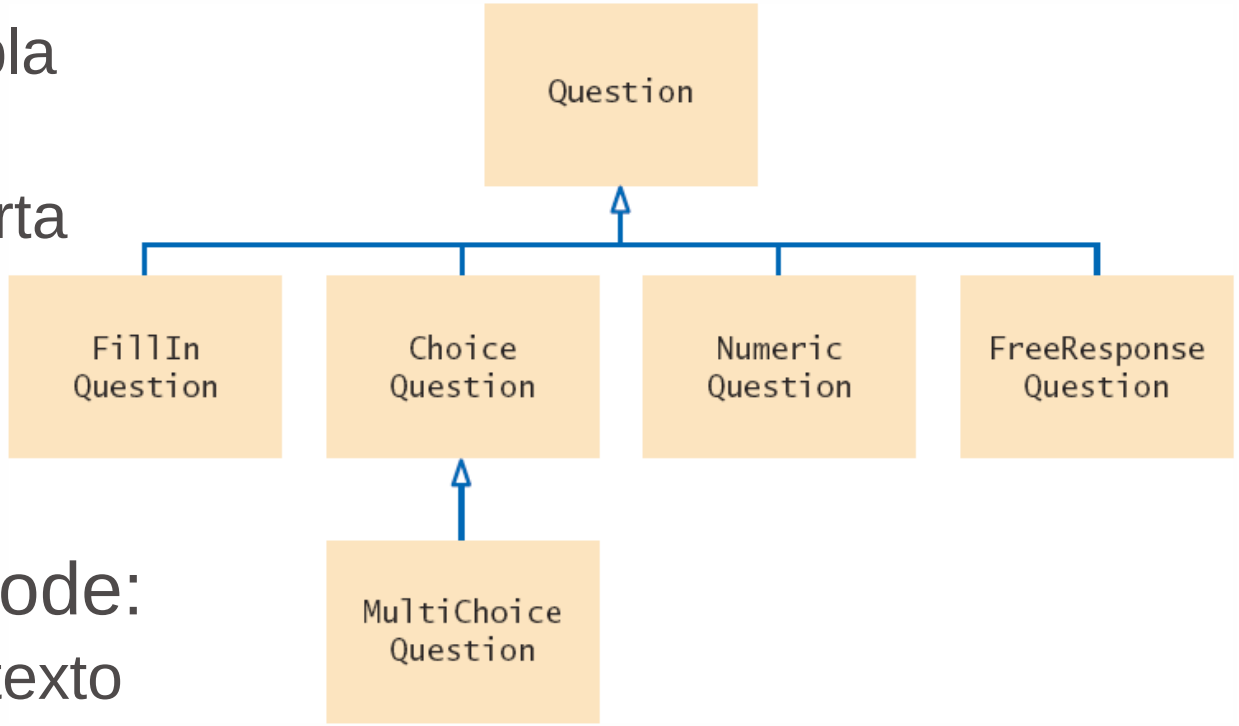

```
Car myCar = new Car(...);
processVehicle(myCar);
```

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.

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 se a resposta está correta

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       Constructs a question with empty question and answer.
11    */
12    public Question()
13    {
14       text = "";
15       answer = "";
16    }
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 superclasse

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

Question.java (2)

UPskill
Digital Skills & Jobs

```
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  }
```

QuestionDemo1.java

UPskill
Digital Skills & Jobs

```
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 }
```

Program Run

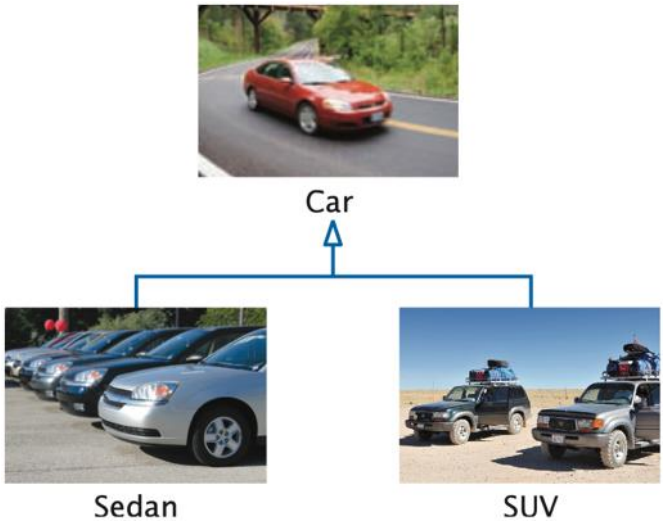
```
Who was the inventor of Java?
Your answer: James Gosling
true
```

Boas Práticas

- Usar uma única classe para variação de valores e 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



Implementação de Subclasses

- Consideremos a implementação da classe `ChoiceQuestion` para lidar com perguntas do tipo:

```
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
  - Para isso terá que existir um método que permita adicionar escolhas múltiplas
  - O método `display` mostrará estas escolhas logo a seguir à questão, numeradas apropriadamente

Herdando de uma Superclasse

- Subclasses herdam da superclasse:
  - Todos os métodos públicos
  - Todas as variáveis de instância
- A Subclasse pode
  - Acrescentar novas variáveis de instância
  - Acrescentar novos métodos
  - Alterar a implementação dos métodos herdados

A subclasse deve ser definida especificando as diferenças relativamente à superclasse

Reescrita (*Overriding*) de Métodos da Superclasse

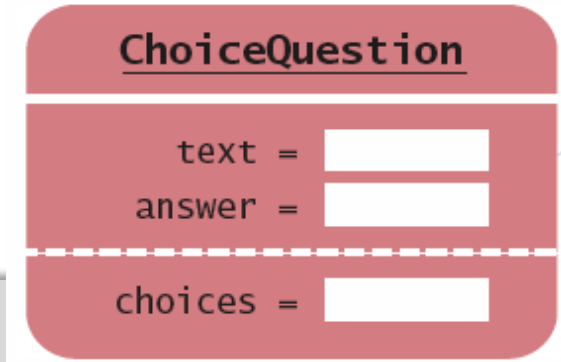
- É possível reutilizar métodos da classe `Question`?
  - Os métodos herdados têm exatamente o mesmo comportamento
  - Se for necessário alterar o seu comportamento (ou seja, a sua forma de funcionamento):
    - Codificar na subclasse um método mais especializado, com o mesmo nome do método da superclasse e com o mesmo conjunto de parâmetros
  - Esta necessidade conduz à reescrita (***overriding)*** 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

Planeamento da subclasse

UPskill
Digital Skills & Jobs

- Usar a palavra reservada `extends` para definir a relação de herança a partir de `Question`
  - As variáveis `text` e `answer` são herdadas
  - Acrescentar a nova variável de instância: `choices`



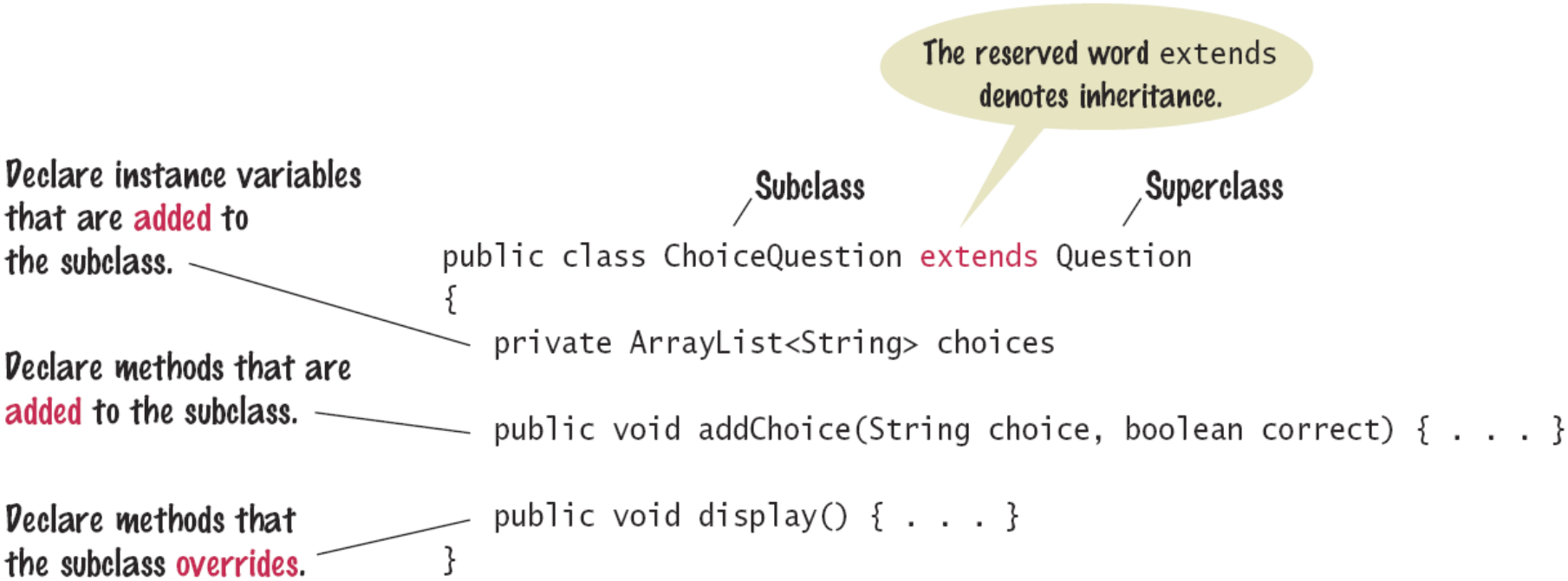

```
public class ChoiceQuestion extends Question
{
  // This instance variable is added to the subclass
  private ArrayList<String> choices;

  // 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() { . . . }
}
```

Declaração da Subclasse

- A subclasse herda da superclasse e estende (`extends`) a funcionalidade da superclasse



Implementação de addChoice

- O método recebe dois parâmetros
  - O texto para a escolha
  - Um booleano que indica se se trata de uma escolha correta
- O método adiciona o texto como uma escolha e no caso de se tratar de uma escolha correta, adiciona o número da escolha à resposta, chamando 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()`

Erro Comum (1)

UPskill
Digital Skills & Jobs

- 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;
  . . .
```

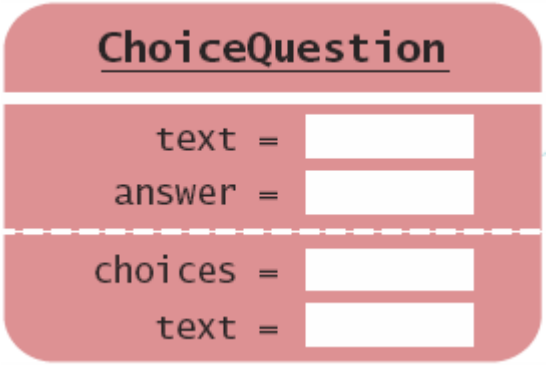
```
public class ChoiceQuestion extends Question
{
  . . .
  text = questionText;   // Complier Error!
```

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` (da subclasse)
- O método display acede à primeira variável (da superclasse)



Erro Comum (2)

UPskill
Digital Skills & Jobs

- Confundir *Super* e *Sub*classes
  - 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`

Reescrita (*Overriding*) de Métodos

- A classe `ChoiceQuestion` necessita de um método `display` que reescreva 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`
    - Mostra a variável de instância String text
    - Mostra a lista local de escolhas (*choices*)

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 uma variável de instância privada da superclasse
  - Como obter acesso à variável para mostrar a questão?
  - Chamar o método `display` da superclasse `Question`!
    - A partir da subclasse, adicionamos um prefixo ao nome do método:

      `super.`

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

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("*7", 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    }
```

Criação de dois objetos da classe `ChoiceQuestion`

Invocação de `presentQuestion` (ver próximo slide)

QuestionDemo2.java (2)

UPskill
Digital Skills & Jobs

```
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   }
```

Invocação do método display de ChoiceQuestion (subclasse)

ChoiceQuestion.java (1)

UPskill
Digital Skills & Jobs

```
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       Constructs a ch
12    */
13    public ChoiceQues
14    {
15       choices = new
16    }
```

Herda da classe `Question`

```
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    }
```

Novo método `addChoice`

ChoiceQuestion.java (2)

UPskill
Digital Skills & Jobs

```
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`

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

Erro Comum (3)

UPskill
Digital Skills & Jobs

- 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 reescrita mas alteramos os parâmetros, estaremos a fazer **overloading** do método herdado e não a reescrevê-lo (**overriding**) – passaremos a ter 2 métodos
    - Ex: Método `display(PrintStream out)` de `ChoiceQuestion`

Erro Comum (4)

UPskill
Digital Skills & Jobs

- Esquecer o uso de `super` nas situações em que é imperativo usá-lo
  - 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;
   }
}
```

Chamada do Construtor da Superclasse

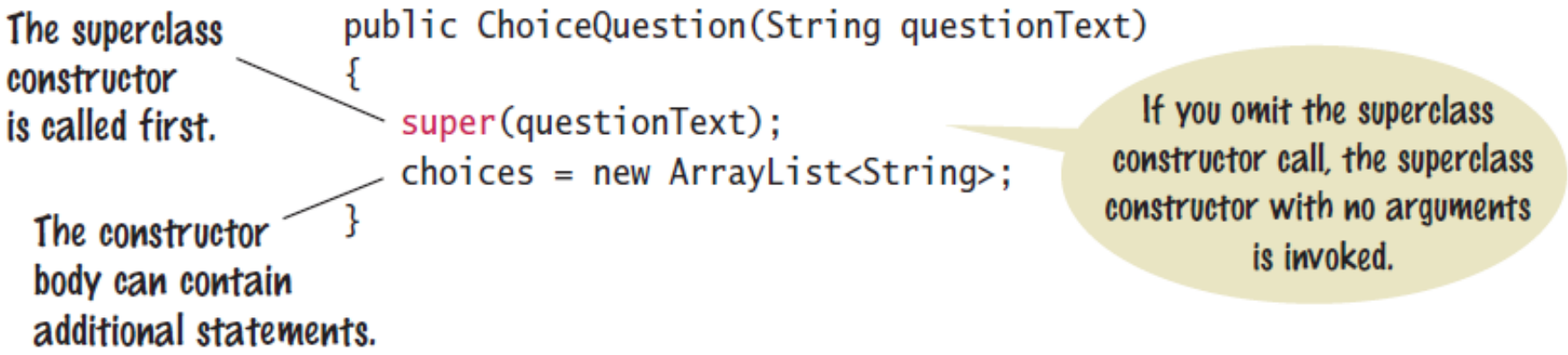
- Quando uma subclasse é instanciada, é invocado o construtor da superclasse sem argumentos
- Se preferirmos chamar um construtor mais específico, podemos invocá-lo substituindo o nome da superclasse pela palavra reservada `super` seguida por `()`:

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

- Tem de ser a primeira instrução no construtor

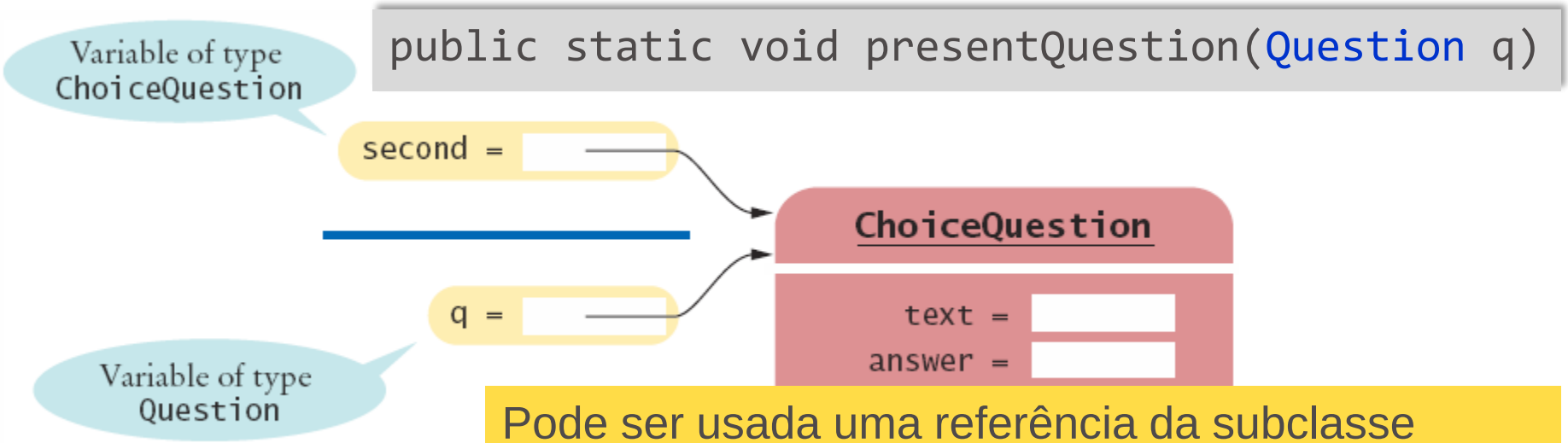
Construtor com Inicialização da Superclasse

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



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?**



Pode ser usada uma referência da subclasse quando é esperada uma referência da superclass

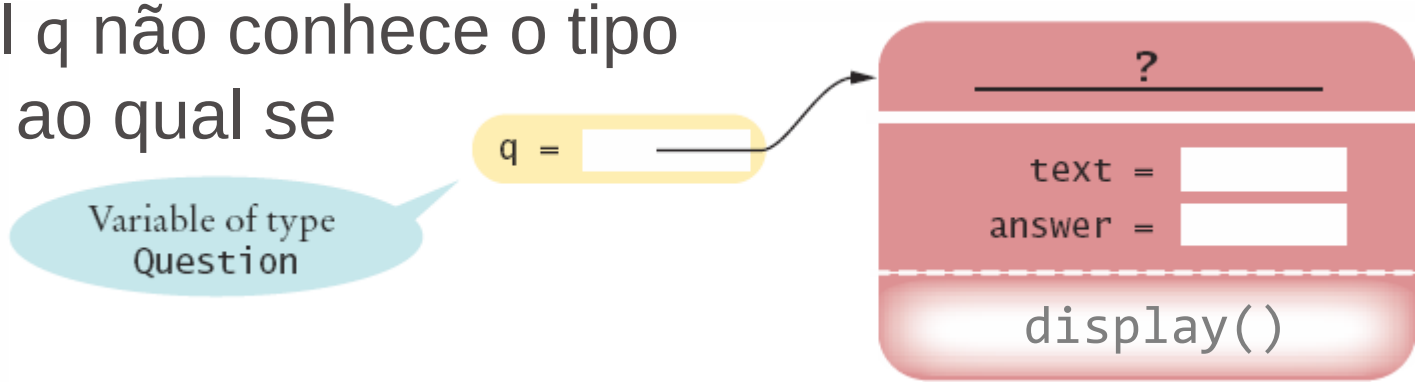
Que versão do método `display` é chamado?

- `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



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 caraterística 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

QuestionDemo3.java (1)

```
1  import java.util.Scanner;
2
3  /**
4     This program shows a simple quiz with two question types.
5  */
6  public class QuestionDemo3
7  {
8     public static void main(String[] args)
9     {
10       Question first = new Question();
11       first.setText("Who was the inventor of Java?");
12       first.setAnswer("James Gosling");
13
14       ChoiceQuestion second = new ChoiceQuestion();
15       second.setText("In which country was the inventor of Java born?");
16       second.addChoice("Australia", false);
17       second.addChoice("Canada", true);
18       second.addChoice("Denmark", false);
19       second.addChoice("United States", false);
20
21       presentQuestion(first);
22       presentQuestion(second)
23    }
24
```

Criação de um objeto da classe `Question`

Criação de um objeto da classe `ChoiceQuestion` e invocação do novo método `addChoice`

Invocação de `presentQuestion` (ver no próximo slide) passando ambos os tipos de objetos

QuestionDemo3.java (2)

UPskill
Digital Skills & Jobs

```
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   }
```

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

Referências Abstratas

- Uma classe que pode ser instanciada é denominada classe concreta
- Não é possível instanciar uma classe 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

Métodos e Classes Final

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

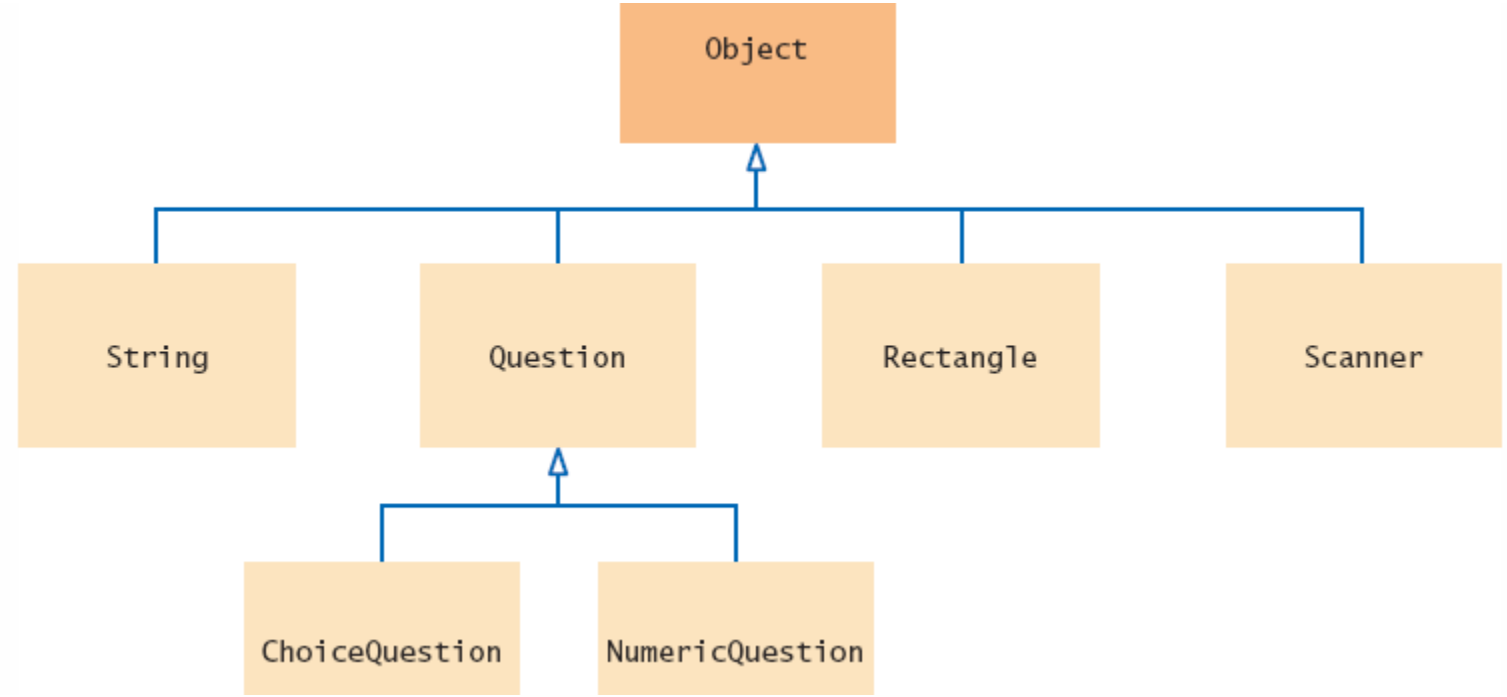
```
public final class String { ... }
```

- Exemplo de um método que não pode ser reescrito:

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

Object: a Superclasse Raiz

- Em Java, todas as classes estendem a classe `Object`



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

Definição de um Método toString

- 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`

Reescrita do Método toString

- 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

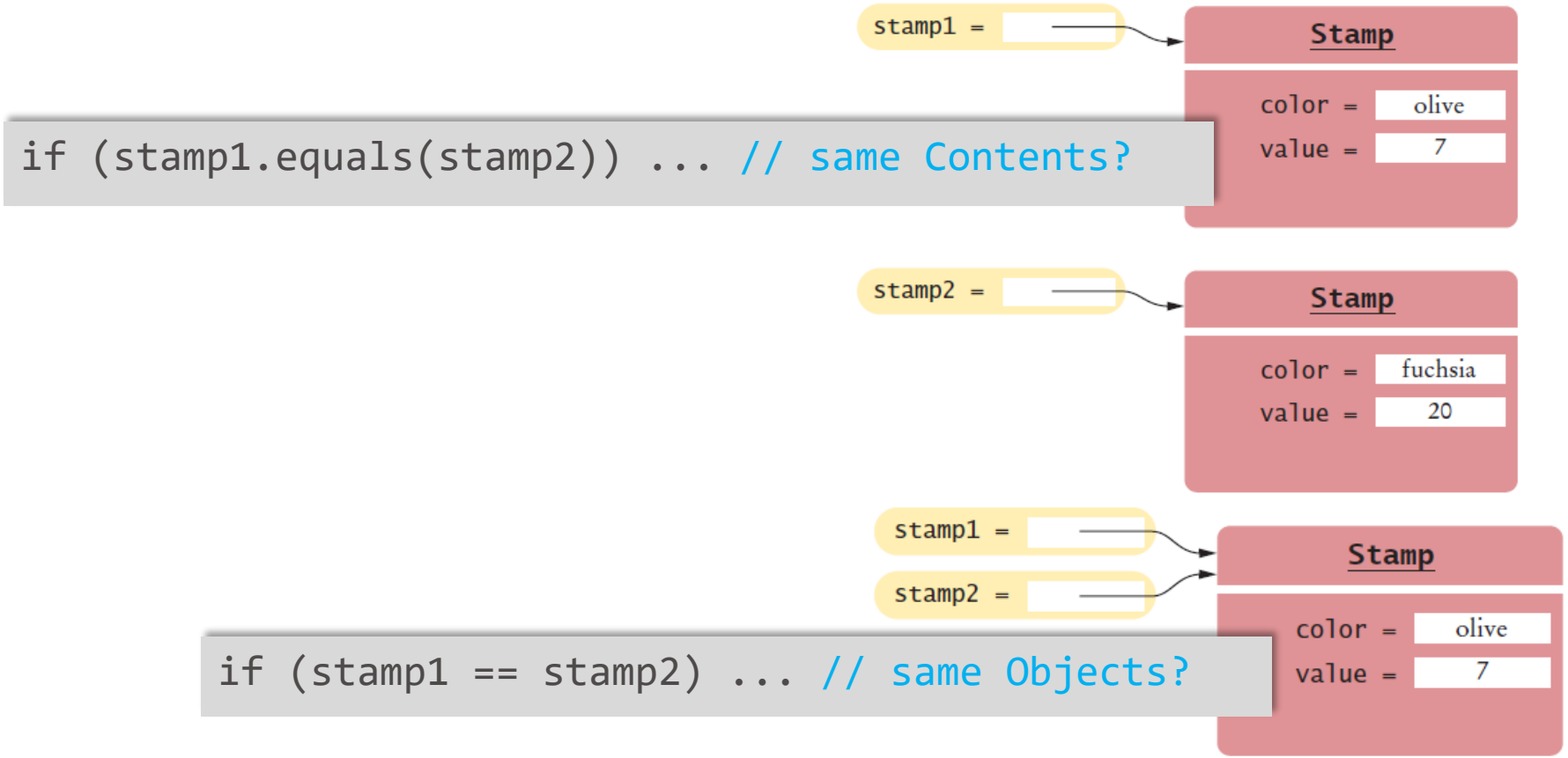
- Podemos pretender incluir informação sobre o objeto

```
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

Reescrita do Método equals

- 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).



Reescrita do Método equals

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

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

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

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

Fazer *cast* do parâmetro da classe `Stamp`

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;
```

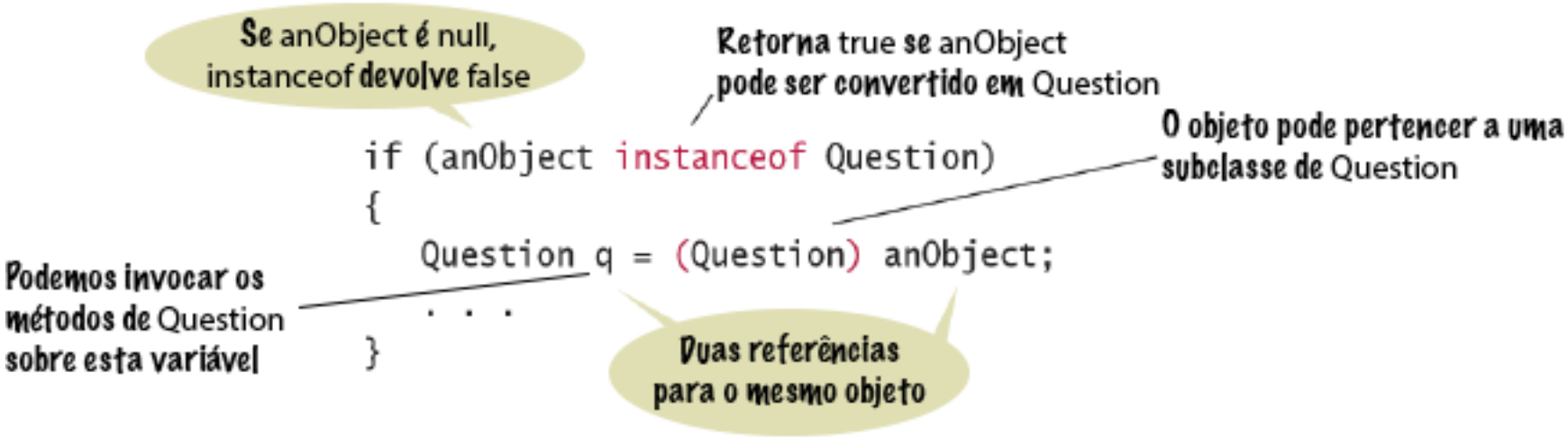
- Para garantir que se trata de um objeto do tipo `Question`, é possível verificá-lo com o operador `instanceof`:

`instanceof` devolve um boolean

```
if (obj instanceof Question) {
  Question q = (Question) obj;
}
```

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 `Object` para um tipo de classe mais específica



Erro Comum (5)

UPskill
Digital Skills & Jobs

- 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 testes terão que ser revistos
    - Por exemplo ao adicionarmos a classe `NumericQuestion`
  - Deixemos o polimorfismo selecionar o método correto:
    - Declarar um método `executarTarefa` na superclasse
    - Reescrevê-lo nas subclasses

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 superclasse mostre as variáveis de instância privadas

Resumo: Herança

- Uma subclasse herda dados e comportamento 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

Resumo: Reescrita de Métodos

UPskill
Digital Skills & Jobs

- 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**.

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

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