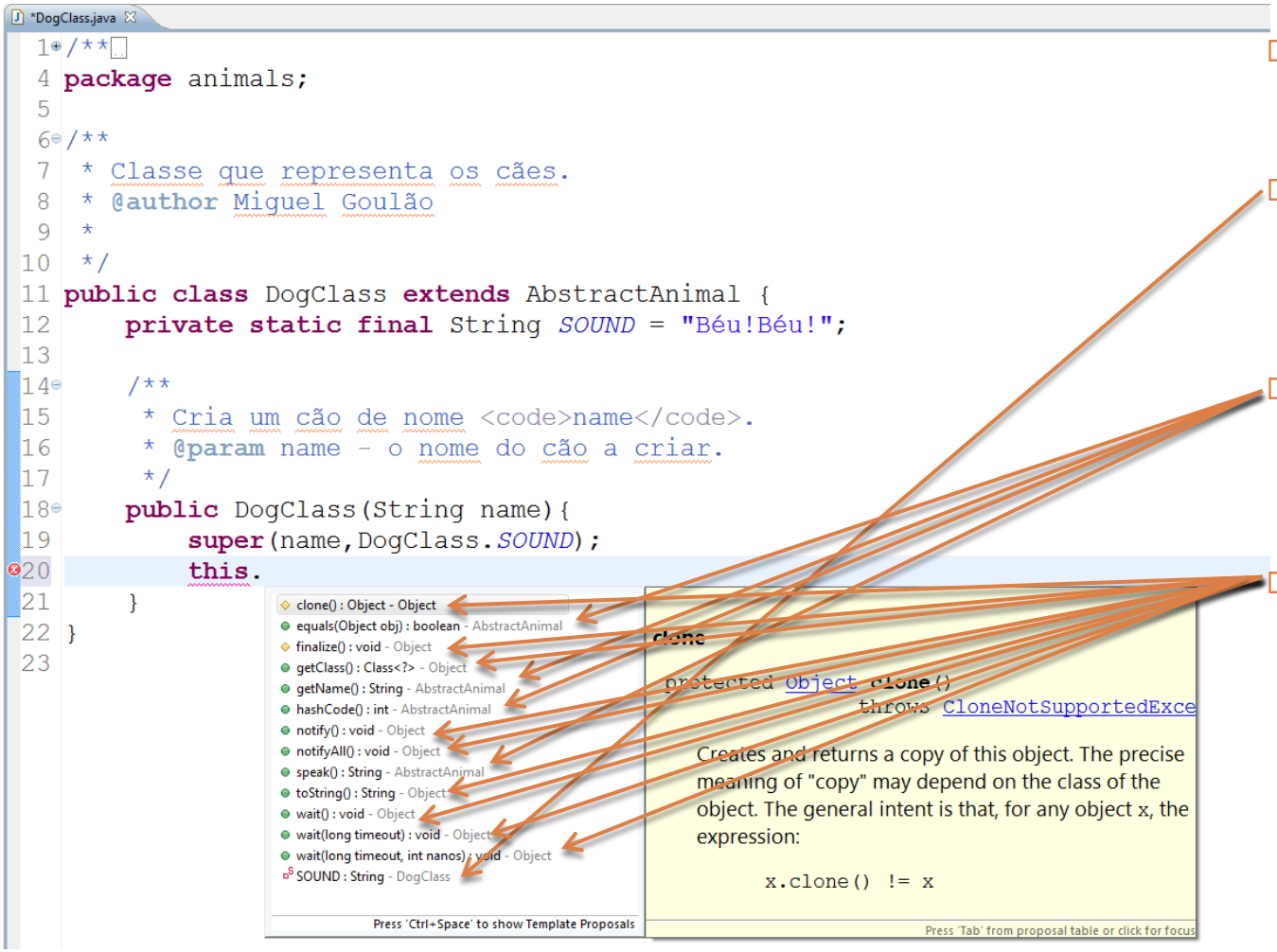
PROGRAMAÇÃO ORIENTADA PELOS OBJECTOS

Aula 10

Extensibilidade e a classe Object

A classe Object

Já reparou bem na lista de membros que o Eclipse constuma apresentar a seguir ao `this`.?



- De onde surgiram estes membros?
- Alguns são específicos da classe `DogClass`
- Outros são herdados de `AbstractAnimal`
- Outros, da classe `Object`

A classe `Object`

- É a super-classe de **TODAS** as classes em Java
  - Todas as classes, existentes e a criar, são sub-classes de `Object`
  - `Object` não herda de nenhuma classe
  - Quando não se declara que uma classe é sub-classe de outra, automaticamente, ela fica sub-classe directa de `Object`
  - As nossas classes herdam membros da classe `Object`

Membros herdados de `Object`

- `public boolean equals(Object obj)`
  - Compara dois objectos
- `protected Object clone() throws CloneNotSupportedException`
  - Cria e devolve uma cópia do objecto
- `protected void finalize() throws Throwable`
  - Operação invocada pelo garbage collector sobre um objecto, quando o mecanismo de garbage collection determina que já não existem mais referências para o objecto
- `public final Class getClass()`
  - Devolve a classe concreta de um objecto, ou seja, a sua classe em tempo de execução
- `public int hashCode()`
  - Devolve o valor do hash code de um objecto
- `public String toString()`
  - Retorna uma **`String`** representando o objecto

Membros herdados de `Object`

- Os métodos **`notify`**, **`notifyAll`**, e **`wait`** são usados na sincronização de **`threads`** em programas concorrentes
  - `public final void notify()`
  - `public final void notifyAll()`
  - `public final void wait()`
  - `public final void wait(long timeout)`
  - `public final void wait(long timeout, int nanos)`

O método equals

O método `equals`: propriedades

- O método `equals` implementa uma relação de equivalência de referências não nulas a objectos:
  - **Reflexivo**: para cada referência não nula ao objecto `x`, `x.equals(x)` deve retornar `true`
  - **Simétrico**: para referências não nulas `x` e `y`, `x.equals(y)` retorna `true` se e só se `y.equals(x)` retorna `true`
  - **Transitivo**: para referências não nulas x, y e z, se x.equals(z) retorna true e y.equals(z) retorna true, então x.equals(y) também tem de retornar true
  - **Consistente**: para quaisquer referências não-nulas x e y, múltiplas invocações de x.equals(y) retornam consistentemente true, ou consistentemente false, desde que nada se altere nos valores referenciados x e y
  - Para qualquer referência não nula x, x.equals(null) retorna sempre false

O método equals na classe `Object`

- Na classe **`Object`**, o método equals retorna a forma de equivalência mais discriminatória possível:
  - Para quaisquer referências não nulas **`x`** e **`y`**, o método **`equals`** retorna **`true`** se e só se **`x`** e **`y`** forem referências para o mesmo objecto
  - Note que, nesse caso, **`x==y`** também é avaliado como **`true`**
- Sintaxe:

```
public boolean equals(Object obj)
```

- Parâmetros:
  - **`obj`** – a referência do objecto que vamos comparar a **`this`**
- Retorno:
  - **`true`**, se o objecto referenciado por **`this`** for o mesmo que o objecto referenciado por **`obj`**, ou **`false`**, caso contrário

Implementação do método `equals` (exemplo na classe `AbstractAnimal`)


```
public abstract class AbstractAnimal implements Animal {
  //...
  public boolean equals(Object obj) {
    return (name.equals(obj.name)
        && (sound.equals(obj.sound));
}
```

- Funciona?
  - Depende...
    - E se tivermos uma vaca e um papagaio, ambos chamados "Mimosa" e dizendo "Muuuuuh!"?
    - As variáveis de instância são objectos
      - O argumento é um objecto
      - Se nenhum destes for `null`, tudo bem
        - Mas, e se algum for `null`?

O método `equals` (exemplo na classe `AbstractAnimal`)


```
public boolean equals(Object obj) {
  if (this == obj)
    return true;
  if (obj == null)
    return false;
  if (!(obj instanceof AbstractAnimal))
    return false;
  AbstractAnimal other = (AbstractAnimal) obj;
  if (name == null) {
    if (other.name != null)
      return false;
  } else if (!name.equals(other.name))
    return false;
  if (sound == null) {
    if (other.sound != null)
      return false;
  } else if (!sound.equals(other.sound))
    return false;
  return true;
}
```

- Método redefinido de **`Object`**, na classe **`AbstractAnimal`**
- Se **`obj`** referencia a mesma memória que **`this`**, são o mesmo objecto
- Um objecto nunca é igual a **`null`**
- Se **`obj`** não for do mesmo tipo (ou de um subtipo deste tipo), não são iguais
- As variáveis de instância têm de ser iguais
- Os vários testes a **`null`** evitam que o programa possa terminar com erro, caso alguma das variáveis esteja a **`null`**
- Vale a pena redefinir esta operação nas sub-classes?

O método clone

O método `clone`


```
protected Object clone() throws CloneNotSupportedException
```

- Cria e retorna uma cópia do objecto **`this`**
- O significado exacto de "cópia" depende da classe do objecto mas, **em geral**, quando clonamos um objecto referenciado por **`x`**:
  - **`x.clone() != x`**
  - **`x.clone().equals(x)`**
  - **`x.clone().getClass() == x.getClass()`**
  - Por convenção, o objecto clonado deve ser obtido invocando **`super.clone`**
    - Se uma classe e todas as suas super-classes (com excepção de **`Object`**) seguirem esta convenção, é garantido que **`x.clone().getClass() == x.getClass()`**

O método `clone`


```
protected Object clone() throws CloneNotSupportedException
```

- Por convenção, o objecto a retornar por este método deve ser independente do original (que está a ser clonado)
  - Isto pode implicar modificar um ou mais campos do objecto retornado por `super.clone`, antes de o retornar
  - Normalmente, é necessário clonar os objectos internos mutáveis
  - Não é necessário ter este cuidado adicional com tipos elementares, ou com referências a objectos imutáveis
- **Apenas se podem clonar objectos de classes que implementem a interface `Cloneable`**

O método `clone` na classe `Object`

- O método `clone` da classe `Object` realiza uma operação de clonagem específica
- Se uma classe de um determinado objecto não implementar a interface `Cloneable`, o método lança a excepção `CloneNotSupportedException`
  - Por omissão, todos os arrays suportam a interface `Cloneable`

Os perigos da clonagem...

- Se a classe implementa a interface **`Cloneable`**, o método clone cria uma nova instância da classe do objecto e inicializa os membros de dados, fazendo uma cópia dos campos do objecto original
  - Esta cópia é "*superficial*", como acontece quando fazemos uma atribuição simples
    - O conteúdo dos membros de dados não é clonado!
    - Apenas é criada uma nova referência para objectos já existentes
- Perigo:
  - A clonagem pode ter efeitos secundários indesejados, devido a esta cópia superficial
    - O clone e o objecto original podem ficar a referenciar um mesmo objecto mutável – se o objecto mudar o seu estado, isso vai afectar quer o objecto original, quer o clone! Se um dos objectos destruir o objecto referenciado e depois disso o seu clone tentar usar esse objecto, isso provocará um erro
  - É por este motivo que o método clone() de Object é declarado como protegido

O método `clone` na classe `Object`

- A classe `Object` não implementa a interface `Cloneable`; invocar o método clone num objecto da classe `Object` resulta no lançamento de uma excepção
- **Retorno:**
  - Um clone da instância `this`
- **Excepções:**
  - `CloneNotSupportedException`
    - Se a classe do objecto não implementar a interface `Cloneable`, esta excepção é lançada
    - Uma classe que redifina o método `clone` também pode lançar esta excepção, se por qualquer motivo uma instância não puder ser clonada

Exemplo de clonagem


```
public interface Animal {
  /**
   * Devolve o nome do animal
   * @return nome do animal
   */
  String getName();

  /**
   * Devolve o "falar" do animal
   * @return onomatopeia da voz do animal
   */
  String speak();

  /**
   * Altera o nome do animal
   * @param newName - o novo nome do animal
   */
  void rename(String newName);
}
```

- O método **`rename`** é novo neste exemplo, todos os restantes já existiam

Exemplo de clonagem


```
public abstract class AbstractAnimal implements Animal, Cloneable {
  // Tudo igual ao exemplo da aula anterior
  // Mas agora acrescentamos 2 novos métodos:

  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }

  public void rename(String newName) {
    name = newName;
  }
}
```

- O método **clone** é necessário para implementar a interface **Cloneable**
- O método **rename** foi acrescentado à interface **Animal**

Snoopy ou Snuppy?


```
public class CloneTester {
  public static void main(String[] args) {
    Animal snoopy = new DogClass("Snoopy");
    try {
      Animal snuppy = (Animal) ((AbstractAnimal) snoopy).clone();
      System.out.println(snoopy.getName());
      System.out.println(snuppy.getName());
      snuppy.rename("Lassie");
      System.out.println(snoopy.getName());
      System.out.println(snuppy.getName());
    } catch (CloneNotSupportedException e) {
      e.printStackTrace();
    }
  }
}
```

Snoopy
Snoopy
Snoopy
Lassie

- Tudo ok, as strings são objectos **imutáveis**
  - ou seja, são constantes

Mas, e se os objectos forem mutáveis?

- Imagine que queremos poder definir um dono aos animais de estimação
  - Comecemos por criar uma interface **`Person`**, que permita
    - Obter o nome da pessoa
    - Atribuir à pessoa um animal de estimação
    - Obter o animal de estimação dessa pessoa
    - Actualizar o nome da pessoa
      - Pode parecer estranho a princípio, mas uma pessoa pode mudar de nome, por exemplo, quando se casa... ☺

Ao contrário das Strings, os objectos que implementam pessoas são mutáveis

A interface Person


```
public interface Person {
   String getName();
   void setPet(Pet p);
   Pet getPet();
   void rename(String newName);
}
```

A classe `PersonClass`


```
public class PersonClass implements Person, Cloneable {
  private String name;
  private Pet pet;

  public PersonClass(String name) {
    this.name = name;
    this.pet = null;
  }
  public void setPet(Pet p) { this.pet = p; }
  public Pet getPet() { return pet; }
  public String getName() { return name; }
  public void rename(String newName) { name = newName; }
  public Object clone() throws CloneNotSupportedException {
    return super.clone();
  }
}
```

A interface Pet


```
public interface Pet {
  public void setOwner(Person p);
  public Person getOwner();
}
```

A classe DogClass


```
public class DogClass extends AbstractAnimal implements Pet {
  private static final String SOUND = "Béu!Béu!";
  private Person owner;
  public DogClass(String name){
    super(name, DogClass.SOUND);
    owner = null;
  }
  public void setOwner(Person p) { owner = p; }
  public Person getOwner() { return owner; }
  public Object clone() throws CloneNotSupportedException {
    DogClass cloned = (DogClass) super.clone();
    /* Se não fizermos mais nada, o cão clonado vai-se referir ao
       mesmo objecto que representa o dono do cão original. */
    return cloned;
  }
}
```


Isto introduz uma situação de efeitos secundários potenciais!

Efeitos secundários sobre variáveis mutáveis clonadas


```
public static void main(String[] args) {
  Animal snoopy = new DogClass("Snoopy");
  Person charlie = new PersonClass("Charlie Brown");
  ((DogClass) snoopy).setOwner(charlie);
  charlie.setPet((Pet) snoopy);
  try {
    Animal snuppy = (Animal) ((AbstractAnimal) snoopy).clone();
    System.out.println(snoopy.getName());
    System.out.println(((Pet) snoopy).getOwner().getName());
    System.out.println(snuppy.getName());
    System.out.println(((Pet) snuppy).getOwner().getName());
    snuppy.rename("Lassie");
    ((Pet)snoopy).getOwner().rename("Mr. Charlie Brown");
    System.out.println(snoopy.getName());
    System.out.println(((Pet) snoopy).getOwner().getName());
    System.out.println(snuppy.getName());
    System.out.println(((Pet) snuppy).getOwner().getName());
  } catch (CloneNotSupportedException e) {
    e.printStackTrace();
  }
}
```

Snoopy
Charlie Brown
Snoopy
Charlie Brown
Snoopy
Mr. Charlie Brown
Lassie
Mr. Charlie Brown

Repare que introduzimos um efeito secundário, ao renomear o dono de Snoopy

A classe DogClass


```
public class DogClass extends AbstractAnimal implements Pet {
  private static final String SOUND = "Béu!Béu!";
  private Person owner;
  public DogClass(String name){
    super(name, DogClass.SOUND);
    owner = null;
  }
  public void setOwner(Person p) { owner = p; }
  public Person getOwner() { return owner; }
  public Object clone() throws CloneNotSupportedException {
    DogClass cloned = (DogClass) super.clone();
    if (this.owner != null)
       cloned.owner = (Person) ((PersonClass)(this.owner)).clone();
    return cloned;
  }
}
```

Com esta alteração removemos os efeitos secundários!

Efeitos secundários sobre variáveis mutáveis clonadas


```
public static void main(String[] args) {
  Animal snoopy = new DogClass("Snoopy");
  Person charlie = new PersonClass("Charlie Brown");
  ((DogClass) snoopy).setOwner(charlie);
  charlie.setPet((Pet) snoopy);
  try {
    Animal snuppy = (Animal) ((AbstractAnimal) snoopy).clone();
    System.out.println(snoopy.getName());
    System.out.println(((Pet) snoopy).getOwner().getName());
    System.out.println(snuppy.getName());
    System.out.println(((Pet) snuppy).getOwner().getName());
    snuppy.rename("Lassie");
    ((Pet)snoopy).getOwner().rename("Mr. Charlie Brown");
    System.out.println(snoopy.getName());
    System.out.println(((Pet) snoopy).getOwner().getName());
    System.out.println(snuppy.getName());
    System.out.println(((Pet) snuppy).getOwner().getName());
  } catch (CloneNotSupportedException e) {
    e.printStackTrace();
  }
}
```

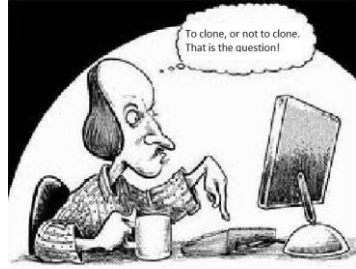
```
Snoopy
Charlie Brown
Snoopy
Charlie Brown
Snoopy
Mr. Charlie Brown
Lassie
Charlie Brown
```

Com a correcção feita, o efeito secundário desapareceu

Resumindo, para clonar...

- Temos de implementar a interface **`Cloneable`**
- Temos de tornar o método **`Object clone()`** público, redefinindo o método protegido herdado
- Seguidamente, invocamos o método **`clone`** definido na classe **`Object`**, de modo a criar uma cópia superficial do objecto
- Depois, devemos fazer uma cópia profunda de todos os objectos mutáveis, se a cópia superficial (da referência para esses objectos) não for suficiente

Clonar ou não clonar?



- Quando o estado de um objecto a clonar é mutável, acabamos com 2 objectos que não são independentes
  - As cópias superficiais provocaram já inúmeros bugs, no passado!
  - Não há garantia nenhuma de que a clonagem respeite invariantes estabelecidos pelo construtor
  - Mais vale criar um **construtor por cópia**, em geral
    - A cópia pode até ter uma representação diferente do original
- Vários gurus do Java recomendam que o mecanismo de cloning não seja usado, em geral, por trazer mais problemas que benefícios

O método toString

O método `toString`

- O método **`toString`** da classe **`Object`** devolve uma **`String`** representando o objecto
- A representação como **`String`** de um objecto depende completamente do objecto, pelo que tipicamente as classes devem redefinir o método **`toString`**, em vez de usar a definição de **`Object`**
- Podemos usar a operação **`toString()`** juntamente com o **`System.out.println()`**, para escrever na consola, de modo prático, informação sobre o objecto

Método `toString` (exemplo na classe `AbstractAnimal`)

- Exemplo:

```
public class AbstractAnimalClass implements Animal {
   ...
   public String toString() {
     return this.getName() + ":" + this.speak();
   }
}

public class Main() {
   public static void main(String[] args) {
     Animal garfield = new CatClass("Garfield");
     System.out.println(garfield);
   }
}
```

- Retorno:

```
Garfield:Miau!
```