

Programação Orientada Pelos Objectos

Extensibilidade e a classe Object

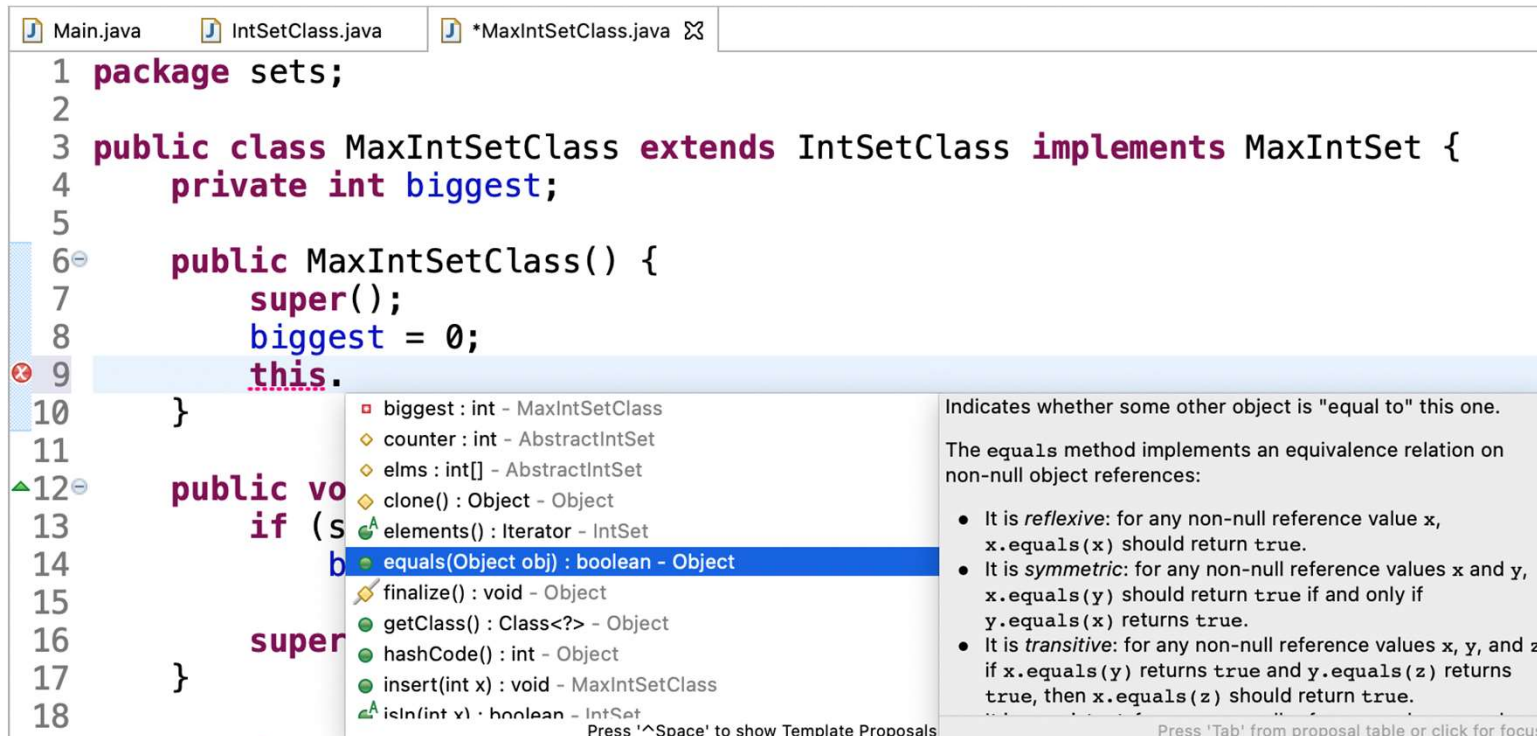


2

A classe Object

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

3



```
1 package sets;
2
3 public class MaxIntSetClass extends IntSetClass implements MaxIntSet {
4     private int biggest;
5
6     public MaxIntSetClass() {
7         super();
8         biggest = 0;
9         this.
10    }
11
12    public void insert(int x) {
13        if (x > biggest) {
14            biggest = x;
15        }
16    }
17
18    super.insert(x);
19 }
```

Code completion popup for `this.`:

- biggest : int - MaxIntSetClass
- counter : int - AbstractIntSet
- elms : int[] - AbstractIntSet
- clone() : Object - Object
- elements() : Iterator - IntSet
- equals(Object obj) : boolean - Object**
- finalize() : void - Object
- getClass() : Class<?> - Object
- hashCode() : int - Object
- insert(int x) : void - MaxIntSetClass
- isIn(int x) : boolean - IntSet

Indicates whether some other object is "equal to" this one.

The equals method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value x, x.equals(x) should return true.
- It is *symmetric*: for any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
- It is *transitive*: for any non-null reference values x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.

Press 'Tab' from proposal table or click for focus

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

A classe `Object`

4

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

Membros herdados de Object

5

- **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 colector de lixo (*garbage collector*) sobre um objecto, quando máquina virtual 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`

6

- 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)`

7

Identidade dos objectos e o método `equals()`

Identidade *versus* igualdade

8

- A **identidade** é uma propriedade fundamental da Programação Orientada por Objectos
 - Permite aos objectos referenciarem-se uns aos outros
 - Permite a um objecto ser referenciado por múltiplos outros objectos
 - Não depende do tipo estático da referência ou variável
 - Testa-se comparando duas referências com ==

Identidade *versus* igualdade

9

- A **identidade** é uma propriedade fundamental da Programação Orientada por Objectos
 - Permite aos objectos referenciarem-se uns aos outros
 - Permite a um objecto ser referenciado por múltiplos outros objectos
 - Não depende do tipo estático da referência ou variável
 - Testa-se comparando duas referências com ==

```
void doSomething(Animal a, GermanSheppherd b) {  
    if (a == b) System.out.println("São o mesmo objecto!");  
}
```

```
void doSomething(CatClass a, CatClass b) {  
    if (a == b) System.out.println("São o mesmo objecto!");  
}
```

Identidade *versus* igualdade

10

- A **identidade** é uma propriedade fundamental da Programação Orientada por Objectos
 - Permite aos objectos referenciarem-se uns aos outros
 - Permite a um objecto ser referenciado por múltiplos outros objectos
 - Não depende do tipo estático da referência ou variável
 - Testa-se comparando duas referências com ==

```
void doSomething(Animal a, GermanSheppherd b) {  
    if (a == b) System.out.println("São o mesmo objecto!");  
}
```

```
void doSomething(CatClass a, CatClass b) {  
    if (a == b)
```

A identidade não depende do tipo da referência
que usamos para ter acesso ao objecto

Identidade *versus* igualdade

11

- A **igualdade** não é o mesmo que identidade
 - Difere sempre que admitimos que dois objectos distintos possam ser considerados iguais em certas circunstâncias
 - Usamos o método **equals** para determinar que circunstâncias são essas
 - Frequentemente, o critério para a igualdade é a igualdade dos dados

```
void doSomething(Animal a, Animal b) {  
    if (a == b)  
        System.out.println("Objectos iguais!");  
    if (a.equals(b))  
        System.out.println("Objectos identicos!");  
}
```

Propriedades do método `equals()`

12

- O método `equals()` é responsável por implementar uma relação de equivalência de referências não nulas a objectos:

O método `equals()` é reflexivo

13

- O método `equals()` é responsável por implementar 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`

O método `equals()` é simétrico

14

- O método `equals()` é responsável por implementar uma relação de equivalência de referências não nulas a objectos:
 - **Simétrico:**
 - para quaisquer referências não nulas `x` e `y`,
`x.equals(y)` retorna `true` se e só se
`y.equals(x)` retorna `true`

O método `equals()` é transitivo

15

- O método `equals()` é responsável por implementar uma relação de equivalência de referências não nulas a objectos:
 - **Transitivo:**
 - para quaisquer 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`

O método `equals()` é consistente

16

- O método `equals()` é responsável por implementar uma relação de equivalência de referências não nulas a objectos:
 - **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`

O método `equals()` devolve sempre `false` com referências nulas

17

- O método `equals()` é responsável por implementar uma relação de equivalência de referências não nulas a objectos:
 - Para qualquer referência não nula `x`,
`x.equals(null)` retorna sempre `false`

O método `equals()` na classe `Object`

18

- 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
- Na classe `Object`, o método `equals()` retorna a forma de equivalência mais discriminatória possível:

```
public class Object {  
    ...  
    public boolean equals(Object obj) {  
        return (this == obj);  
    }  
    ...  
}
```

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

19

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

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

20

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

○ Funciona?

○ Depende...

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



O método `equals` (classe `AbstractAnimal`)

21

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

Método redefinido de `Object`, na classe
`AbstractAnimal`

```
}
```

Se `obj` referencia a mesma memória que `this`, são o mesmo objecto

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

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

```
    if (this == obj)
```

```
        return true;
```

```
}
```


Um objecto nunca é igual a `null`

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

```
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;
```

```
}
```

Se `obj` não for do mesmo tipo (ou de um subtipo deste tipo), não são iguais

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

```
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (!(obj instanceof AbstractAnimal))  
        return false;
```

```
}
```

Para as verificações seguintes, temos de tratar o `obj` como instância de `AbstractAnimal`

25 O método `equals()` (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;
```

```
}
```

As variáveis de instância têm de ser iguais

26 O método `equals()` (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;  
}
```

Os vários testes a **null** evitam que o programa possa terminar com erro, caso alguma das variáveis esteja a **null**

27 O método `equals()` (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;  
}
```

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

28

```
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;  
}
```

Devemos redefinir o equals () nas subclasses

29 O método equals () (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;  
}
```


30

O método `toString()`

O método `toString()`

31

- O método `toString()` da classe `Object` devolve uma `String` representando o objecto (e.g. `mypackage.Person@2f92e0f4`)
- 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() (classe AbstractAnimal)

32

○ 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);  
    }  
}
```

Método toString (classe AbstractAnimal)

33

○ 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!