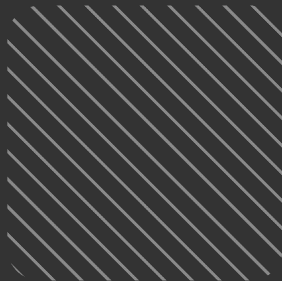


Introdução a Java

// Encapsulamento, herança e
polimorfismo

IT BOARDING

BOOTCAMP



Índice



01 Encapsulamento

02 Herança

03 Polimorfismo



IT BOARDING

BOOTCAMP

Encapsulamento

IT BOARDING

BOOTCAMP



O encapsulamento consiste em **controlar o acesso aos dados** que compõem um objeto ou instância de uma classe, ou seja, devemos indicar quais métodos e atributos são públicos para que possam ser acessados por outras entidades e até mesmo modificados. Em termos de uma classe Java, significa configurar a classe de forma que apenas os métodos dessa classe com suas variáveis possam se referir às variáveis de instância.


Java oferece suporte a modificadores de acesso para proteger os dados contra acesso e modificações imprevistas ou não controladas.



Setters e Getters

- **Getter:** É um método que, quando chamado, retorna o valor de uma variável.
- **Setter:** É um método que, quando chamado, define ou configura o valor de uma variável.

Variáveis de instância são geralmente definidas como **private**, enquanto os métodos setter e getter são **públicos** (uma vez que faz parte do princípio de encapsulamento POO), dando a outras classes a possibilidade de interagir com a classe em questão, sem expor seus métodos e atributos publicamente .



```
public class Libro {  
  
    private String titulo;  
    private String autor;  
  
    public String getTitulo() {  
        return titulo;  
    }  
  
    public void setTitulo(String titulo) {  
        this.titulo = titulo;  
    }  
  
    public String getAutor() {  
        return autor;  
    }  
  
    public void setAutor(String autor) {  
        this.autor = autor;  
    }  
}
```

Pacotes em Java (Packages)



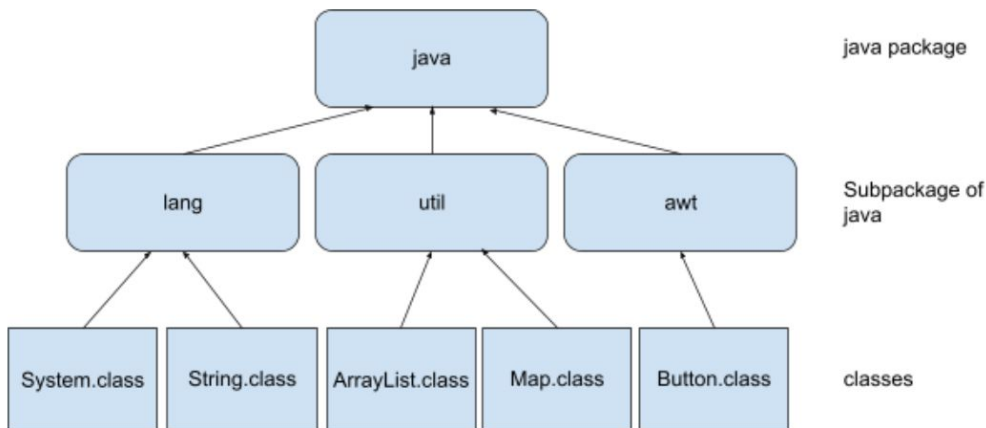
Um pacote é um grupo de classes, interfaces e subpacotes semelhantes. Eles podem ser categorizados como:

- Pacote embutido (**Built-in package**)
- Pacote definido pelo usuário (**User-defined package**)

Existem vários pacotes integrados, como **java**, **lang**, **awt**, **javax**, etc.

Vantagem

1. Eles são usados para categorizar classes e interfaces para que possam ser facilmente mantidos
2. Fornece proteção de acesso
3. Evita problemas com colisão de nomes



Modificadores de acesso



Java oferece 4 alternativas para modificadores de acesso:

- **public** → *Pode ser chamado ou acessado de qualquer classe*
- **private** → *Pode ser chamado apenas dentro da mesma classe*
- **protected** → *Pode ser chamado de classes no mesmo pacote ou subclasse*
- **default (package private)** → Pode ser chamado apenas de classes no mesmo pacote. Não há palavra reservada para acesso por padrão. Você apenas tem que omitir o modificador de acesso





Modificadores de acesso

Modificador	Mesma classe	Mesmo pacote	Subclasse	Outro Pacote
private	SIM	NÃO	NÃO	NÃO
default	SIM	SIM	NÃO	NÃO
protected	SIM	SIM	SIM	NÃO
public	SIM	SIM	SIM	SIM

INTRODUÇÃO A JAVA

Herança

IT BOARDING

BOOTCAMP

Herança



Quando criamos uma classe em Java, podemos definir que ela herda de outra classe existente. Referimo-nos ao processo no qual a **classe ou subclasse** incluirá automaticamente qualquer comportamento (*métodos*), propriedades públicas ou protegidas (*atributos*) de outra classe existente.



Herança

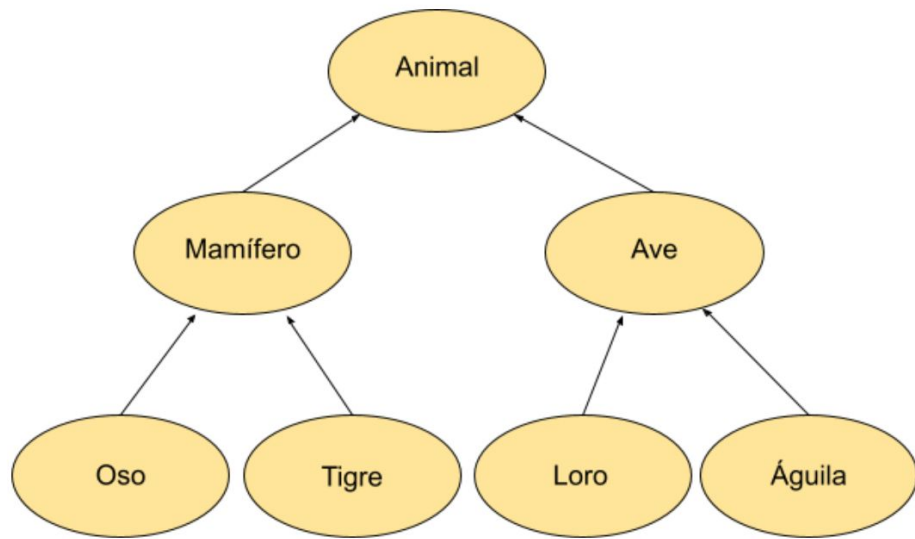


- Quando uma classe herda ou deriva de uma classe pai, nós a chamamos de ***classe filha ou subclasse***.
- Quando nos referimos a uma classe da qual as classes filhas herdarão, a chamamos de ***classe pai ou superclasse***.
- Java suporta ***herança simples***, ou seja, uma classe pode herdar apenas de uma classe pai diretamente.
- Você pode herdar ***de uma classe*** quantas vezes quiser, isso permitirá que seus filhos tenham acesso aos membros da classe pai.
- Java ***não oferece suporte a herança múltipla*** (em que uma classe pode herdar de várias classes pai), no entanto, algo semelhante a esse tipo de herança pode ser alcançado, o que veremos mais tarde neste curso.

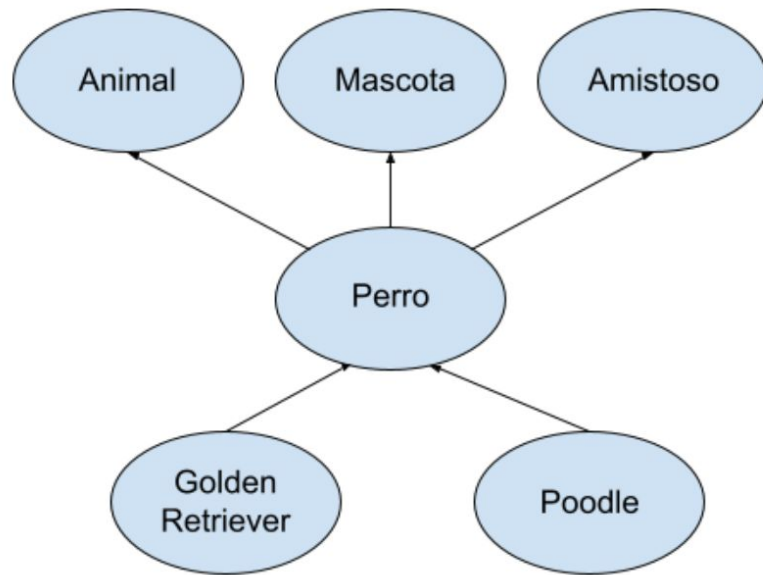




Tipos de Herança



Herança Simples



Herança Múltipla



Foi demonstrado que a herança múltipla pode levar a um código complexo que é difícil de manter, e é por isso que o Java não permite esse tipo de herança de seu design.

A classe Object

Cada classe em Java deriva (direta ou indiretamente) da classe Object. Ou seja, todas as classes *herdam de Object*.

- **boolean equals (objeto Object):**
Este método indica se outro objeto é "igual" ao atual.

"Exemplo".equals("Exemplo") → **true**

- **String toString():**
Este método retorna uma representação String do objeto.

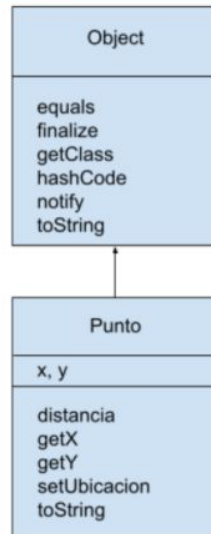
"Exemplo".toString() → **Exemplo**

- **Class<?> getClass():**
Retorna a classe de tempo de execução do objeto.

Pessoa pessoa = new Pessoa ();
pessoa.getClass() → **class Pessoa**

A Classe Object

- A Classe Object é a raiz da árvore de herança de todas as classe Java.
 - Toda Classe é implicitamente uma subclasse de Object
- A classe Object define vários métodos que serão parte de cada classe que escreves. Por Exemplo:
 - **public String toString()**
Devolve uma representação em texto de um objeto



Herdar de uma classe em Java



Podemos herdar ou derivar de uma classe adicionando o nome da classe pai em sua definição usando a palavra-chave **extends**.

Diagram illustrating the structure of a Java class definition with inheritance:

```
public class Mamifero extends Animal {  
    // Métodos e variáveis definidas aqui  
}
```

Labels and annotations:

- Modificador de acesso**: public
- Palavra-chave da classe (obrigatório)**: class
- Nome da classe**: Mamifero
- Palavra-chave extends + nome da classe Pai**: extends Animal

Vantagens de Herança



Encoraja a reutilização de código. As subclasses usam o código das superclasses.



Facilita a manutenção do aplicativo. Podemos mudar as classes que usamos facilmente.



Facilita a extensão de aplicativos. Podemos criar novas classes a partir das existentes.



Sobrescrita (Overriding)

É a maneira pela qual uma classe que herda de outra pode **redefinir** os métodos da classe pai, permitindo a criação de novos métodos que tenham o mesmo nome de sua superclasse, mas aplicando comportamentos diferentes. Podemos identificar um método sobrescrito, pois possui a anotação **@Override**, não é obrigatório em todos os casos, mas é recomendado utilizá-lo.

As condições para substituir um método são as seguintes:

- Sua assinatura deve ser **IGUAL** ao método original da classe pai, ou seja, deve ter o mesmo nome, tipo e número de argumentos dos parâmetros.
- O tipo de retorno deve ser o mesmo.
- Não deve ter um nível de acesso mais restritivo do que o original (por exemplo, se estiver protegido na classe pai, não pode ser privado na classe filha).
- Eles não devem ser métodos estáticos ou finais (uma vez que estático representa métodos globais e finais constantes).



Exemplo de sobreescrita

```
public class Instrumento {  
  
    public String tipo;  
  
    public void tocar() {  
        System.out.println("Tocar un instrumento");  
    }  
}  
  
public class Guitarra extends Instrumento {  
  
    @Override  
    public void tocar() {  
        //Lógica para tocar la guitarra  
        System.out.println("Tocar la guitarra");  
    }  
}
```

Conforme
observado aqui,
fazer isso também
é válido

```
Instrumento instrumento = new Instrumento();  
instrumento.tocar();  
Guitarra guitarra = new Guitarra();  
guitarra.tocar();  
Instrumento instrumentoGuitarra = new Guitarra();  
instrumentoGuitarra.tocar();
```

```
"C:\Program Files\Java\jdk-11.0.11\bin\java.exe" "-  
Tocar un instrumento  
Tocar la guitarra  
Tocar la guitarra
```

E se você quiser criar um novo instrumento ... por exemplo, um trompete ... as etapas para aprender a tocar esse instrumento seriam as mesmas de um violão?

Exemplo de sobreescrita

Provavelmente não! Portanto, o comportamento da classe pai **Instrumento** pode ser sobrescrito novamente e adaptado ao novo objeto a ser criado.

```
}  
  
public class Trompeta extends  
Instrumento {  
  
    @Override  
    public void tocar() {  
        //Lógica para tocar la trompeta  
        System.out.println("Tocar la  
trompeta");  
    }  
}
```

```
Instrumento instrumento = new Instrumento();  
instrumento.tocar();  
Guitarra guitarra = new Guitarra();  
guitarra.tocar();  
Trompeta trompeta = new Trompeta();  
trompeta.tocar();
```

```
Main  
"C:\Program Files\Java\jdk-11.0.11\bin\java  
Tocar un instrumento  
Tocar la guitarra  
Tocar la trompeta  
  
Process finished with exit code 0
```

INTRODUÇÃO A JAVA

Polimorfismo

IT BOARDING

BOOTCAMP

"É um dos **pilares** da programação orientada a objetos, refere-se à **propriedade** de um objeto assumir **diferentes formas**".

IT BOARDING

BOOTCAMP



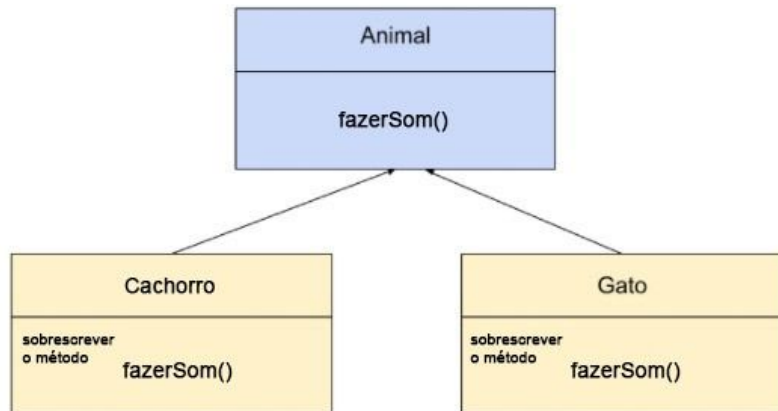


Vamos ver um exemplo

IT BOARDING

BOOTCAMP





```
public class Animal {  
  
    public void fazerSom() {  
        System.out.println("O animal faz um som");  
    }  
}
```

```
public class Gato extends Animal {  
  
    @Override  
    public void fazerSom() {  
        System.out.println("Miau");  
    }  
}
```

```
public class Cachorro extends Animal {  
  
    @Override  
    public void fazerSom() {  
        System.out.println("Auau");  
    }  
}
```

O que é impresso se executarmos o código a seguir?

```
public class ExemploPolimorfismo {  
  
    public static void main(String[] args) {  
        Animal animal = new Animal();  
        animal.fazerSom();  
        Cachorro cachorro = new Cachorro();  
        cachorro.fazerSom();  
        animal = new Gato();  
        animal.fazerSom();  
    }  
}
```

O animal faz um som

Auau

Miau





Obrigado

IT BOARDING

BOOTCAMP

