

Sobrescritas Importantes, Empacotamento e Acesso

Apresentação



Fonte: <https://goo.gl/EMCdcL>

Neste momento, daremos uma pausa nos pilares da orientação a objetos. Você já estudou a abstração, o encapsulamento e a herança. Ainda falta o polimorfismo. Antes disso, nesta aula, serão abordados alguns recursos importantes que reforçarão os conteúdos estudados anteriormente em outros contextos.

*Na aula anterior, o foco foi a Herança e a Sobrescrita. Esses conceitos são muito importantes e merecem ser abordados de outro prisma. Nesta aula, você conhecerá então a classe **Object**, e o que ela representa para todas as*

classes Java. Conhecerá sobreescritas importantes, como `equals` e `toString` .

Além disso, serão vistos os modificadores de acesso e o padrão de empacotamento dos arquivos de um projeto Java. Relembre os conceitos da aula passada e vamos em frente.

Conteúdo

A Classe *Object*

No Java, existe uma classe genérica, que é superclasse de todas as classes, o nome dela é **Object**. Todas as classes criadas no Java herdam de Object implicitamente. Esta classe fornece alguns métodos iniciais e serve para que suas entidades sejam reconhecidas como objeto. O método `add` de ArrayList, por exemplo, possui uma assinatura `public boolean add(Object obj)`, ou seja, qualquer tipo de objeto você pode adicionar em uma coleção do tipo ArrayList (`ArrayList col = new ArrayList();`). Se passar no teste "é um", ou seja, se "herdar" de object você pode adicionar. Como todas herdam então... todas podem ser adicionadas. Um objeto criado por um programador Java do tipo Professor, String, Animal, Veiculo tem a codificação implícita "extends Object".

Para Refletir

Se a extensão de Object não acontecesse de forma implícita, qual seria o prejuízo?

Lembre-se, no Java não existe Herança múltipla, caso não fosse implícita a herança entre Object e todas as classes do Java não seria possível realizar outros relacionamentos de Herança, ou fazer com que todas as classes fossem tratadas como um Object (uma classe genérica para todas).

Na última aula, você estudou o conceito de Herança... o que este importante pilar preconiza? É a possibilidade de herdar características e ações de outra entidade anteriormente criada para fins de **REUSO**, certo? A classe *object* fornece alguns métodos importantes para todas as classes java. A classe possui métodos com muitas finalidades, como o uso de reflexão, o controle de concorrência, manipulação de memória, etc. Mas estes não são temas desta disciplina, que abordará somente dois métodos importantes: o `toString` e o `equals`.

O Método `toString`

Este método tem uma definição muito simples. É uma representação textual do estado de um objeto (valores presentes nos atributos e/ou resultados de métodos) em um dado momento. O objeto retorna uma String que o representa em forma de texto por meio deste método. O significado é bem a tradução do nome, to String = "Para String".

O `toString` é originário de `Object`. Consequentemente, sua implementação original não atende a todas as necessidades das subclasses. O `toString` original retorna o nome da classe concatenado com `@` e o endereço de memória do objeto. Exemplo de saída para uma classe `Aluno: Aluno@234e435a`. Lembre-se: cada vez que rodar o programa, o endereço de memória pode ser diferente. Logo, devemos usar o recurso da sobrescrita, para que este método se comporte da forma esperada; afinal endereço de memória é pouco significativo. Observe o exemplo do código na Tabela 1.

Tabela 1 - Exemplo de codificação Java com sobrescrita do `toString`

Programa em Java: Programa.java

```
public class Programa {  
    public static void main(String[] args) {  
        Aluno aluno = new Aluno("Pedro Brito", 10, 6);  
        System.out.println(aluno.toString());  
    }  
}  
class Aluno{  
    private String nome;  
    private double notaUm;  
    private double notaDois;  
  
    public double getMedia(){  
        return (getNotaUm() + getNotaDois()) / 2;  
    }  
  
    //toString para representação do objeto em forma de String.  
    @Override  
    public String toString() {  
        String texto = "\nNome: "+getNome()+"\n"  
            +"Nota 1: "+getNotaUm()+"\n"  
            +"Nota 2: "+getNotaDois()+"\n"  
            +"Media : "+getMedia()+"\n"  
            +"Status: "+(getMedia() > 7 ? "Aprovado":"Reprovado")+"\n";  
        return texto;  
    }  
    //sobrescrita do construtor  
    public Aluno() {}  
    public Aluno(String nome, double notaUm, double notaDois) {  
        setNome(nome);  
        setNotaUm(notaUm);  
    }  
}
```

```
        setNotaDois(notaDois);
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public double getNotaUm() {
        return notaUm;
    }
    public void setNotaUm(double notaUm) {
        this.notaUm = notaUm;
    }
    public double getNotaDois() {
        return notaDois;
    }
    public void setNotaDois(double notaDois) {
        this.notaDois = notaDois;
    }
}
```

Perceba que o `toString`, quando é acionado, traz em forma de String (texto) os valores que estão em seus atributos, inclusive o que é calculado por método.

Não existe uma regra para implementação do `toString`. Portanto, faça da forma que considerar melhor a representação textual.

O Método **Equals**

Imagine a seguinte situação: Você deve comparar se um objeto do tipo pessoa é igual a outro. Suponha que uma pessoa tenha nome e cpf. Observe o trecho de código na Tabela 2.

Tabela 2 – Exemplo de codificação Java de comparação.

Programa em Java:**Comparacao.java**

Programa em Java:Comparacao.java

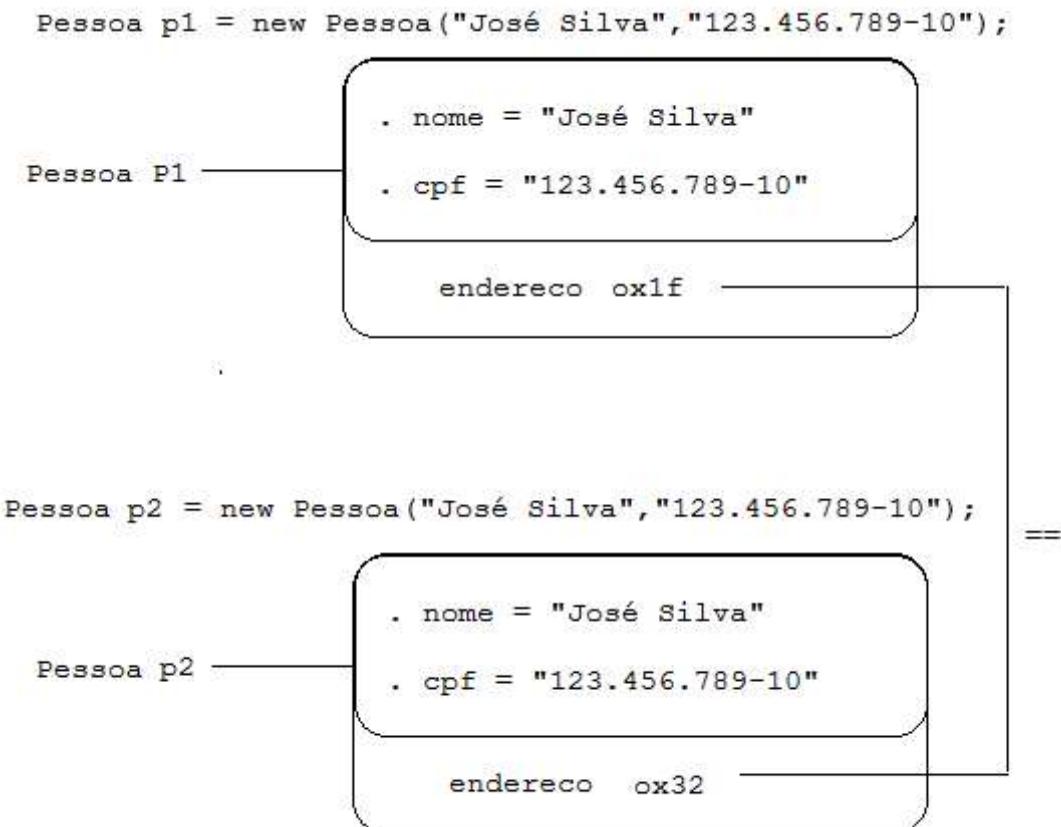
```
public class Comparacao {  
    public static void main(String[] args) {  
        Pessoa p1 = new Pessoa("José Silva", "123.456.789-10");  
        Pessoa p2 = new Pessoa("José Silva", "123.456.789-10");  
        //L1 if(p1 == p2){  
        //L2     System.out.println("As pessoas são iguais!");  
        //L3 } else{  
        //L4     System.out.println("As pessoas são diferentes!");  
        //      }  
    }  
    class Pessoa{  
        private String nome;  
        private String cpf;  
        //sobre carga do método construtor  
        public Pessoa() {}  
        public Pessoa (String nome, String cpf){  
            setNome(nome);  
            setCpf(cpf);  
        }  
        //métodos acessores  
        public String getNome() {  
            return nome;  
        }  
        public void setNome(String nome) {  
            this.nome = nome;  
        }  
        public String getCpf() {  
            return cpf;  
        }  
        public void setCpf(String cpf) {  
            this.cpf = cpf;  
        }  
    }  
}
```

Analisando a codificação acima, perceba que dois objetos do tipo Pessoa foram criados exatamente com os mesmos valores. Significativamente, eles são iguais. Você acha que vai acontecer o quê? Será impresso que as pessoas são iguais ou diferentes? Rode este programa no seu IDE para verificar o que acontece.

E aí? Surpreso? Pois é... Imprimiu que as pessoas são diferentes. Certo?

O primeiro erro da codificação acima é que na L1 está sendo utilizado o operador de igualdade `==` para comparação. Nada mais justo. Entretanto, este operador, quando aplicado com objetos, compara endereço de memória. A Figura 1 representa o que aconteceu em memória no exemplo da Tabela 1.

Figura 1 - Representação da memória do exemplo da Tabela 1



Perceba que os endereços alocados para os objetos são diferentes. Logo, a comparação retornará falso. E os objetos são declarados como diferentes.

A comparação de endereço de memória para a programação de alto nível e para o que queremos pouco importa, não é verdade? Mas o que fazer? Resposta: Trocar a comparação por equals, ao invés de `==`. Vamos lá?

Troque a comparação que está sendo feita em L1 por `if(p1.equals(p2))` e observe o que acontece.

Viu? Não mudou nada, pois continuou dizendo que são diferentes. Veja... você definiu o método equals? Não. Não existe este método na codificação presente na Tabela 1. Como você conseguiu chamá-lo então? Este método está sendo herdado da classe Object. E quando ele é herdado, o comportamento padrão deste método é comparar também o endereço de memória. Mas o que fazer? Lembra-se de que na última aula você aprendeu que uma subclasse, quando herda características e ações, pode

sobrescrever (especializar) um determinado método a sua maneira? Quando sobreescrito, ele se comportará conforme o redefinido na subclasse. Então... Preparado para sobreescriver o método equals?

Olhando para a classe Pessoa, qual seria um atributo candidato a diferenciar um objeto de outro? O CPF, não é mesmo? Pois bem, devemos sobreescrivê-lo o método equals, para que ele compare os cpfs dos objetos como vistas a determinar se são iguais ou diferentes. Acompanhe a Tabela 3 que apresenta uma proposta da classe pessoa com a sobreescrita do método equals.

Tabela 3 – Proposta de método equals para a classe Pessoa.

Programa em Java: Class Pessoa

Programa em Java: Class Pessoa

```
class Pessoa{  
    private String nome;  
    private String cpf;  
    //sobre carga do método construtor  
    public Pessoa() {}  
    public Pessoa (String nome, String cpf){  
        setNome(nome);  
        setCpf(cpf);  
    }  
    //métodos acessores  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
    public String getCpf() {  
        return cpf;  
    }  
    public void setCpf(String cpf) {  
        this.cpf = cpf;  
    }  
    @Override  
    public boolean equals(Object obj) {  
        if(obj instanceof Pessoa){// obj é uma pessoa?  
            //casting de Object para Pessoa.  
            Pessoa param = (Pessoa) obj;  
            //usando o equals de String, não confunda.  
            if(getCpf().equals(param.getCpf())){  
                return true;//são iguais  
            }else  
                return false;//são diferentes  
        }  
        return false;//Se não for pessoa já retorna.  
    }  
}
```

Para Refletir ⚡

O que o `instanceof` faz? Por que fazer este teste no `equals`?

O operador instanceof serve para verificar se o objeto da esquerda é uma instância da classe da direita. Como o argumento passado é um Object (qualquer coisa), esse teste faz com que não haja um erro de conversão (o *casting* que acontece abaixo do comando *if*). Ora, se o argumento não for de um tipo Pessoa nem precisa executar nenhuma instrução, apenas retorne false. Não faz sentido comparar bananas com laranjas, certo?

Com essa codificação, ficou determinada de forma significativa como um objeto Pessoa é comparado. Se você atualizar a classe pessoa, conforme a Tabela 3, e executar o programa, perceberá que agora dirá que os objetos são iguais. Mude o valor do CPF de uma das pessoas e verá que as pessoas serão diferentes.

Definição de Pacotes

A linguagem Java é orientada a objetos, entretanto, a organização dos seus arquivos é orientada a pacotes. Imagine o seguinte: você quando está programando em Java e cria arquivos .java para materializar seus programas. Então, um programador define uma classe com o mesmo nome que a sua. É de conhecimento de todos que o sistema operacional não aceita arquivos com mesmo nome no mesmo diretório, não é verdade? Pronto, temos um problema. Você pode se perguntar: "Está sendo falado de pacote ou de diretórios?" Na verdade, quando se define um pacote de um arquivo .java, o que está sendo definido é a sua organização em diretório.

Definir o pacote de um arquivo .java é determinar o seu caminho relativo em relação ao projeto onde ficam os fontes. Por exemplo, se tivermos um projeto chamado SISA (Sigla fictícia para Sistema Acadêmico), se declararmos um pacote br.ucb.sisa.entidades e colocarmos dentro dele Pessoa.java, o eclipse (seu IDE) criará uma estrutura de diretórios assim: SISA/src/br/ucb/sisa/entidades/Pessoa.java

Percebeu a nomenclatura de pacotes? Somente pela definição, conseguimos concluir que quem desenvolveu o sistema foi a UCB, que o nome do sistema é sisa e que o pacote tem objetivo de guardar as entidades do sistema. E o "br"? Vem do site da UCB (www.ucb.br). Se o final do site da UCB fosse "com.br" a declaração do pacote ficaria assim: br.com.ucb.sisa.entidades. Se a empresa desenvolvedora não tiver *site*, deve ser utilizada a localização da empresa no globo, no caso seria br porque a empresa é do Brasil. Se fosse de Portugal, seria pt e assim por diante.

Para realizar a declaração, ficaria da forma como apresentada na Tabela 4.

Tabela 4 – Trecho de código que representa a declaração de uma classe com pacote.

Programa em Java: Pessoa.java

Programa em Java: Pessoa.java

```
package br.ucb.sisa.entidades;
class Pessoa{
    private String nome;
    private String cpf;
    ...
    ...
//fim código
}
```

A declaração do package deve estar na primeira linha do arquivo .java. Uma classe pode ter apenas uma declaração de package, afinal é a declaração do caminho relativo da classe. Mas ela pode ter tantas importações de pacotes (import) quantas precisar. Todos eles devem vir abaixo da declaração do package. Lembra-se do import lá atrás, sobre o qual não foram apresentados muitos detalhes? Chegou a hora!

Para poder fazer uso de uma classe que está em outro diretório, você precisa dizer ao java onde ela está localizada. O import serve para que você não precise colocar o nome completo da classe (o nome completo de uma classe inclui o seu pacote). Ou seja, para que não seja necessário fazer o seguinte na criação de um objeto do tipo Scanner:

```
java.util.Scanner scan = new java.util.Scanner(System.in);
```

Sem o import, os programadores ficam obrigados a colocar o nome completo para utilização do .java.

Agora, se declararmos o import, basta o seguinte:

```
import java.util.Scanner;
Scanner scan = new Scanner(System.in);
```

Modificadores de Acesso

No tópico anterior, você aprendeu que é possível organizar uma aplicação em pacotes (pastas). Isso ajuda na organização, distribuição e criação de componentes. A programação sofre impacto direto por conta desta organização.

Finalizando... C

Chegamos ao final desta aula conhecendo vários conceitos novos e apurando algumas arestas. Pode reconhecer a importância da sobrescrita de alguns métodos originários da classe Pai de todas as classes: a Object.

Você conheceu os modificadores de acesso e percebeu que a organização dos arquivos impacta na programação por meio da visibilidade dos membros.

A partir disso, acreditamos que agora você está pronto para prosseguir em direção ao polimorfismo.

Bons estudos!

Quando se fala em modificadores de acesso, é o mesmo que dizer sobre visibilidade de um membro para outras entidades. Até o momento, você conheceu dois modificadores de acesso: o `public` e o `private`.

Um membro com modificador de acesso `public` pode ser acessado de qualquer lugar, ou seja, não precisa estar na mesma classe e pode estar em outros pacotes. Já, o modificador de acesso `private` restringe o acesso do membro à própria classe. Quando não há modificador nenhum, somente a declaração do membro o acesso é *default*. Isso quer dizer que, dentro do mesmo pacote, o membro pode ser acessado. Por fim, falta apresentar a você o modificador `protected` em que um membro pode ser acessado de duas formas: por pacote, assim como o *default*, e por herança. Mesmo que uma classe esteja em outro pacote, se houver herança, o membro poderá ser acessado.

Resumindo, o Java possui 3 modificadores de acesso (`private`, `protected` e `public`) e 4 níveis de acesso: na própria classe (`private`), por pacote ou herança (`protected`), de qualquer lugar (`public`) e somente por pacote (`default`, sem declaração).

Assista ao **vídeo 1** para você visualizar alguns conceitos desta aula na prática.

Vídeo 1 

Prática Profissional - Programação orientada à objeto - Unidad...



Na Prática

"Prezado(a) estudante,

Esta seção é composta por atividades que objetivam consolidar a sua aprendizagem quanto aos conteúdos estudados e discutidos. **Caso alguma dessas atividades seja avaliativa, seu (sua) professor (a) indicará no Plano de Ensino e lhe orientará quanto aos critérios e formas de apresentação e de envio.**"

Bom Trabalho!

Atividade 01

^

Crie uma classe pessoa com nome, código inteiro e salário. A classe social de uma pessoa é calculada conforme o seguinte:

Classe

A1: inclui as famílias com renda mensal maior que R\$ 14.400

Classe A2: maior que R\$ 8.100

Classe B: maior que R\$ 4.600

Classe C: maior que R\$ 2.300

Classe D: maior que R\$ 1.400

Classe E: maior que R\$ 950

Classe F: maior que R\$ 400

Receba do usuário os dados de uma pessoa e apresente todas as informações da pessoa, inclusive a classe por meio do método `toString`. **Atenção!** O método `toString` não deve fazer o cálculo de classe da pessoa.

Atividade 02



Utilizando a mesma modelagem do exercício anterior, receba os dados de várias pessoas e armazene em um ArrayList. Entretanto, não é permitido armazenar duas pessoas com o mesmo código (sobrescreva o equals). Após receber os dados, enquanto o usuário desejar, apresente todas as pessoas informadas usando o toString. Além da implementação do equals e toString, realize o empacotamento e distribua suas classes sabendo que você trabalha na empresa anonimousti e ela possui um site: www.anonimousti.com.br . O nome do sistema é peoplesoft.

Saiba Mais

Para ampliar seu conhecimento a respeito desse assunto, veja abaixo a(s) sugestão(ões) do professor:

- Leia o Capítulo 4 do livro *Core Java - Fundamentos*, por Cay S. Horstman e Gary Cornell.
- Leia o artigo "[Métodos, atributos e classes no Java](#)" para ampliar seus conhecimentos sobre os modificadores de acesso.

Referências

- BOOCH, G. **Object-Oriented Design with Applications**, Benjamin-Cummings, 1991.
- BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **UML – guia do usuário**. 2. ed. Rio de Janeiro: Campus. 2006.
- CARDELLI, L.; WEGNER, P. **On Understanding Types, Data Abstraction, and Polymorphism**. ACM Computing Surveys (CSUR). vol. 17, pp. 471-523. 1985.
- DEITEL H. M.; DEITEL, P. J. **Java, como programar**. 6. ed. Porto Alegre: Bookman. 2006.
- DICIONÁRIO AURELIO. 2017. Disponível em:
<https://contas.tcu.gov.br/dicionario/home.asp> Acesso em: 19 mar. 2017.
- ERICH, G. et al. **Padrões de projeto**: soluções reutilizáveis de software rientado a objetos. Porto Alegre: Bookman. 2000.
- HORSTMANN, C. S.; CORNELL, G. **Corejava 2 – Volume I – Fundamentals**. São Paulo: Makron Books. 2010.
- NEWRELIC. **The Most Popular Programming Languages of 2016**. 2016. Disponível em:
<https://blog.newrelic.com/2016/08/18/popular-programming-languages-2016-go>. Acesso em: 9 Mar 2017.
- NIEMEYER, P.; KNUDSEN, J. **Aprendendo Java**. Rio de Janeiro: Campus. 2000.
- ORACLE. **Java Licensing Logo**. 2017. Disponível em:
<http://www.oracle.com/us/technologies/java/java-licensing-logo-guidelines-1908204.pdf>. Acesso em: 13 mar. 2017.
- SIERRA, K.; BATES, B. **Certificação Sun para Programador JAVA 5 Guia de Estudo**. Rio de Janeiro: Alta Books, 2006.
- SILBERSCHATZ, A; GALVIN, P. B.; GAGNE, G. **Sistemas operacionais com Java**. 6. ed. Rio de Janeiro. Editora Campus, 2004.

- STUCKEY, P. J.; SULZMANN, M. **A theory of overloading**. International Conference on Functional Programming. Proceedings of the seventh ACM SIGPLAN international conference on Functional programming. Pittsburgh, PA, USA, 2002. pp. 167-178.
- WIKIPÉDIA. Desenvolvido pela Wikimedia Foundation. Conteúdo sobre o **Ambiente de Desenvolvimento Integrado**. 2017. Disponível em: <https://pt.wikipedia.org/wiki/Ambiente_de_desenvolvimento_integrado>. Acesso em: 5 mar. 2017.