fusberti@ic.unicamp.br

Polimorfismo: Classes Abstratas e Interfaces

prof. Fábio Luiz Usberti

MC322 - Programação Orientada a Objetos

Instituto de Computação - UNICAMP - 2014

Sumário

Introdução

Recapitulando: características de herança em Java

- Uma subclasse tem acesso direto a todos os membros `public` e `protected` de sua superclasse, independente se as duas classes se encontram no mesmo pacote ou não.
- Se uma subclasse e superclasse estão no mesmo pacote, a subclasse também tem acesso direto aos membros `package-private` da superclasse.
- É possível declarar um atributo em uma subclasse com o mesmo nome de um atributo de uma superclasse (não recomendado), o que causa um ocultamento.
- É possível declarar novos atributos na subclasse que não se encontram na superclasse.

Introdução

Recapitulando: características de herança em Java

- É possível declarar um método de instância na subclasse com a mesma assinatura de um método de instância da superclasse, o que causa uma sobreposição.
- É possível declarar um método estático na subclasse com a mesma assinatura de um método estático da superclasse, o que causa um ocultamento.
- É possível declarar novos métodos na subclasse que não se encontram na superclasse.
- É possível declarar um construtor para a subclasse que chama o construtor da superclasse, implicitamente ou explicitamente com a palavra-chave `super`.

Introdução

Motivações do comportamento polimórfico

- Polimorfismo é uma característica de uma linguagem orientada a objetos que permite programar em um nível geral ao invés de programar em um nível específico.
- O polimorfismo permite manipular objetos de diferentes classes que compartilham de uma mesma superclasse (direta ou indireta), dado que eles são considerados objetos da superclasse.
- Por exemplo, simulação do movimento de diversos tipos de animais.
- Suponha que as classes Peixe, Sapo e Pássaro sejam subclasses de Animal, onde são herdados um estado (posição do animal), e um comportamento (método mover).

Introdução

Motivações do comportamento polimórfico

- Para simular os movimentos dos animais, o programa instancia objetos das classes Peixe, Sapo e Pássaro e envia mensagens para que eles se movimentem.
- Cada objeto deve realizar o movimento de acordo com seu tipo específico: o peixe deve nadar X metros, o sapo deve saltar Y metros e o pássaro deve voar Z metros.
- Desso modo, cada animal deve alterar sua posição de acordo com seu movimento.
- A mesma mensagem (mover) enviada para cada objeto resultou em formas distintas de comportamento, justificando o nome polimorfismo.
- O conceito central do polimorfismo é de assegurar que cada objeto se comporte de forma apropriada de acordo com seu tipo.

Introdução

Motivações do comportamento polimórfico

- Com polimorfismo, é possível projetar e implementar sistemas computacionais extensíveis, pois novas classes podem ser adicionadas com poucas mudanças no código.
- Para isso, novas classes devem ser parte de uma hierarquia de herança que o programa processa de forma generalizada.
- Por exemplo, adicionar uma nova subclasse de animal, Tartaruga, requer somente a implementação do comportamento dessa nova classe.
- O código responsável pela simulação dos animais não precisará sofrer alterações, pois o comportamento específico da subclasse Tartaruga não é relevante para esse código.

Introdução

Motivações do comportamento polimórfico

- Através do polimorfismo, é possível comandar objetos para realizarem serviços sem precisar conhecer seus tipos específicos, contanto que esses objetos pertençam a uma mesma hierarquia de herança.
- A extensibilidade é atingida dado que o polimorfismo confere ao programa a independência dos tipos específicos dos objetos para os quais mensagens são enviadas.
- Objetos de tipos novos, que implementam métodos já existentes, podem ser incorporados no sistema sem precisar modificar o código que processa os objetos de forma generalizada.
- Somente a porção do código responsável por instanciar novos objetos deverá ser modificada para acomodar novos tipos.

Exemplos de polimorfismo

Exemplo: Quadriláteros

- Considere a classe Retângulo que deriva da classe Quadrilátero.
- Qualquer operação relativa a um quadrilátero, como calcular o perímetro ou área, também se aplica a um retângulo.
- Isso também é verdade para quadrados, paralelogramas e trapézios.
- Nesse contexto, um exemplo de polimorfismo ocorre ao invocar um método através de uma referência Quadrilátero, pois nesse caso o método executado será o da subclasse a qual o objeto pertence.

Introdução

Comportamento polimórfico

- Como um objeto de uma subclasse é um objeto da superclasse, isso permite manipulações interessantes desses objetos.
- Por exemplo, é possível criar um vetor do tipo de uma superclasse que referencia um conjunto de objetos de subclasses distintas.
- Outro exemplo, é possível escrever um método onde um dos parâmetros corresponde ao tipo de uma superclasse. Quando chamado, esse método pode receber um objeto de qualquer subclasse.

Introdução

Acoplamento dinâmico

- A referência de uma superclasse pode ser utilizada para chamar somente métodos que estão declarados na superclasse, mesmo que o objeto seja de uma subclasse.
- A invocação de um método da superclasse torna necessário a resolução pela JVM de qual método deve ser chamado (caso o objeto corresponda a uma subclasse).
- Esse processamento polimórfico que determina o método apropriado de uma subclasse é denominado acoplamento dinâmico (*dynamic binding* ou *dynamic dispatch*).
- O acoplamento dinâmico é um processo que ocorre em tempo de execução.

Downcasting

- Sabe-se que o objeto de uma subclasse pode ter como referência uma variável com um tipo mais geral, de uma superclasse.
- As vezes, torna-se necessário converter desse tipo mais geral para o tipo mais específico, condizente com a subclasse a qual o objeto pertence.
- O compilador Java permite designar a uma referência de superclasse um tipo mais específico, processo denominado de downcasting.
- O downcasting é interessante quando queremos invocar algum método específico da subclasse, que não são acessíveis através da referência da superclasse.
- Se um downcasting for realizado entre duas referências incompatíveis, ocorrerá um erro de execução.

Introdução

Referências de superclasses e subclasses

A seguir é descrito uma síntese de atribuições permitidas entre variáveis de uma superclasse e sua subclasse:

		Referência para	
		Superclasse	Subclasse
Referência de	Superclasse	OK	Erro de compilação*
	Subclasse	OK**	OK

1. * O erro de compilação pode ser evitado por meio de um downcasting.
2. ** Esse caso não resulta em erro pois um objeto do tipo subclasse é um objeto do tipo superclasse. No entanto, a referência da superclasse só poderá acessar os membros da superclasse.

Métodos e classes abstratas

Motivações

- Nem sempre, ao modelar uma classe, estamos interessados em instanciar objetos dessa classe.
- Algumas vezes desejamos declarar classes que são utilizadas somente como superclasses em uma hierarquia de herança, sem nunca ter um objeto instanciado (com exceção dos objetos de subclasses).
- Por exemplo, em uma hierarquia de uma classe Forma, referente a formas geométricas, as subclasses poderiam herdar somente uma noção geral do que significa ser um objeto Forma.
- Essa noção geral pode expressa expressa por meio de atributos (cor, borda, espessura, posição) e comportamentos (desenhar, mover, redimensionar, recolorir), sem atenção aos detalhes de implementação.

Métodos e classes abstratas

Motivações

- Uma classe abstrata tem por objetivo ser uma superclasse através da qual subclasses possam herdar e compartilhar de um certo modelo ainda não totalmente concretizado.
- As classes abstratas são classes gerais que não instanciam objetos, mas que especificam de forma geral fatores em comum entre suas subclasses.
- Em contrapartida, as classes que instanciam objetos são denominadas classes concretas.
- As classes concretas devem fornecer implementações (ou herdar implementações) para todos os métodos declarados em sua hierarquia.
- Por exemplo, uma superclasse Forma2D pode servir de classe abstrata para as subclasses concretas Círculo, Quadrado e Triângulo.

Métodos e classes abstratas

Declaração

- Uma classe ou método abstrato podem ser declarados com a palavra-chave `abstract`. Exemplos:

```
public abstract class myAbstractClass {
    public abstract void myAbstractMethod();
}
```

- Métodos abstratos contêm somente uma assinatura, portanto não devem fornecer uma implementação.
- Uma classe deve ser declarada abstrata se contém pelo menos um método abstrato, ainda que ela também contenha métodos concretos.
- Uma subclasse concreta precisa sobrescrever todos os métodos abstratos das superclasses, fornecendo uma implementação.

Métodos e classes abstratas

Declaração

- Construtores e métodos estáticos não podem ser declarados abstratos, pois métodos abstratos tem por função ser sobrepostos para assumirem comportamentos específicos nas subclasses.
- Como os construtores não são herdados, eles nunca poderiam ser sobrepostos em uma subclasse.
- Apesar dos métodos estáticos (não privados) serem herdados, eles não podem ser sobrepostos (somente ocultados). No entanto, é possível utilizar o nome de superclasses abstratas para chamar métodos estáticos contidos nessas classes.

Métodos e classes abstratas

Declaração

- As variáveis de instância e métodos concretos de uma classe abstrata estão sujeitos às regras normais de herança.
- Tentar instanciar um objeto com o tipo de uma classe abstrata consiste em um erro de compilação.
- Deixar de implementar um método abstrato em uma subclasse consiste em um erro de compilação, exceto se a subclasse também for declarada abstrata.

Estudo de caso com classe abstrata

Sistema de pagamento de funcionários

Uma companhia deseja implementar um aplicativo que realiza o pagamentos de seus funcionários, considerando os seguintes requisitos:

- Os funcionários são pagos semanalmente.
- Os funcionários são enquadrados em quatro categorias: Assalariados, Sob honorários, Comissionados e Comissionados com salário base.
- Para a semana corrente, a companhia decidiu gratificar os comissionados com salário base um adicional de 10% sob seus salários bases.

Estudo de caso com classe abstrata

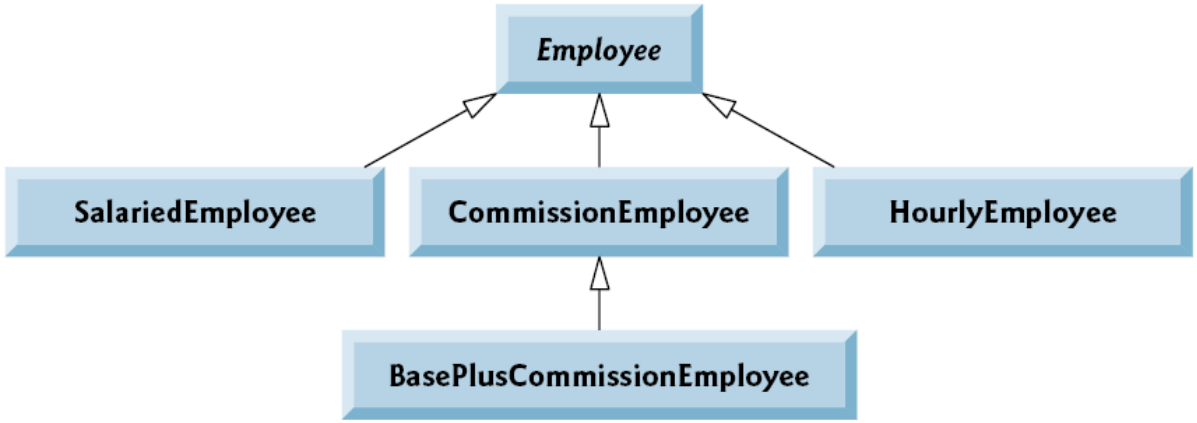
Sistema de pagamento de funcionários

Categorias de funcionários:

- Assalariados (*salaried employees*): são funcionários pagos com um valor fixo semanal, independente do número de horas trabalhadas.
- Sob honorários (*hourly employees*): são funcionários pagos por hora de trabalho e que recebem hora-extra (uma vez e meia o valor do honorário) para todas as horas trabalhadas acima da jornada de 40 horas.
- Comissionados (*comission employees*): são pagos com base em uma porcentagem de suas vendas.
- Comissionados com salário base (*base-salaried comission employee*): são funcionários que recebem um salário base mais uma porcentagem de suas vendas.

Estudo de caso com classe abstrata

Hierarquia de classes de funcionários



Estudo de caso com classe abstrata

```
// Employee.java
// classe Employee representa um funcionário
public abstract class Employee {

    private String firstName;
    private String lastName;
    private String cpf;

    // construtor
    public Employee(String first, String last, String argCpf) {
        firstName = first ;
        lastName = last;
        cpf = argCpf;
    } // fim construtor

    // Métodos acessores

    public void setFirstName(String first ) {
        firstName = first ;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setLastName(String last) {
        lastName = last;
    }

    public String getLastName() {
        return lastName;
    }

    /* continua na próxima página */
```

Estudo de caso com classe abstrata

```
    /* continua da página anterior */

    public void setCpf(String argCpf) {
        // TODO: incluir método de validação de CPF
        cpf = argCpf;
    }

    public String getCpf() {
        return cpf;
    }

    // método toString retorna uma string representando o objeto Employee
    @Override
    public String toString() {
        return String.format("%s: %s %s\n%s: %s", "Nome", getFirstName(),
                getLastName(), "CPF", getCpf());
    } // fim método toString

    // método abstrato que deve ser sobreposto pelas subclasses concretas
    public abstract double earnings(); // somente assinatura

} // fim classe Employee
```

Estudo de caso com classe abstrata

```
// SalariedEmployee.java
// classe SalariedEmployee representa um funcionário assalariado
public class SalariedEmployee extends Employee {
    private double weekSalary;

    // construtor
    public SalariedEmployee(String first, String last, String ssn, double salary) {
        super(first, last, ssn); // pass to Employee constructor
        setWeekSalary(salary); // validate and store salary
    } // fim construtor

    // Métodos acessores

    public void setWeekSalary(double salary) {
        weekSalary = salary < 0.0 ? 0.0 : salary;
    }

    public double getWeekSalary() {
        return weekSalary;
    }

    // método earnings retorna os vencimentos do funcionário
    @Override
    public double earnings() {
        return getWeekSalary();
    } // fim método earnings

    // método toString retorna uma string representando o objeto
    @Override
    public String toString() {
        return String.format("Assalariado:\n%s\n%s: $%,.2f", super.toString(),
                "Salário Semanal", getWeekSalary());
    } // fim método toString

} // fim classe SalariedEmployee
```

Estudo de caso com classe abstrata

```
// HourlyEmployee.java
// classe HourlyEmployee representa um funcionário com honorários
public class HourlyEmployee extends Employee {
    private double wage; // valor da hora trabalhada
    private double hours; // número de horas trabalhadas

    // construtor
    public HourlyEmployee(String first, String last, String ssn,
            double hourlyWage, double hoursWorked) {
        super(first, last, ssn);
        setWage(hourlyWage);
        setHours(hoursWorked);
    } // fim construtor

    // Métodos acessores

    // método setWage valida e armazena o valor dos honorários
    public void setWage(double hourlyWage) {
        wage = (hourlyWage < 0.0) ? 0.0 : hourlyWage;
    } // fim método setWage

    public double getWage() {
        return wage;
    }

    // método setHours valida e armazena as horas trabalhadas
    public void setHours(double hoursWorked) {
        hours = ((hoursWorked >= 0.0) && (hoursWorked <= 168.0)) ? hoursWorked : 0.0;
    } // fim método setHours

    public double getHours() {
        return hours;
    }

    /* continua na próxima página */
```

Estudo de caso com classe abstrata

```
    /* continua da página anterior */

    // método earnings retorna os vencimentos do funcionário
    @Override
    public double earnings() {
        if (getHours() <= 40) // sem hora-extra
            return getWage() * getHours();
        else
            return 40 * getWage() + (getHours() - 40) * getWage() * 1.5;
    } // fim método earnings

    // método toString retorna uma string representando o objeto
    public String toString() {
        return String.format("Honorário:\n%s\n%s: $%,.2f; %s: %,.2f",
                super.toString(), "Valor do honorário", getWage(),
                "Horas trabalhadas", getHours());
    } // fim método toString

} // end class HourlyEmployee
```

Estudo de caso com classe abstrata

```
// CommissionEmployee.java
// classe CommissionEmployee representa um funcionário comissionado
public class CommissionEmployee extends Employee {
    private double grossSales; // total de vendas
    private double commissionRate; // taxa de comissão

    // construtor
    public CommissionEmployee(String first, String last, String cpf,
            double sales, double rate) {
        super(first, last, cpf);
        setGrossSales(sales); // valida e armazena o total de vendas
        setCommissionRate(rate); // valida e armazena a taxa de comissão
    } // fim construtor

    // Métodos acessores

    public void setGrossSales(double sales) {
        // total de vendas deve ser um valor não-negativo
        grossSales = (sales < 0.0) ? 0.0 : sales;
    }

    public double getGrossSales() {
        return grossSales;
    }

    public void setCommissionRate(double rate) {
        // taxa de comissão deve ser um valor no intervalo aberto (0,1)
        commissionRate = (rate > 0.0 && rate < 1.0) ? rate : 0.0;
    }

    public double getCommissionRate() {
        return commissionRate;
    }

    /* continua na próxima página */
```

Estudo de caso com classe abstrata

```
    /* continua da página anterior */

    // método earnings retorna os vencimentos do funcionário
    @Override
    public double earnings() {
        return getCommissionRate() * getGrossSales();
    } // fim método earnings

    // método toString retorna uma string representando o objeto
    // CommissionEmployee
    @Override
    public String toString() {
        return String.format("Comissionado:\n%s \n%s: %.2f\n%s: %.2f",
                super.toString(), "Total de vendas", getGrossSales(),
                "Taxa de comissão", getCommissionRate());
    } // fim método toString
} // fim classe CommissionEmployee
```

Estudo de caso com classe abstrata

```
// BasePlusCommissionEmployee.java
// classe BasePlusCommissionEmployee representa um funcionário comissionado com um salário base
public class BasePlusCommissionEmployee extends CommissionEmployee {
    private double baseSalary; // salário base semanal

    // construtor
    public BasePlusCommissionEmployee(String first, String last, String ssn,
            double sales, double rate, double salary) {
        super(first, last, ssn, sales, rate);
        setBaseSalary(salary); // valida e armazena o valor do salário base
    } // fim construtor

    // Métodos acessores

    public void setBaseSalary(double salary) {
        baseSalary = (salary < 0.0) ? 0.0 : salary; // valor não—negativo
    }

    public double getBaseSalary() {
        return baseSalary;
    }

    // método earnings retorna os vencimentos do funcionário
    @Override
    public double earnings() {
        return getBaseSalary() + super.earnings(); // salário base mais comissão
    } // fim método earnings

    // método toString retorna uma string representando o objeto
    @Override
    public String toString() {
        return String.format("Salário base + %s\n%s: %.2f", super.toString(),
                "Salário base", getBaseSalary());
    } // fim método toString
} // fim classe BasePlusCommissionEmployee
```

Estudo de caso com classe abstrata

```
// PayrollSystem.java
// Classe PayrollSystem testa todas as subclasses funcionários, com e sem o uso de polimorfismo.
public class PayrollSystem {
    public static void main(String args[]) {
        // cria os objetos das subclasses de funcionários
        SalariedEmployee salariedEmployee = new SalariedEmployee("Fulano",
                "Silva", "111.111.111-11", 800.00);
        HourlyEmployee hourlyEmployee = new HourlyEmployee("Beltrano", "Souza",
                "222.222.222-22", 16.75, 40);
        CommissionEmployee commissionEmployee = new CommissionEmployee(
                "Ciclano", "Costa", "333.333.333-33", 10000, .06);
        BasePlusCommissionEmployee basePlusCommissionEmployee = new BasePlusCommissionEmployee(
                "Mengano", "Santos", "444.444.444-44", 5000, .04, 300);

        // cria um vetor para armazenar quatro objetos Employee
        Employee employees[] = new Employee[4];

        // inicializando o vetor com os objetos das subclasses de funcionários
        employees[0] = salariedEmployee;
        employees[1] = hourlyEmployee;
        employees[2] = commissionEmployee;
        employees[3] = basePlusCommissionEmployee;
    /* continua na próxima página */
```

Estudo de caso com classe abstrata

```
    /* continua da página anterior */

        // Impressão do pagamento dos funcionários
        System.out.println("Pagamento dos Funcionários:\n");

        for (Employee currentEmployee : employees) {
            // chama o método toString de forma polimórfica
            System.out.println(currentEmployee);

            // determina qual objteo é do tipo BasePlusCommissionEmployee
            if (currentEmployee instanceof BasePlusCommissionEmployee) {
                // realiza um downcast de Employee para
                // BasePlusCommissionEmployee
                BasePlusCommissionEmployee employee = (BasePlusCommissionEmployee) currentEmployee;

                double oldBaseSalary = employee.getBaseSalary();
                employee.setBaseSalary(1.10 * oldBaseSalary);
                System.out.printf("Novo salário base com 10%% de aumento será de: $%,.2f\n",
                                  employee.getBaseSalary());
            }

            System.out.printf("Vencimentos: $%,.2f\n\n", currentEmployee.earnings());
        }

        // verifica para cada funcionário qual é sua subclasse correspondente
        for (Employee currentEmployee : employees)
            System.out.printf("Funcionário %s %s é um funcionário tipo %s\n", currentEmployee.getFirstName(),
                currentEmployee.getLastName(),currentEmployee.getClass().getSimpleName());
    } // fim main
} // fim classe PayrollSystemTest
```

Estudo de caso com classe abstrata

Pagamento dos Funcionários:

Assalariado:
Nome: Fulano Silva
CPF: 111.111.111-11
Salário Semanal: $800.00
Vencimentos: $800.00

Honorário:
Nome: Beltrano Souza
CPF: 222.222.222-22
Valor do honorário: $16.75; Horas trabalhadas: 40.00
Vencimentos: $670.00

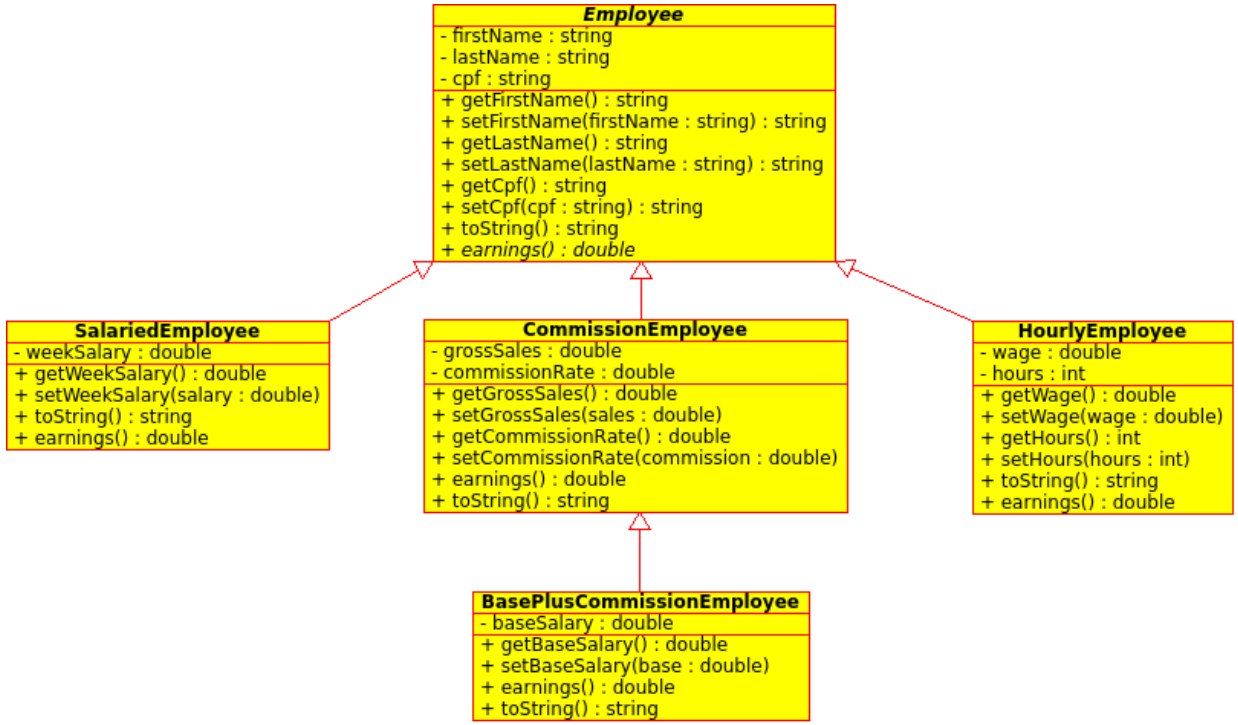
Comissionado:
Nome: Ciclano Costa
CPF: 333.333.333-33
Total de vendas: 10000.00
Taxa de comissão: 0.06
Vencimentos: $600.00

Salário base + Comissionado:
Nome: Mengano Santos
CPF: 444.444.444-44
Total de vendas: 5000.00
Taxa de comissão: 0.04
Salário base: 300.00
Novo salário base com 10% de aumento será de: $330.00
Vencimentos: $530.00

Funcionário Fulano Silva é um funcionário tipo SalariedEmployee
Funcionário Beltrano Souza é um funcionário tipo HourlyEmployee
Funcionário Ciclano Costa é um funcionário tipo CommissionEmployee
Funcionário Mengano Santos é um funcionário tipo BasePlusCommissionEmployee

Estudo de caso com classe abstrata

Representação em UML



Estudo de caso com classe abstrata

Observações

- Os métodos da classe `Employee` podem ser chamados a partir de uma variável `Employee`. Isso inclui os métodos herdados por `Employee`, por exemplo os métodos de `Object`.
- No entanto, nem todos os métodos da classe `CommissionEmployee` podem ser chamados a partir de uma variável `Employee`.
- Tentar atribuir o valor (referência) de uma variável `Employee` para uma variável `ComissionEmployee` resulta em um erro de compilação, exceto se for realizado um *downcasting*. No exemplo, esse erro de compilação decorre do fato de que um `Employee` nem sempre é um `BasePlusComissionEmployee`. O *downcasting* indica ao compilador que esse `Employee` em particular também é um `BasePlusComissionEmployee`.

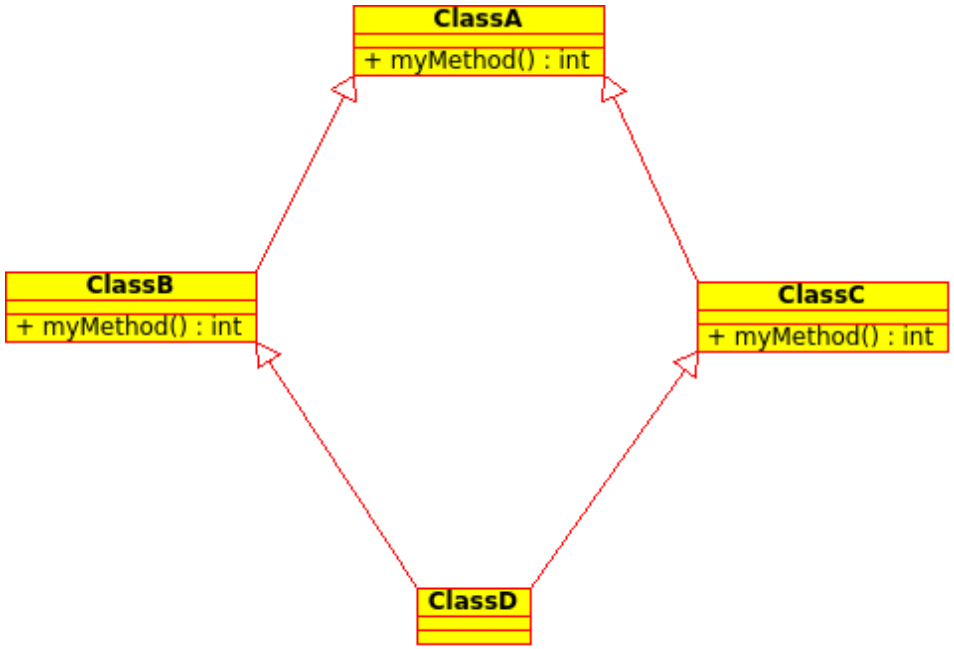
Herança Múltipla

Múltiplas superclasses diretas

- Java suporta herança simples, no qual uma classe pode ter somente uma superclasse direta.
- A herança múltipla é uma característica da programação orientada a objetos onde uma subclasse herda entidades de mais de uma superclasse.
- A herança múltipla têm sido debatida por muitos anos devido à complexidade e ambiguidade que ela pode trazer ao programa.
- O problema do diamante é uma ambiguidade que ocorre quando duas classes `B` e `C` herdam de uma classe `A` e, além disso, uma classe `D` herda das classes `B` e `C`.

Herança Múltipla

Problema do Diamante



- Considere que um método em `ClassA` foi sobreposto em `ClassB` e em `ClassC`. Suponha também que não há sobreposição desse método em `ClassD`. Qual versão do método `ClassD` vai herdar, a versão sobreposta por `ClassB` ou por `ClassC`?

Herança Múltipla

Múltiplas superclasses diretas

- O problema do diamante é um dos motivos pelos quais Java não suporta herança múltipla de implementação, onde métodos já implementados são herdados pela subclasse.
- Java provê uma outra espécie de herança múltipla, conhecida por herança múltipla de tipo.
- Na herança múltipla de tipo, uma subclasse pode herdar as assinaturas de métodos de múltiplas superclasses especiais, denominadas interfaces.
- A herança múltipla de tipos não sofre do problema do diamante.
- Mesmo quando mais de uma interface possui a mesma assinatura de um método, assim que esse método for implementado (definido) em uma classe da hierarquia de herança, ele se sobrepõe a qualquer assinatura na cadeia de suas superclasses.

Interfaces

Motivações

- As interfaces definem e padronizam a interação entre objetos.
- Por exemplo, os controles de um rádio servem como uma interface entre um ouvinte e os componentes internos do rádio.
- Os controles do rádio permitem que os usuários realizem um conjunto de operações (mudar estação, ajustar volume, mudar de AM para FM).
- Essas operações podem ser implementadas de forma manual, digital ou por comando de voz.
- O objetivo da interface consiste portanto em definir quais operações são realizadas mas sem especificar como essas operações serão implementadas.

Interfaces

Declaração

- As interfaces podem ser consideradas classes especiais em Java que possibilitam outras classes implementarem métodos em comum, mesmo quando não há uma clara relação entre essas classes.
- Uma interface em Java é implicitamente abstrata e deve ser declarada com a palavra-chave `interface` e com uma lista de possíveis superinterfaces (com a palavra-chave `extends`).

```
public interface MyInterface extends Interface1, ..., InterfaceN {

    // declaração de constantes

    // número de Euler
    double E = 2.718282;

    // assinatura de métodos
    void doSomething (int i, double x);
    int  doSomethingElse(String s);

}
```

Interfaces

Declaração

- Somente métodos abstratos e constantes podem ser definidos em uma interface.
- Em uma interface, todos os métodos são implicitamente públicos e abstratos e os atributos são implicitamente públicos, estáticos e finais.
- A convenção de nomes Java para interfaces é de que elas sigam as mesmas regras de nomes de classes.
- Preferencialmente, o nome deve se referir a uma capacidade provida pela interface. Por exemplo: `Comparable` (Comparável), `Cloneable` (Clonável), `Serializable` (Serializável).
- Ao declarar um método em uma interface, escolha um nome que descreva o propósito geral do método, dado que ele poderá ser implementado por muitas classes distintas.

Interfaces

Utilização

- Para utilizar uma interface, uma classe deve declarar que implementa essa interface (palavra-chave `implements`).
- Java não permite que uma classe possa ter mais de uma superclasse direta. No entanto, é possível ter uma superclasse direta e implementar múltiplas interfaces.

```
// ClassName é subclasse de SuperClassName e implementa todo o conjunto Interface1, ..., InterfaceN
public class ClassName extends SuperClassName implements Interface1, ..., InterfaceN { }
```

Interfaces

Utilização

- Implementar uma interface pode ser considerado um contrato com o compilador, atestando que todos os métodos da interface serão definidos (implementados) ou que a classe que implementa a interface será declarada abstrata. Caso contrário, haverá um erro de compilação.
- Quando uma classe implementa uma interface, o relacionamento "é um" também se aplica a esse caso, de forma similar ao relacionamento de herança.
- Dessa forma, um objeto cuja subclasse implementa múltiplas interfaces também pode ser considerado como um objeto de cada um dos tipos definidos pelas interfaces.

Interfaces

Polimorfismo em interfaces

- Uma interface é comumente utilizada quando classes díspares precisam compartilhar métodos e constantes.
- As interfaces possibilitam que tais classes, que não apresentam uma relação clara, possam ser processadas de modo polimórfico. Em outras palavras, os objetos de classes que implementam uma mesma interface podem responder a uma mesma chamada de método.
- Utilizando a referência para uma interface, é possível invocar de forma polimórfica qualquer método declarado na interface, suas superinterfaces e métodos da classe Object.
- Quando um parâmetro de um método é declarado com um tipo de uma superclasse ou interface, o método processa de modo polimórfico o objeto recebido como argumento.

Interfaces

Interfaces vs. Classes Abstratas

Considere utilizar uma classe abstrata nas seguintes situações:

- Você deseja compartilhar código para diversas classes que estão intimamente relacionadas.
- Você exige modificadores de acesso como `protected` ou `private`.
- Você deseja declarar atributos não estáticos ou não finais.
- Você deseja definir métodos que acessem e modifiquem o estado do objeto ao qual pertencem.

Interfaces

Interfaces vs. Classes Abstratas

Considere utilizar uma interface nas seguintes situações:

- Você espera que muitas classes, sem uma clara relação entre si, implementem a interface. Por exemplo, as interfaces `Comparable` e `Cloneable` são implementadas por muitas classes completamente distintas.
- Você deseja especificar o comportamento de um tipo de dado, sem se preocupar como ele será implementado.
- Você deseja utilizar herança múltipla de tipo.

Estudo de caso com interface

Sistema de pagamentos gerais

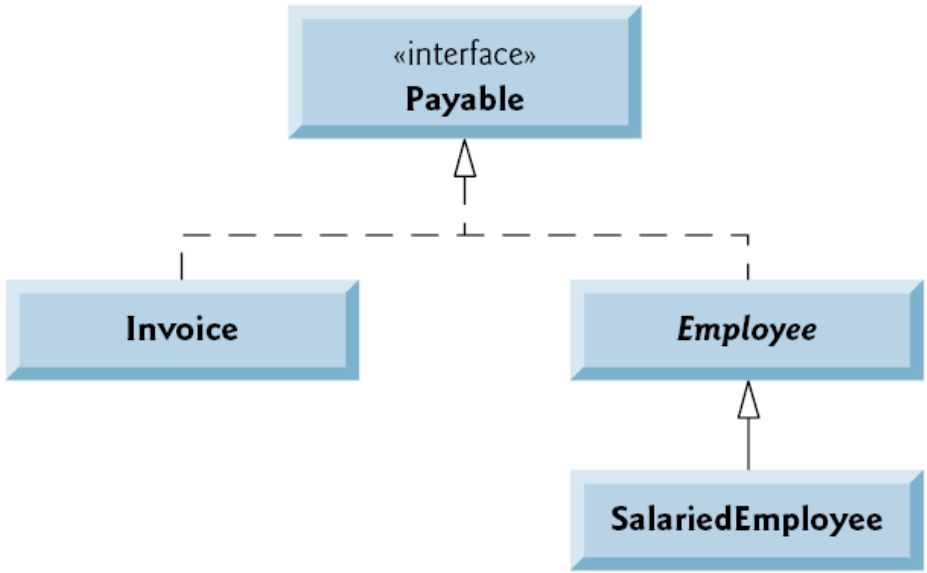
Supondo que a companhia apresenta uma nova demanda de realizar operações de contabilidade diversas em uma mesma aplicação. São considerados os seguintes requisitos:

- Calcular os vencimentos pagos a cada funcionário e os pagamentos realizados para um conjunto de recibos de compras de materiais.
- Apesar de não relacionados, os funcionários e os recibos demandam uma operação em comum que se trata de obter um certo valor de pagamento.
- Para um funcionário, o pagamento se refere aos seus vencimentos.
- Para um recibo, um pagamento se refere ao custo total de materiais listados no recibo.

A aplicação a seguir implementa uma interface `Payable` que será implementada pelas classes `Employee` e `Invoice` para que seja retornado o valor do pagamento.

Estudo de caso com interface

Hierarquia de classes com interface



Estudo de caso com interface

```
// Payable.java
// Declaração da interface Payable que representa um tipo que requer um pagamento.

public interface Payable {
    double getPaymentAmount(); // determina pagamento (método abstrato)
} // fim interface Payable
```

Estudo de caso com interface

```
// Employee.java
// classe Employee representa um funcionário que implementa a interface Payable
public abstract class Employee implements Payable {

    private String firstName;
    private String lastName;
    private String cpf;

    // construtor
    public Employee(String first, String last, String argCpf) {
        firstName = first ;
        lastName = last;
        cpf = argCpf;
    } // fim construtor

    // Métodos acessores

    public void setFirstName(String first ) {
        firstName = first ;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setLastName(String last) {
        lastName = last;
    }

    public String getLastName() {
        return lastName;
    }

    /* continua na próxima página */
```

Estudo de caso com interface

```
    /* continua da página anterior */

    public void setCpf(String argCpf) {
        // TODO: incluir método de validação de CPF
        cpf = argCpf;
    }

    public String getCpf() {
        return cpf;
    }

    // método toString retorna uma string representando o objeto Employee
    @Override
    public String toString() {
        return String.format("%s: %s %s\n%s: %s", "Nome", getFirstName(),
                getLastName(), "CPF", getCpf());
    } // fim método toString

    // método abstrato que deve ser sobreposto pelas subclasses concretas
    @Deprecated
    public abstract double earnings();

    // Obs: O método getPaymentAmount da interface Payable não foi implementado
    // nesta classe portanto é necessário declará-la como abstrata.

} // fim classe Employee
```

Estudo de caso com interface

```
// SalariedEmployee.java
// classe SalariedEmployee representa um funcionário assalariado que implementa a interface Playable
public class SalariedEmployee extends Employee {
    private double weekSalary;

    // construtor
    public SalariedEmployee(String first, String last, String ssn, double salary) {
        super(first, last, ssn); // pass to Employee constructor
        setWeekSalary(salary); // validate and store salary
    } // fim construtor

    // Métodos acessores

    public void setWeekSalary(double salary) {
        weekSalary = salary < 0.0 ? 0.0 : salary;
    }

    public double getWeekSalary() {
        return weekSalary;
    }

    // método earnings retorna os vencimentos do funcionário
    @Deprecated
    @Override
    public double earnings() {
        return getWeekSalary();
    } // fim método earnings

    /* continua na próxima página */
```

Estudo de caso com interface

```
    /* continua da página anterior */

    // retorna os vencimentos do funcionário; implementação do método abstrato
    // da interface Payable
    @Override
    public double getPaymentAmount() {
        return getWeekSalary();
    } // end method getPaymentAmount

    // método toString retorna uma string representando o objeto
    @Override
    public String toString() {
        return String.format("Assalariado:\n%s\n%s: $%,.2f", super.toString(),
                "Salário Semanal", getWeekSalary());
    } // fim método toString

} // fim classe SalariedEmployee
```

Estudo de caso com interface

```
// Invoice.java
// A classe Invoice representa um recibo que implementa a interface Payable.

public class Invoice implements Payable {
    private String itemNumber; // número do item
    private String itemDescription; // descrição do item
    private int quantity; // quantidade de itens
    private double pricePerItem; // preço unitário

    // construtor
    public Invoice(String item, String description, int count, double price) {
        itemNumber = item;
        itemDescription = description;
        setQuantity(count);
        setPricePerItem(price);
    } // fim construtor

    public void setItemNumber(String number) {
        itemNumber = number;
    }

    public String getItemNumber() {
        return itemNumber;
    }

    public void setItemDescription(String description) {
        itemDescription = description;
    }

    public String getItemDescription() {
        return itemDescription;
    }

    /* continua na próxima página */
```

Estudo de caso com interface

```
    /* continua da página anterior */

    // método que valida e armazena a quantidade de produtos adquiridos
    public void setQuantity(int count) {
        quantity = (count < 0) ? 0 : count; // quantidade não pode ser negativa
    } // fim método setQuantity

    public int getQuantity() {
        return quantity;
    }

    // método que valida e armazena o preço por item do produto
    public void setPricePerItem(double price) {
        pricePerItem = (price < 0.0) ? 0.0 : price; // valor não pode ser
                                                    // negativo
    } // fim método setPricePerItem

    public double getPricePerItem() {
        return pricePerItem;
    }

    // método toString retorna uma string representando o objeto
    public String toString() {
        return String.format("%s: \n%s: %s (%s) \n%s: %d \n%s: $%,.2f",
                "Recibo", "Id do item", getItemNumber(), getItemDescription(),
                "Quantidade", getQuantity(), "Preço unitário",
                getPricePerItem());
    } // fim método toString

    // Método que atende ao contrato imposto pela interface Payable
    @Override
    public double getPaymentAmount() {
        return getQuantity() * getPricePerItem(); // determina o custo total
    } // fim método getPaymentAmount
} // fim classe Invoice
```

Estudo de caso com interface

```
// PayrollSystem.java
// Classe PayrollSystem testa a interface Payable.

public class PayrollSystem {
    public static void main(String args[]) {
        // Cria um vetor com referências a objetos do tipo Payable
        Payable payableObjects[] = new Payable[4];

        // inicializa o vetor
        payableObjects[0] = new Invoice("01234", "cadeira", 2, 375.00);
        payableObjects[1] = new Invoice("56789", "pneu", 4, 79.95);
        payableObjects[2] = new SalariedEmployee("Fulano",
                "Silva", "111.111.111-11", 800.00);
        payableObjects[3] = new SalariedEmployee("Beltrano", "Souza",
                "222.222.222-22", 650.00);

        System.out
                .println("Recibos e funcionários com processamento polimórfico:\n");

        // Processamento de cada elemento do vetor
        for (Payable currentPayable : payableObjects) {
            // imprime o valor pago referente a cada objeto:
            System.out.printf("%s \n%s: $%,.2f\n\n", currentPayable.toString(),
                    "Pagamento efetuado", currentPayable.getPaymentAmount());
        } // fim for
    } // fim main
} // fim classe PayrollSystem
```

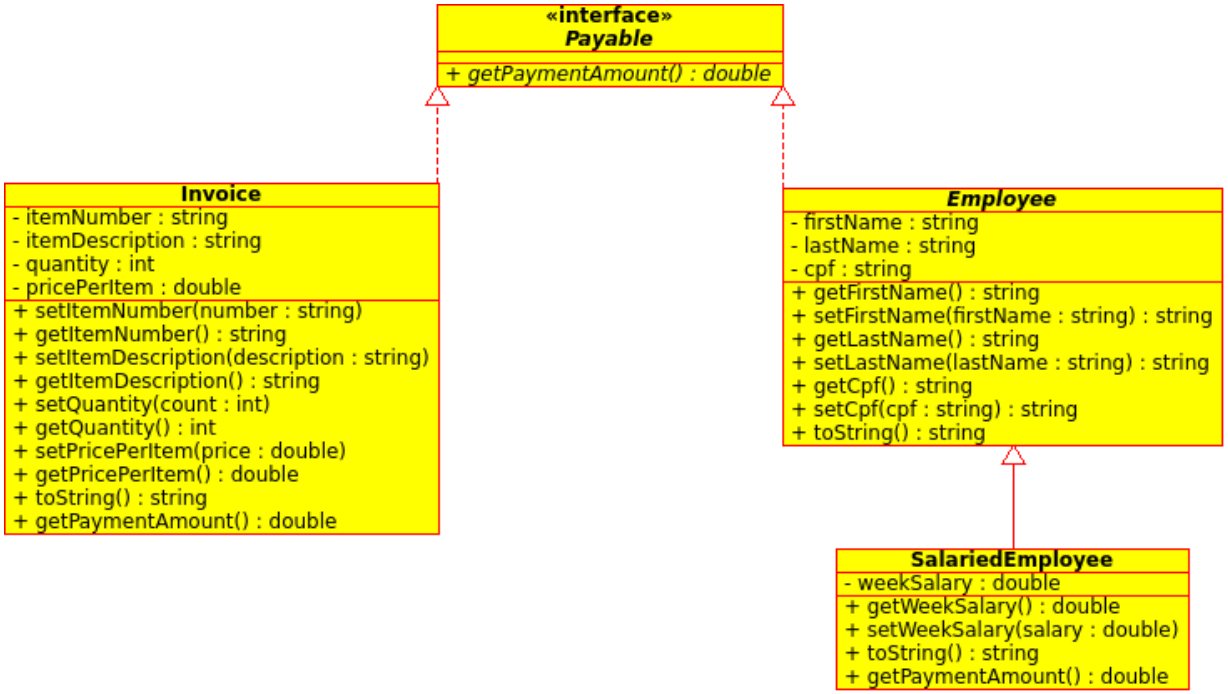
Estudo de caso com interface

Recibos e funcionários com processamento polimórfico:

```
Recibo:
Id do item: 01234 (cadeira)
Quantidade: 2
Preço unitário: $375.00
Pagamento efetuado: $750.00

Recibo:
Id do item: 56789 (pneu)
Quantidade: 4
Preço unitário: $79.95
Pagamento efetuado: $319.80

Assalariado:
Nome: Fulano Silva
CPF: 111.111.111-11
Salário Semanal: $800.00
Pagamento efetuado: $800.00

Assalariado:
Nome: Beltrano Souza
CPF: 222.222.222-22
Salário Semanal: $650.00
Pagamento efetuado: $650.00
```

Estudo de caso com interface

Representação em UML



Interface `Comparable`

Ordenação de objetos

- A interface `Comparable<T>` é responsável por impôr uma ordem aos objetos de uma classe genérica `T`.
- Essa interface contém um único método `compareTo` responsável por definir a ordem dos objetos.
- Vetores e `ArraysList` de objetos cujas classes implementam a interface `Comparable` podem ser ordenados utilizando os métodos `Arrays.sort` e `Collections.sort`.

Interface `Comparable`

Método `compareTo`

- O método `compareTo` recebe a referência de um objeto e retorna um número inteiro, como mostra sua assinatura:

```
int compareTo(T o)
```

- Esse método compara o objeto `this` (corrente) com o objeto referenciado pelo parâmetro `o`.
- O valor de retorno deve ser um inteiro negativo, zero ou positivo se o objeto `this` for menor, igual ou maior do que o objeto referenciado `o`, respectivamente.
- Considere o método `sgn(<expression>)`, que retorna -1, 0 ou 1, se o valor de `expression` for negativo, zero ou positivo, respectivamente.

Interface `Comparable`

Método `compareTo`

- Na implementação do método `compareTo`, o programador deve respeitar as seguintes restrições:
  1. Simetria: `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))`.
  2. Transitividade: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implica em `x.compareTo(z)>0`.
  3. Distributividade: `x.compareTo(y)==0` implica em `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`.
- É fortemente recomendado, porém não obrigatório, que a seguinte condição seja respeitada: `(x.compareTo(y)==0) == (x.equals(y))`. Isso implica, em regra geral, na necessidade de sobrepor o método `equals` da classe `Object`.

Interface `Comparable`

```
// Employee.java
// classe Employee representa um funcionário
public abstract class Employee implements Comparable<Employee> {

    private String firstName;
    private String lastName;
    private String cpf;

    // construtor
    public Employee(String first, String last, String argCpf) {
        firstName = first ;
        lastName = last;
        cpf = argCpf;
    } // fim construtor

    // Métodos acessores

    public void setFirstName(String first ) {
        firstName = first ;
    }

    public String getFirstName() {
        return firstName;
    }

    /* continua na próxima página */
```

Interface `Comparable`

```
/* continua da página anterior */

public void setLastName(String last) {
    lastName = last;
}

public String getLastName() {
    return lastName;
}

public void setCpf(String argCpf) {
    // TODO: incluir método de validação de CPF
    cpf = argCpf;
}

public String getCpf() {
    return cpf;
}

// método toString retorna uma string representando o objeto Employee
@Override
public String toString() {
    return String.format("%s: %s %s\n%s: %s", "Nome", getFirstName(),
            getLastName(), "CPF", getCpf());
} // fim método toString

// método abstrato que deve ser sobreposto pelas subclasses concretas
public abstract double earnings(); // somente assinatura

/* continua na próxima página */
```

Interface `Comparable`

```
    /* continua da página anterior */

    // ordenando os objetos de modo crescente
    @Override
    public int compareTo(Employee obj) {
        String str1 = this.cpf.replaceAll("[-.]", "");
        String str2 = obj.cpf.replaceAll("[-.]", "");
        long x = Long.parseLong(str1);
        long y = Long.parseLong(str2);
        if (x < y)
            return -1;
        else if (x > y)
            return 1;
        return 0;
    }

    // sobrepondo o método equals para corresponder ao método compareTo
    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Employee) {
            Employee obj1 = (Employee) obj;
            return this.compareTo(obj1) == 0;
        }
        return false;
    }

} // fim classe Employee
```

Interface Comparable

```
import java. util .Arrays;
// SortingTest.java
// Classe SortingTest testa a ordenção dos objetos da classe Employee.
public class SortingTest {
    public static void main(String args[]) {
        // cria os objetos das subclasses de funcionários
        SalariedEmployee salariedEmployee = new SalariedEmployee("Fulano",
                "Silva", "111.111.111-11", 800.00);
        HourlyEmployee hourlyEmployee = new HourlyEmployee("Beltrano", "Souza",
                "222.222.222-22", 16.75, 40);
        CommissionEmployee commissionEmployee = new CommissionEmployee(
                "Ciclano", "Costa", "333.333.333-33", 10000, .06);
        BasePlusCommissionEmployee basePlusCommissionEmployee = new BasePlusCommissionEmployee(
                "Mengano", "Santos", "444.444.444-44", 5000, .04, 300);

        // inicializando o vetor com os objetos das subclasses de funcionários
        Employee employees[] = new Employee[4];
        employees[0] = basePlusCommissionEmployee;
        employees[1] = commissionEmployee;
        employees[2] = salariedEmployee;
        employees[3] = hourlyEmployee;

        System.out.println("ANTES da ordenação:\n");
        for (Employee currentEmployee : employees)
            System.out.println(currentEmployee + "\n");

        Arrays.sort(employees); // ordenando de modo crescente

        System.out.println("DEPOIS da ordenação:\n");
        for (Employee currentEmployee : employees)
            System.out.println(currentEmployee + "\n");

    } // fim main
} // fim classe SortingTest
```

Interface `Comparable`

```
ANTES da ordenação:

Salário base + Comissionado:
Nome: Mengano Santos
CPF: 444.444.444-44
Total de vendas: 5000.00
Taxa de comissão: 0.04
Salário base: 300.00

Comissionado:
Nome: Ciclano Costa
CPF: 333.333.333-33
Total de vendas: 10000.00
Taxa de comissão: 0.06

Assalariado:
Nome: Fulano Silva
CPF: 111.111.111-11
Salário Semanal: $800.00

Honorário:
Nome: Beltrano Souza
CPF: 222.222.222-22
Valor do honorário: $16.75; Horas trabalhadas: 40.00
```

Interface `Comparable`

```
DEPOIS da ordenação:

Assalariado:
Nome: Fulano Silva
CPF: 111.111.111-11
Salário  Semanal: $800.00

Honorário:
Nome: Beltrano Souza
CPF: 222.222.222-22
Valor  do  honorário:  $16.75; Horas trabalhadas: 40.00

Comissionado:
Nome: Ciclano Costa
CPF: 333.333.333-33
Total  de vendas: 10000.00
Taxa de comissão: 0.06

Salário  base + Comissionado:
Nome: Mengano Santos
CPF: 444.444.444-44
Total  de vendas: 5000.00
Taxa de comissão: 0.04
Salário  base: 300.00
```

Referências

1. Java: Como Programar, Paul Deitel & Heivey Deitel; Pearson; 7a. Ed. (no. chamada IMECC – 05.133 D368j)
2. Data Structures and Algorithms with Object Oriented Design Patterns in Java, Bruno Preiss; (http://www.brpreiss.com/books/opus6/)
3. The Java Tutorials (Oracle) (http://docs.oracle.com/javase/tutorial/)
4. Guia do Usuário UML, Grady Booch et. al.; Campus(1999)
5. Java Pocket Guide - Robert Liguori & Patricia Liguori; O'Reilley, 2008.