Introdução à Programação Orientada por Objectos em Java

- Enquadramento e Objectivos
- Abstracção
- Encapsulamento
- Herança
- Polimorfismo
- Conclusão

Introdução

- Realçar as vantagens da aplicação do paradigma da POO
- Analogia com muitas actividades económicas
  - um produto é fabricado através da integração de vários componentes seleccionados e adquiridos
- Em programação
  - utilização de módulos integráveis com características e funcionalidades independentes do contexto

Conceitos Fundamentais

- A POO baseia-se em quatro princípios chave:
  - Abstracção
  - Encapsulamento
  - Herança
  - Polimorfismo

Abstracção

- Extracção das características essenciais de um conceito ou entidade no mundo real para o poder processar num computador.
- Nível superior de abstracção:
  - Através do encapsulamento, e
  - Através do polimorfismo

Encapsulamento

- Todas as linguagens têm uma maneira de armazenar pedaços de informação relacionada juntos, normalmente designados estrutura ou registo.

Encapsulamento

- O encapsulamento consiste na agregação dos dados e seus comportamentos numa única unidade organizacional - **classe**.
  - Estrutura de uma Classe
  - Construtores
  - Modificadores de acesso
  - Integridade dos tipos de dados
  - Vantagens do encapsulamento

Linguagem C

- Representar frutos através do peso e calorias por grama

```
typedef  struct  {
        int gramas;
        int caloriasPorGrama;
}  Fruta ;
```

```
int totalCalorias( Fruta * f ) {
        return (f->gramas)*(f->caloriasPorGrama);
}
```

```
int calorias;
Fruta f1;
f1.gramas = 5;
f1.caloriasPorgrama = 60;
calorias = totalCalorias(& f1);
```

Pseudo-código C

```
struct Fruta {
        int gramas;
        int caloriasPorGrama;
        int totalCalorias( Fruta * f ) {
                return (f->gramas)*(f->caloriasPorGrama);
        }
};
```

- Convenção: todas as funções que operam sobre uma variável do tipo de dados *Fruta* recebem um apontador para essa variável como primeiro argumento.

Linguagem C++

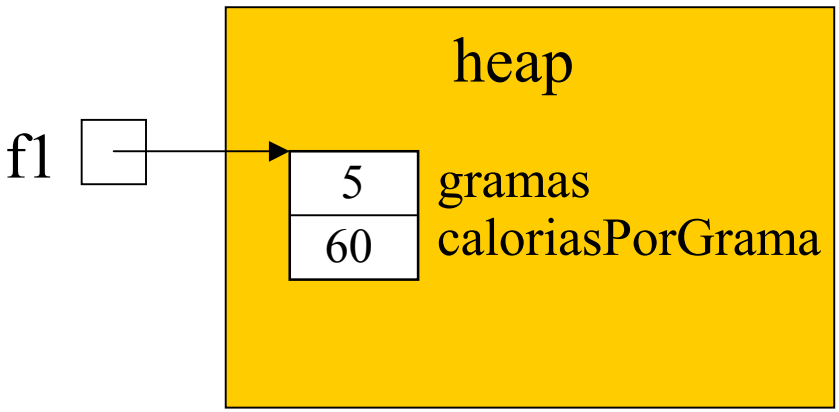
```
class Fruta {
        int gramas;
        int caloriasPorGrama;
        int totalCalorias() {
                return gramas*caloriasPorGrama;
        }
};
```

```
Fruta f1;
f1.gramas = 5;
f1.caloriasPorGrama = 60;
int calorias = f1.totalCalorias();
```

Linguagem Java

```
class Fruta {
    int gramas;
    int caloriasPorGrama;
    int totalCalorias() {
        return gramas*caloriasPorGrama;
    }
}
```

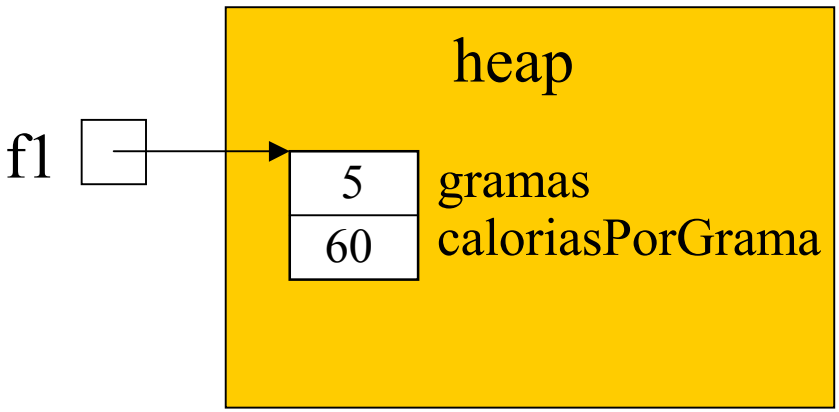
```
Fruta f1 = new Fruta();
f1.gramas = 5;
f1.caloriasPorGrama = 60;
int calorias = f1.totalCalorias();
```



Linguagem Java

```
class Fruta {
    int gramas;
    int caloriasPorGrama;
    int totalCalorias() {
        return gramas*caloriasPorGrama;
    }
}
```

```
Fruta f1;
f1 = new Fruta();
//  Fruta f1 = new Fruta();
f1.gramas = 5;
f1.caloriasPorGrama = 60;
int calorias = f1.totalCalorias();
```



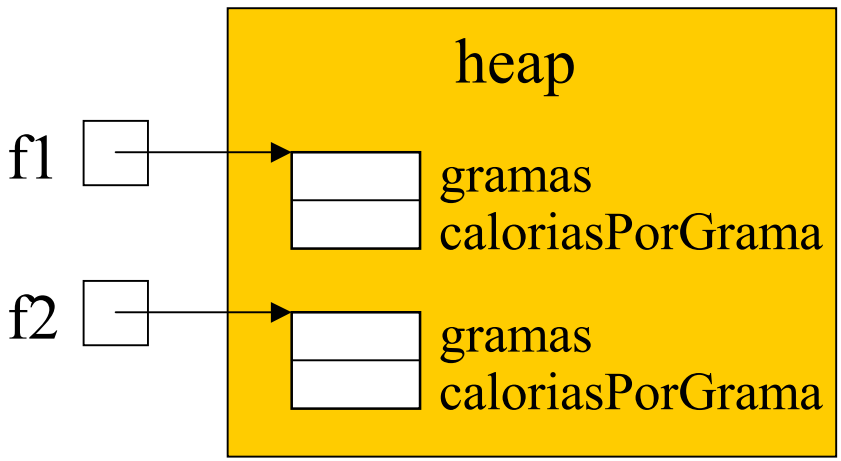
Variáveis de instância

```
class Fruta {
    int gramas;
    int caloriasPorGrama;
    int totalCalorias() {
        return gramas*caloriasPorGrama;
    }
}
```

Campos de dados, ou Variáveis de instância

Método de instância

```
Fruta f1 = new Fruta();
Fruta f2 = new Fruta();
```

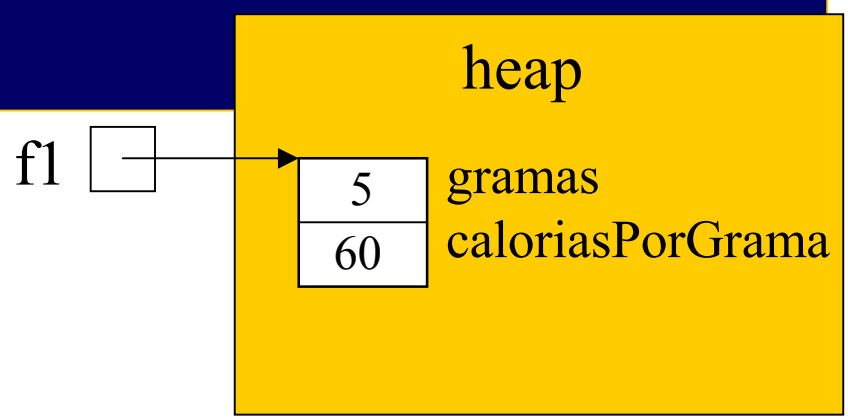


Construtores

- Servem para colocar o objecto criado num estado inicial.
- Nome igual ao da classe.
- Recebem 0 ou mais parâmetros.
- Não têm tipo de retorno.

```
class Fruta {
    int gramas;
    int caloriasPorGrama;
    Fruta(int g, int c) {
        gramas = g;
        caloriasPorGrama = c;
    }
    int totalCalorias() {
        return gramas*caloriasPorGrama;
    }
}
```

```
Fruta f1 = new Fruta(5, 60);
int calorias = f1.totalCalorias();
```



Construtor por omissão

- A maioria das classes têm pelo menos um construtor.
- Se não é definido nenhum explicitamente, o compilador cria automaticamente um construtor por omissão
  - construtor sem argumentos (*no-arg constructor*).
- É normal criar objectos com chamadas do tipo:
  - *Bicicleta b = new Bicicleta();*
  - *Cerveja cheers = new Cerveja();*
  - *Fruta f1 = new Fruta();*

```
class Fruta {
    int gramas;
    int caloriasPorGrama;
    Fruta() {
    }
    Fruta(int g, int c) {
        gramas = g;
        caloriasPorGrama = c;
    }
    int totalCalorias() {
        return gramas*caloriasPorGrama;
    }
}
```

2 maneiras de criar objectos:

```
Fruta  f1 = new Fruta();
Fruta  f2 = new Fruta(10, 80);
```

Inicialização de variáveis

- Variáveis membros de classes são automaticamente inicializadas quando se criam objectos.
- A memória alocada no *heap* é preenchida com 0´s.
- Criação de objectos:
  - alocação de memória
  - colocação dos valores iniciais por omissão
  - execução dos seus inicializadores explícitos
  - execução dos construtores.

Variáveis de classe

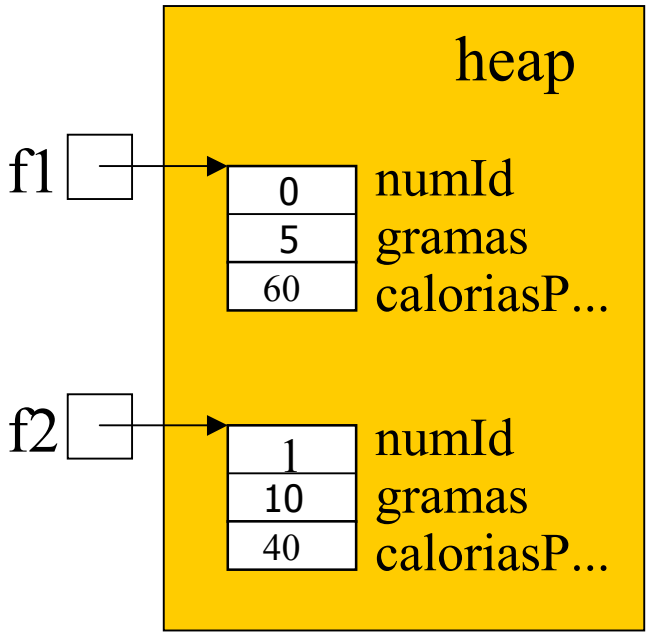
- Problema: atribuir um número único a cada objecto fruta criado.
- Mas após ser criado um objecto, continua a ser possível alterar o campo de dados *numId* !

```
class Fruta {
    int numId;
    int gramas;
    int caloriasPorGrama;
    static int proxNumId = 0;
    Fruta(int g, int c) {
        numId = proxNumId ++;
        gramas = g;
        caloriasPorGrama = c;
    }
    int totalCalorias() {
        return gramas*caloriasPorGrama;
    }
}
```

Modificadores de acesso

- Membros declarados como:
  - private - só podem ser acedidos por código dentro da classe.
  - public - podem ser acedidos de fora da classe.
- Para não permitir que código fora da classe possa alterar um campo de dados
  - devemos declarar o campo de dados privado.
- Para poder ler o valor do campo de dados de fora da classe
  - devemos acrescentar à classe um método público de acesso que retorne o valor do campo de dados.

```
public class Fruta {
   private int numId;
   private int gramas;
   private int caloriasPorGrama;
   private static int proxNumId = 0;
   public Fruta(int g, int c) {
        numId = proxNumId ++;
        gramas = g;
        caloriasPorGrama = c;
   }
   public int getNumId() {
        return numId;
   }
   public int totalCalorias() {
        return gramas*caloriasPorGrama;
   }
}
```



Fruto n.º 0 = 300 cal.
Fruto n.º 1 = 400 cal.

```
Fruta f1 = new Fruta(5, 60);
Fruta f2 = new Fruta(10, 40);
System.out.println(
  "Fruto n.º "+f1.getNumId()+"="+f1.totalCalorias() + " cal. \n"+
  "Fruto n.º "+f2.getNumId()+"="+f2.totalCalorias() + " cal. \n");
```

Integridade dos tipos de dados

- O Encapsulamento permite
  - Reforçar a integridade dos tipos de dados, não permitindo aos programadores o acesso aos campos de dados individuais de um modo inapropriado.
- Exemplo da criação da classe Data
  - Objectos com 3 atributos: ano, mês e dia.
  - Só valores válidos de **mês (1..12)**,

    e **dia (1..31 mas dependente do mês e do ano)**.

```
public class Tempo {
    private int hora;              // 0 - 23
    private int minuto;            // 0 - 59
    private int segundo;           // 0 - 59
    public Tempo()  { }
    public Tempo( int h, int m, int s )  {
        setHora( h );  setMinuto( m ); setSegundo( s );
    }
    public void setHora( int h ) {
        hora = ( ( h >= 0 && h < 24 ) ? h : 0 );
    }
    public void setMinuto( int m ) {
        minuto =( ( m >= 0 && m < 60 ) ? m : 0 );
    }
    public void setSegundo( int s ) {
        segundo =( ( s >= 0 && s < 60 ) ? s : 0 );
    }
    public int getHora()       {  return hora;          }
    public int getMinuto()     {  return minuto;        }
    public int getSegundo()    {  return segundo;       }
}
```

Métodos de acesso

- Métodos que regulam o acesso a dados internos designam-se por *métodos de acesso*.
- Normalmente os campos de dados de uma classe declaram-se como *private*.
- Adicionam-se métodos para colocar (*set*) e retribuir (*get*) os valores desses campos de dados.
- Esses métodos devem verificar a consistência dos dados só permitindo alterações se adequadas.

```
public class Data {
  private int ano;                    // qualquer ano
  private int mes;                    // 1-12
  private int dia;                    // 1-31 mas dependente do mês
  private static int [] diasPorMes = {
          0,31,28,31,30,31,30,31,31,30,31,30,31
  };
  private static String [] nomeMes = { "Invalido",
           "Janeiro", "Fevereiro", "Março", "Abril",
           "Maio", "Junho", "Julho", "Agosto",
           "Setembro","Outubro", "Novembro", "Dezembro"
  };
  public Data( int a, int m, int d ) {
      ano = a;
      if ( m > 0 && m <= 12 ) mes = m;       // valida o mês
      else {     mes = 1;
          System.out.println( "Mês " + m + " inválido. Colocado 1.");
      }
      dia = validaDia( d );                  // valida o dia
  }
```

```
  // Confirma o valor do dia baseado no mês e ano.
  private int validaDia( int d ) {
      if ( d > 0 && d <= diasPorMes[ mes ] ) return d;
      // se Fevereiro: Verifica se ano bissexto
      if ( mes == 2 && d == 29 && anoBissexto(ano) ) return d;
      System.out.println( "Dia " + d + " inválido. Colocado 1." );
      return 1;   // Deixa o objecto num estado consistente
  }

  private static boolean anoBissexto(int a) {          //Ano bissexto.
      return  ( a % 400 == 0 || a % 4 == 0 && a % 100 != 0 ) ;
  }

  // Cria uma String da forma "dia de mês de ano"
  public  String  porExtenso() {
      return dia + " de " + nomeMes[mes] + " de " + ano;
  }

  public String toString() {
      return  ano + "/" + mes + "/" + dia;
  }
}
```

Vantagens do encapsulamento

- Modularidade - agregando dados e comportamentos numa única unidade.
- Privilégios idênticos aos dos tipos de dados primitivos.
- Protecção dos dados - garantindo a sua consistência e segurança.

Vantagens do encapsulamento

- Separa a implementação da interface.
  - Simplifica a *percepção* que os utilizadores têm da classe - nível de abstracção.
  - Facilita a possibilidade de *modificação da implementação* da classe.
- Independência do contexto - promove o desacoplamento.
- Melhor qualidade na produção de software.

Herança

- Reutilização de código:
  - Aproximação tradicional - consiste em copiar e adaptar.
  - Aproximação da POO - consiste na criação de classes que usam outras classes já existentes.
- Na POO há 2 modos de reutilização de código:
  - Composição - dentro de uma classe criam-se objectos de classes já existentes.
  - Herança - cria-se uma nova classe como um tipo de uma classe existente.

Reutilização de Código por Composição

```
public class Empregado {
    private String primNome;
    private String ultNome;
    private Data dataNasc;
    private Data dataInicio;

    public Empregado(String pN, String uN, Data dN, Data dI) {
        primNome = pN;
        ultNome = uN;
        dataNasc = dN;
        dataInicio = dI;
    }

    public String toString()  {
        return primNome + " " + ultNome + " nascido a " +
                    dataNasc + " e contratado em: " + dataInicio;
    }
}
```

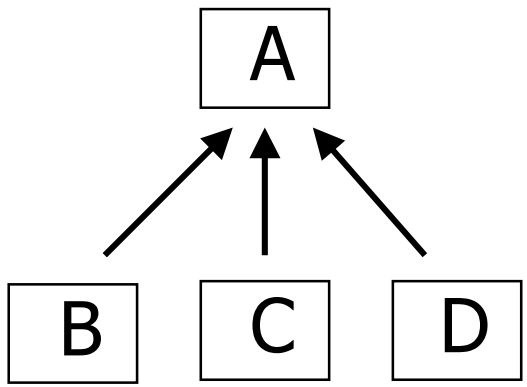
Reutilização de Código por Herança

```
public class  Ponto  {
    double x, y;
    public  Ponto(int x, int y) {
        this.x = x;
        this.y = y;
        System.out.println("Ponto (" + x + ", " + y + ")");
    }
}
```

```
import java.awt.Color;
public class  Pixel  extends  Ponto  {
    Color cor;
    public Pixel() {
        super(0,0);
        cor = null;
    }
}
```

Herança

- Herança é uma forma de reutilizar software
  - Novas classes são criadas a partir de classes existentes,
  - Herdando os atributos e comportamentos, e
  - Acrescentando novos atributos e comportamentos que necessitem.
- Herança é utilizada para especialização
  - Todo o objecto da subclasse é também um objecto da superclasse.
  - O inverso não é verdade.



Reescrita (*Overriding*) de Métodos

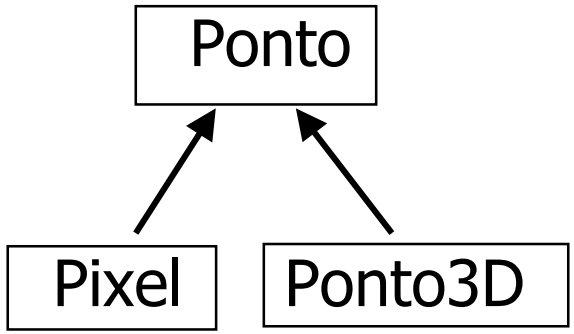
- Uma subclasse começa idêntica à superclasse mas tem a possibilidade de definir adições ou substituições das características herdadas da superclasse.

```
public class  Ponto  {
    double x, y;
    public void limpar() {
        x = 0;
        y = 0;
    }
}
```

```
import java.awt.Color;
public class  Pixel  extends  Ponto  {
    Color cor;
    public void limpar() {
        super.limpar();
        cor = null;
    }
}
```

Conversão entre Referências de Objectos

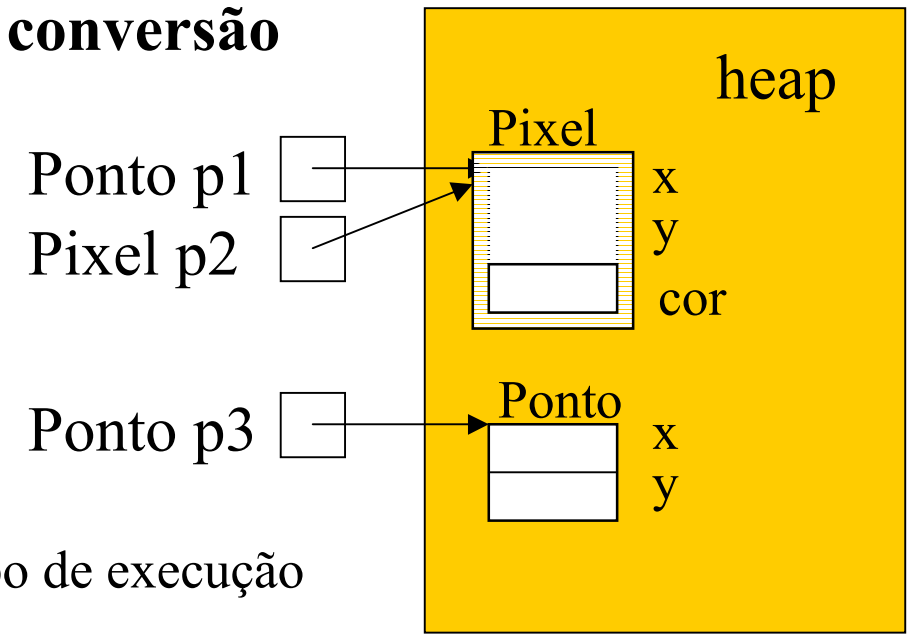
- Java efectua **conversão implícita** de uma subclasse para superclasse (*upcasting*)
  - Ex.: Ponto p1 = new Pixel();



- A conversão para uma subclasse só é possível se o objecto é realmente um objecto da subclasse, e é necessário efectuar **conversão explícita** (*downcasting*)
  - Ex.:

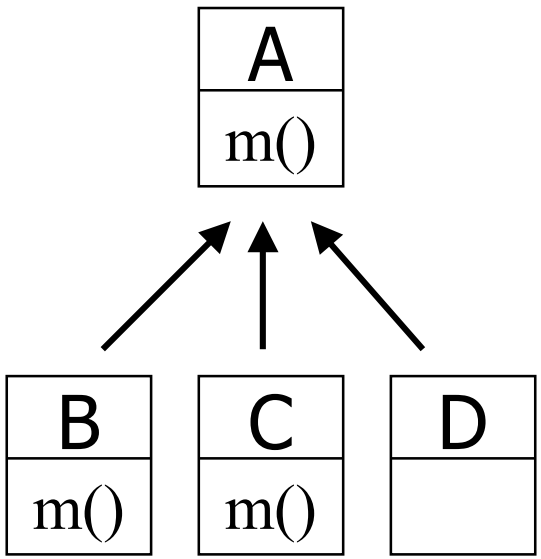
```
Ponto p1 = new Pixel();
Pixel p2 = (Pixel) p1;

Ponto p3 = new Ponto();
Pixel p4 = (Pixel) p3;    // erro em tempo de execução
```



Polimorfismo

- Significa que através do mesmo nome se pode invocar diferentes métodos.
- O mecanismo de polimorfismo permite que:
  - usando uma referência de uma superclasse para referenciar um objecto, instância de uma subclasse, e
  - invocando um método que exista na superclasse e que também exista (*overriden*) na subclasse do objecto,
  - o programa escolherá dinamicamente (isto é, em tempo de execução) o método correcto da subclasse.



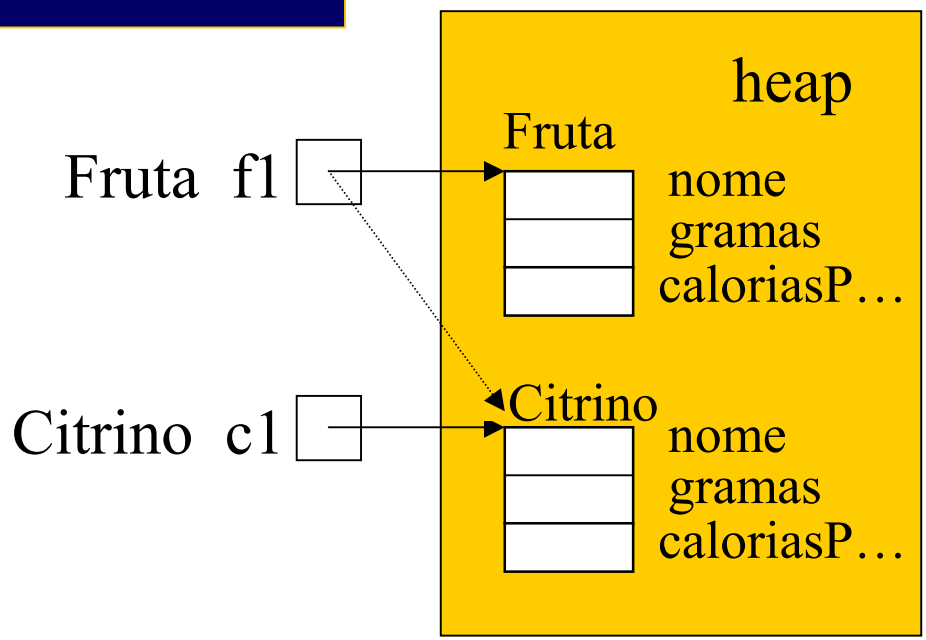
```
public class Fruta {
  private String nome;
  private int gramas;
  private int caloriasPorGrama;
  public Fruta(String n, int g, int c) {
      nome = n;
      gramas = g;
      caloriasPorGrama = c;
  }
  public void descascar() {
      System.out.println("Descascar um Fruto");
  }
}
```

```
public class Citrino extends Fruta {
  public Citrino(String n, int g, int c) {
      super(n, g, c);
  }
  public void descascar() {
      System.out.println("Descascar um Citrino");
  }
}
```

Polimorfismo

```
Fruta f1 = new Fruta("Pera", 130, 1);
Citrino c1 = new Citrino("Laranja", 200, 5);
f1.descascar();
c1.descascar();
f1 = c1;
f1.descascar();
```

Descascar um Fruto
Descascar um Citrino
Descascar um Citrino



Exemplo de Polimorfismo

- Programa para calcular os vencimentos mensais de diferentes tipos de trabalhadores:
  - Trabalhadores à comissão (TrabCom), cujo vencimento é igual a um salário base mais uma percentagem das vendas;
  - Trabalhadores à peça (TrabPeca), com um vencimento proporcional ao número de peças produzidas; e
  - Trabalhadores à hora (TrabHora), com um vencimento proporcional às horas de trabalho.

Superclasse Trabalhador

```
public abstract class Trabalhador{
  private String primNome;
  private String ultNome;
  public Trabalhador(String pNome, String uNome )  {
      primNome = pNome;
      ultNome = uNome;
  }
  public String toString() {
       return  primNome + " " + ultNome;
  }
  public abstract double vencimento();
}
```

Subclasse TrabCom

```
public class TrabCom extends Trabalhador {
  private double salario;                  // salário base
  private double comissao;                 //percentagem das vendas
  private double quantidade;               // total de vendas
  public TrabCom( String pNome, String uNome,
                  double sal, double com, double quant) {
      super( pNome, uNome );               // invoca constr. da superclasse
      salario = sal;
      comissao = com;
      quantidade = quant;
  }
  public double vencimento() {
      return salario + comissao * quantidade;
  }
}
```

Subclasse TrabPeca

```
public class TrabPeca extends Trabalhador {
  private double pagPeca;              // pagamento por peça
  private int quantidade;              // quantidade de peças
  public TrabPeca( String pNome, String uNome,
                                       double pP, int quant ) {
      super( pNome, uNome );           // invoca constr. da superclasse
      pagPeca = pP;
      quantidade = quant;
  }
  public double vencimento() {
          return quantidade * pagPeca;
  }
}
```

Subclasse TrabHora

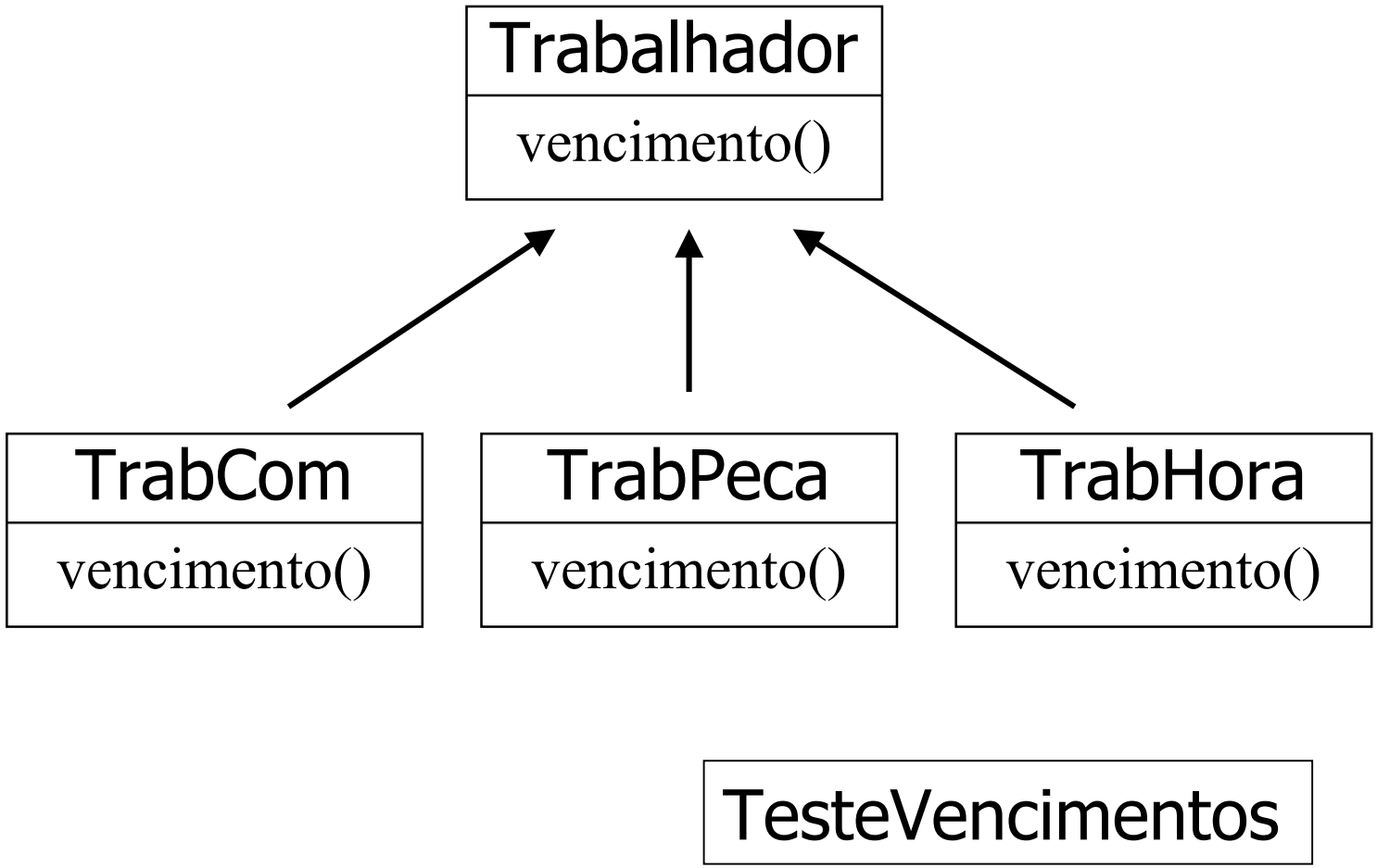
```
public class TrabHora extends Trabalhador {
  private double pagHora;                  // pagamento por hora
  private double horas;                    // horas de trabalho
  public TrabHora ( String pNome, String uNome,
                                    double ph, double hs ) {
      super( pNome, uNome );               // invoca constr. da superclasse
      pagHora = ph;
      horas = hs;
  }
  public double vencimento()  {
          return pagHora * horas;
  }
}
```

Classe TesteVencimentos

```
public class TesteVencimentos {
  public static void main(String []args)  {
      TrabCom tc = new TrabCom("Jorge","Silva",400.0, 0.06, 150.0);
      TrabPeca tp = new TrabPeca("Miguel", "Mendes", 2.5, 200);
      TrabHora th = new TrabHora("Carlos", "Miguel", 3.0, 160);
      Trabalhador[] a = new Trabalhador[3];
      a[0] = tc;
      a[1] = tp;
      a[2] = th;
      for (int i=0; i<a.length; i++)
          if (a[i] != null)
              System.out.println( a[i] + " tem o vencimento de "
                                          + a[i].vencimento() );
  }
}
```

```
Jorge Silva tem o vencimento de 409.0
Miguel Mendes tem o vencimento de 500.0
Carlos Miguel tem o vencimento de 480.0
```

Hierarquia de classes



Benefícios do Polimorfismo

- Tratamento genérico de código
  - numa hierarquia de tipos com o mesmo interface (os mesmos métodos públicos), código que funciona com um tipo genérico também funcione com qualquer objecto que seja subtipo desse tipo.
- Permite uma fácil extensão de um programa com um mínimo de modificações
  - podem-se adicionar novos tipos de objectos que respondem a mensagens existentes.
- Simplifica a escrita de código, a leitura e compreensão e a manutenção - nível de abstracção.

Comparação com a Programação Procedimental

- Programação Orientada por Procedimentos
  - identificar as tarefas a ser realizadas
  - decompõe-se em subtarefas
  - Para problemas pequenos funciona bem
- Programação Orientada por Objectos
  - primeiro isolam-se as classes,
  - só depois se identificam os métodos de cada classe.
  - Para problemas grandes tem vantagens:
    - classes são um bom mecanismo de agrupamento de métodos
    - classes escondem a representação dos seus dados

Conclusão

- A programação orientada por objectos permite implementar quatro conceitos fundamentais: abstracção, encapsulamento, herança e polimorfismo.
- Os módulos do programa – classes – agrupando dados e comportamentos podem ser desenvolvidos com características e funcionalidades independentes do contexto
  - reduz a dependência dos dados,
  - facilita a detecção de erros,
  - facilita a reutilização de código, e
  - melhora a qualidade da produção de software.

Conclusão (Encapsulamento)

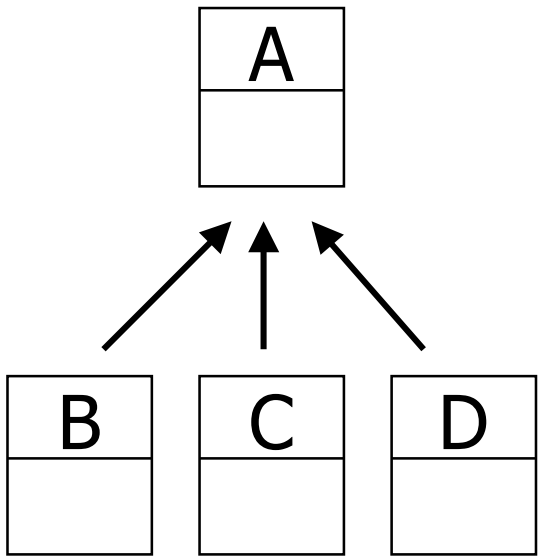
- O encapsulamento consiste:
  - Na inclusão de dados e métodos dentro de classes, e
  - No controlo do acesso aos membros da classe.
- Através do encapsulamento é possível separar:
  - A estrutura e mecanismos internos do programa, da
  - Interface que deve ser utilizada pelo programador cliente.
- O encapsulamento
  - Facilita a compreensão e a reutilização de código,
  - Permite garantir a consistência e segurança dos dados.

Conclusão (Reutilização de Código)

- Para reutilizar código o programador apenas necessita saber como comunicar com os objectos não necessitando compreender o seu funcionamento.
- Dois modos de reutilização de código:
  - **Composição** - usa-se quando se pretende criar uma classe composta por objectos de classes existentes.
  - **Herança** - usa-se quando se pretende especializar uma classe existente para um fim particular.
- Relação entre os objectos
  - ***tem-um*** (*has-a*) é expressa através da composição
    - Ex.: Uma circunferência tem um ponto que é o centro.
  - ***é-um*** (*is-a*) exprime-se através da herança
    - Ex.: Um carro é um veículo.

Conclusão (Herança)

- A herança é usada com 2 objectivos:
  - reutilização de código, e
  - implementação de polimorfismo.



Conclusão (Polimorfismo)

- Benefícios:
  - Tratamento genérico de código - código que funciona com um tipo genérico também funciona com qualquer objecto que seja subtipo desse tipo.
  - Permite a programação incremental e a extensibilidade dos programas - podem-se adicionar novos tipos de objectos que respondem a mensagens existentes sem modificar o sistema.
  - Simplifica a escrita de código, a leitura, compreensão e a manutenção.

Conclusão

- Programação Orientada por Objectos
  - Melhora a qualidade e economia da produção de software.
  - A lógica de organização do programa aproxima-se da visão humana sobre a forma como as entidades se relacionam no mundo real.
  - Facilita o desenvolvimento de aplicações reais.