POO – JAVA 21

PROGRAMAÇÃO ORIENTADA A OBJETO

Prof Alex Gondim

INSTITUTO FEDERAL
Bahia
Campus
Santo Antônio
de Jesus

INSTITUTO FEDERAL
Bahia
Campus
Santo Antônio
de Jesus

Apresentação

- Contato : 75 999331334
- Alex Gondim Lima
- agllima@ufba.br

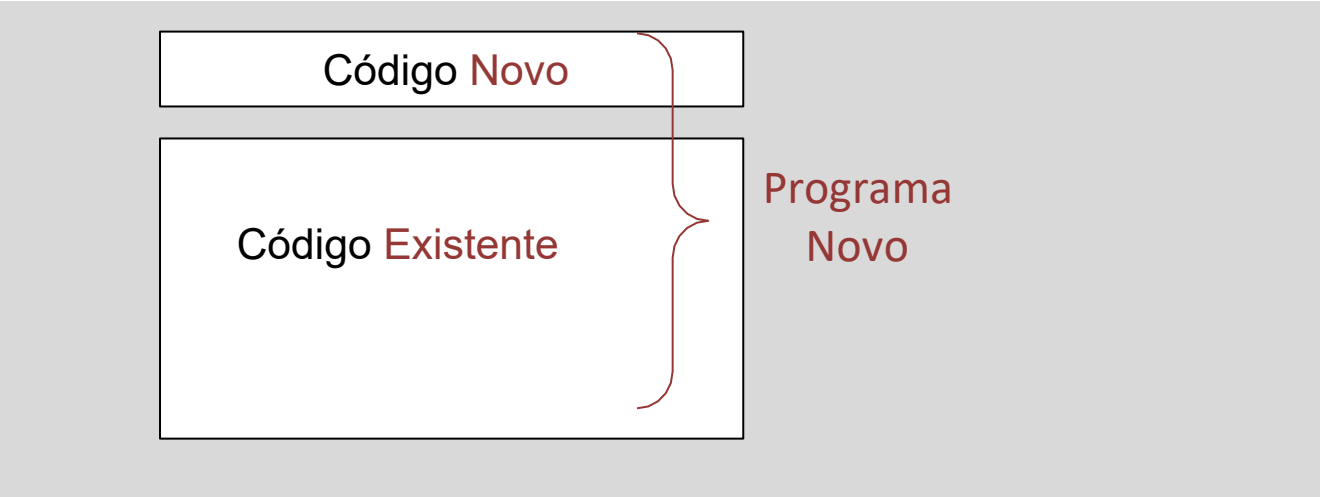
Relações entre Classes

Agregação
Composição
Associação

(Livro Big Java, Late Objects – Capítulo 12)

Interesse Geral das Relações entre Classes

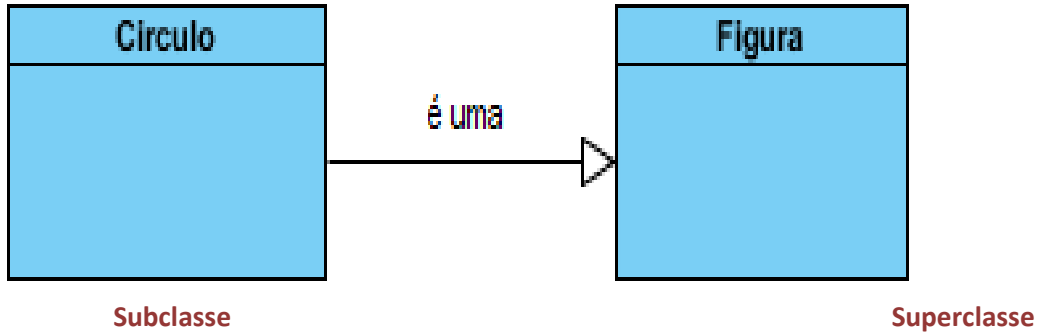
- **Reutilização de Código**
  - **Redução** do **esforço** de **programação** ⇒ **redução** de **custos** de produção de software
    - Uma das **vantagens** da POO
    - Como?
      - Programa **novo** obtido programando
        - Não todo o programa
        - Apenas uma **pequena** **parte nova** sobre código existente (reutilização)



        - Concretamente
          - Construção de **classes novas** a partir de **classes existentes**
            - ie., relacionando classes
              - **Objetos** de uma classe **usam serviços** fornecidos por **objetos** da outra

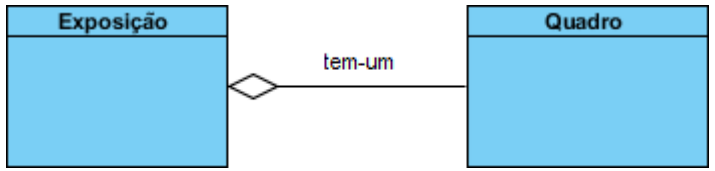
Relação de Herança

- **Conhecida**
  - Relação do tipo **“é-um”** // ou **“é-uma”**
- **Indica**
  - Uma classe **é uma** especialização/generalização de outra classe
- **Exemplo**
  - Notação UML



Relação de Agregação

- **Conhecida** // ou "**tem-uma**" ou "**é-parte-de**"
  - Relação do tipo "**tem-um**"
- **Indica**
  - Objeto de uma classe **contém** (ie., **agrega**) um objeto de outra classe ... e o objeto agregado **tem existência independente** do objeto agregador.
    ie., objeto agregado **pode existir** após **eliminação** do objeto agregador ie., objeto agregado **não pertence** ao objeto agregador
- **Exemplo**
  - Notação UML:



Classe Agregadora Classe Agregada

- **Classe Agregada faz parte da estrutura da Classe Agregadora**
  - Objeto agregado **é parte do** objeto agregador ⇒ guardado em **variável de instância** ⇒ classe **agregada** usada na **declaração** de variável de instância
- **Relação de Dependência Forte**
  - Uma classe **usa objeto** de outra classe ...
    ... na **estrutura** da classe/objeto

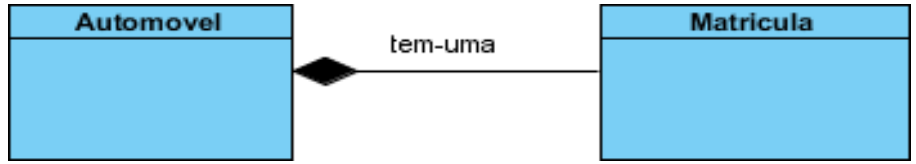
// relação de dependência
// forte

- **Relação de Agregação Fraca**
  - Objeto agregado **não pertence** ao objeto agregador

⇒ Objeto agregador tem **referência partilhada** do objeto agregado.

Relação de Composição

- **Relação de Agregação Forte**
  - Objeto de uma classe **contém** (ie., **agrega**) um objeto de outra classe ... e o objeto agregado **tem existência dependente** do objeto agregador.
    ie., objeto agregado **não pode existir** após **eliminação** do objeto agregador
    ie., **pertence** ao objeto agregador
- **Exemplo**
  - Notação UML



Classe Agregadora ou Classe Composta

Classe Agregada

- **Objeto Agregado**
  - **Pertence** ao objeto agregador ⇒ não tem **referência partilhada**

Implementação da Composição

- **Distinguir Tipos de Classes Agregadas**
  - Classes Mutáveis
    - Criam instâncias **mutáveis**, ie, com conteúdos **modificáveis** (usando **set**)
  - Classes Imutáveis
    - Criam instâncias **imutáveis**, ie., com conteúdos **não modificáveis**
    - Não disponibilizam métodos de modificação (set)
    - Exemplos
      - String, Integer, Double, Float
- **Se Classe Agregada é Mutável**
  - Classe agregadora
    - Não permite partilha de referências dos objetos agregados (**objetos mutáveis**)
    - Usa a **clonagem** (**cópia exata**) de instâncias
- **Se Classe Agregada é Imutável**
  - Comportamento de objetos imutáveis com referências partilhadas
    - Como pertencentes apenas à classe agregadora
    - Igual ao de objetos mutáveis sem partilha de referências
  - Classe Agregadora
    - Permite a partilha de referências dos objetos agregados (**objetos imutáveis**)

Classe Agregadora permite Referências Partilhadas de Objetos Agregados

- **Aplicação**
  - Agregação: usada em classes agregadas mutáveis e imutáveis
  - Composição: usada apenas em classes agregadas imutáveis
- **Exemplo**

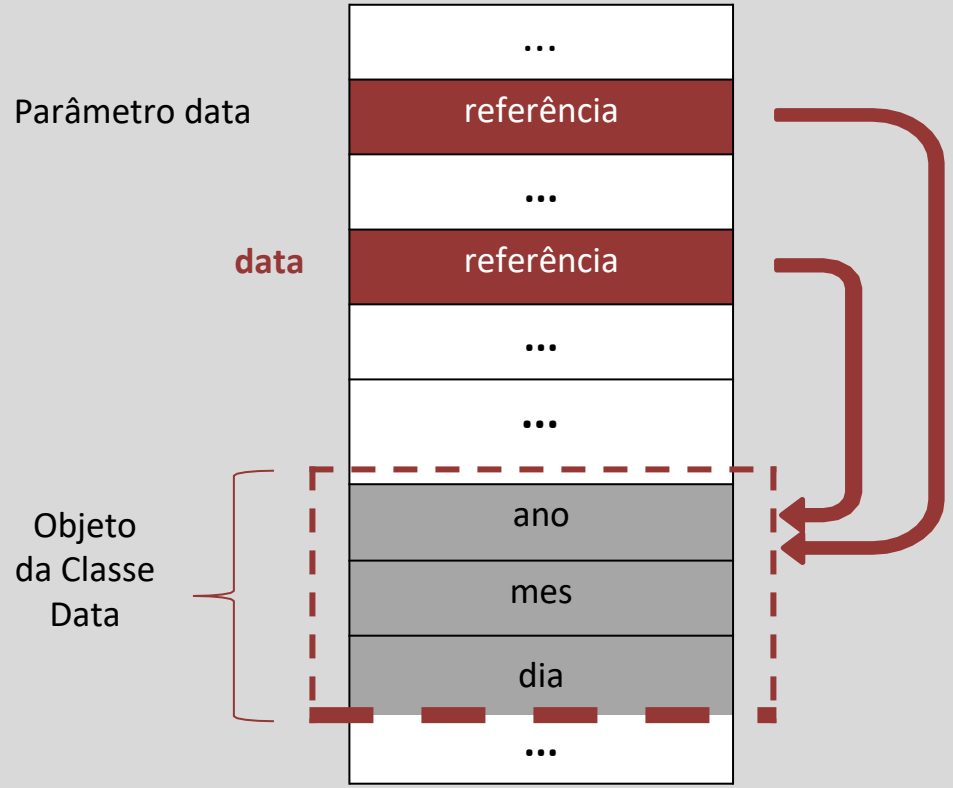
```
public class Demo {
    ...
    private Data data;                          // classe Data agregada à classe Demo (objeto Demo tem uma Data)
                                                    // objetos Data caracterizados por ano, mês e dia
                                                    // referência guardada pode ser partilhada com variável fora de obj
     ...
     public Demo( ..., Data data ){             // parâmetro data recebe cópia da referência de um objeto data
          ...
          this.data = data ;                    // data guarda a referência recebida ⇒
     }                                              // referência pode ser partilhada com variável fora de objeto Demo
     ...
     public Data getData() { return             // retorna referência guardada em data
          data;                                     // permite partilha da referência retornada ⇒ referência de data
     }
     ...
     public void setData( Data data ) {         // data guarda referência recebida
          this.data = data;                         // referência pode ser partilhada com variável fora de objeto Demo
     }
     ...
     public String toString() {
          return ... + " Data: " + data;
     }
```

Classe Agregadora permite Referências Partilhadas de Objetos Agregados

- **Exemplo**

```
public class Demo {
        ...
        private Data data;
        ...

        public Demo( ..., Data data ){
              ...
              this.data = data ;
        }

        ...
        public Data getData() { return data;
        }

         ...
        public void setData( Data data ) { this.data =
              data;
        }
        ...
}
```

Modelo de Memória RAM (Tempo de Execução - *RunTime*)



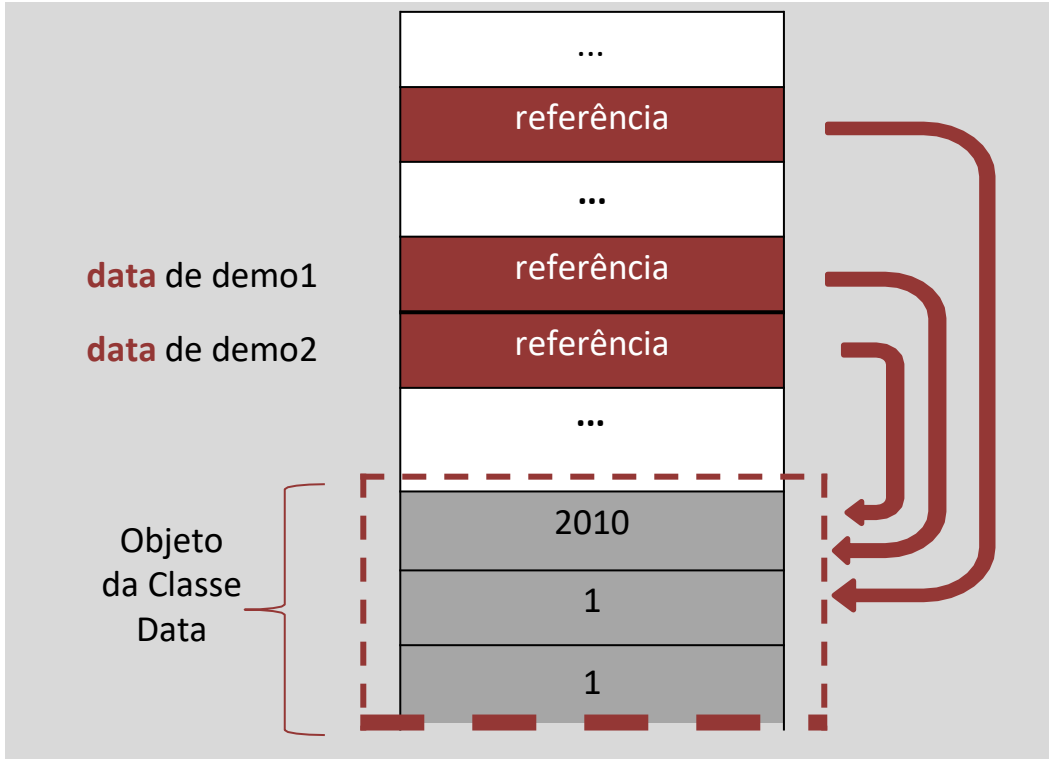
Classe Agregadora permite Referências Partilhadas de Objetos Agregados

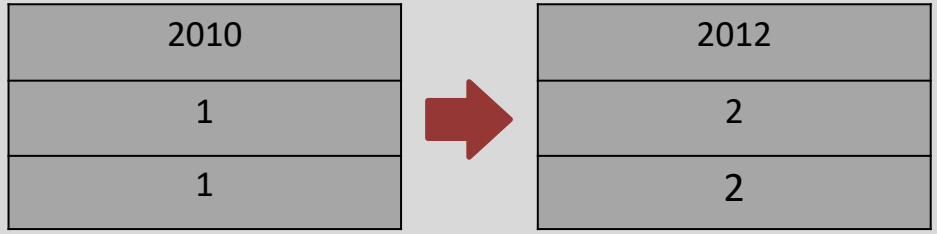
Exemplo

Dados de objetos Demo partilhados

```
public class TesteDemo {
    public static void main( String[ ] args ) {
        Data d = new Data(2010, 1, 1);

        Demo demo1 = new Demo( ..., d );                    //2010-1-1
        System.out.println( demo1.getData() );
        Demo demo2 = new Demo( ..., d );
        System.out.println( demo2.getData() );              //2010-1-1

        d.setData(2012, 2, 2);      // modifica demo1 e demo2

        System.out.println( demo1.getData() );              // 2010-2-2
        System.out.println( demo2.getData() );              // 2010-2-2
    }
}
```

Modelo de Memória RAM (em Execução)





Modificação da data d

Classe Agregadora permite Referências Partilhadas de Objetos Agregados

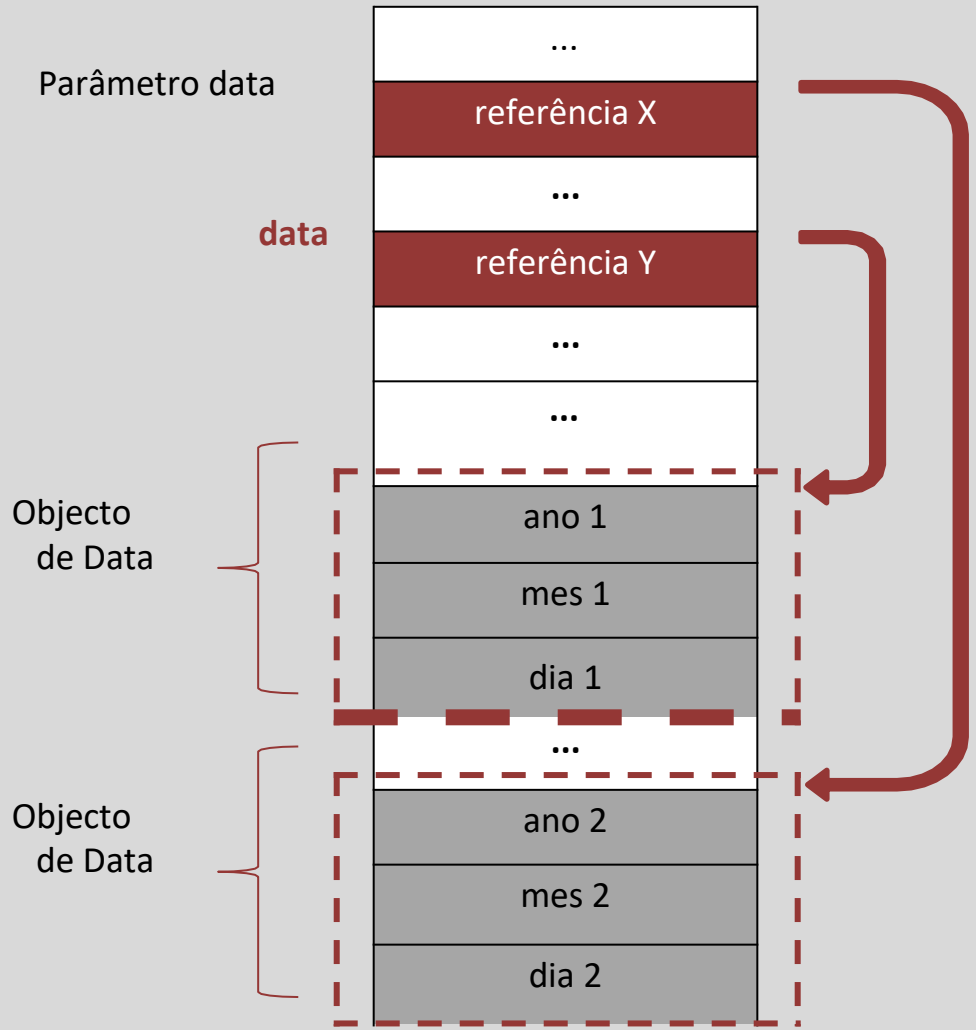
- **Aplicação**
  - Composição: em classes mutáveis
- **Exemplo**

```
public class Demo {
    ...
    private Data data;                          // guarda referência não partilhada ... com variável fora de objeto Demo
    public Demo( ..., Data data ){
        ...
        this.data = new Data( data );           // construtor de cópia da classe Data
        ...                                         // cria objeto clone do objeto data recebido
    }                                               // data e data são 2 objetos iguais
    ...                                             // data guarda nova referência de Data
    public Data getData() {
        return new Data( data );                // retorna referência de novo objeto Data, clone de data
    }                                               // não retorna referência guardada em data
    ...                                             // mantém referência não partilhada em data
    public void setData( Data data ){
        this.data.setData( data.getAno(),           // setData da classe Data modifica apenas conteúdo da data
                           data.getMes(),               // não cria novo objeto Data em cada modificação
                               data.getDia() );    // poupa memória
    }                                               // mantém referência não partilhada em data
    ...
    public String toString(){
        return ... + " Data:" + data;
    }
}
```

- **Exemplo**

```
public class Demo {
        ...
        private Data data;

        public Demo( ..., Data data ){

              ...

               this. data = new Data( data );
        }
        ...
        ...
        public Data getData() {
              return new Data( data );
        }
        ...
        public void setDataRegisto( Data data ) {

                                        data.setData( data .getAno(),
                                                      data .getMes(),
                                                      data .getDia() );
        }
        ...
}
```

Modelo de Memória RAM (Tempo de Execução)



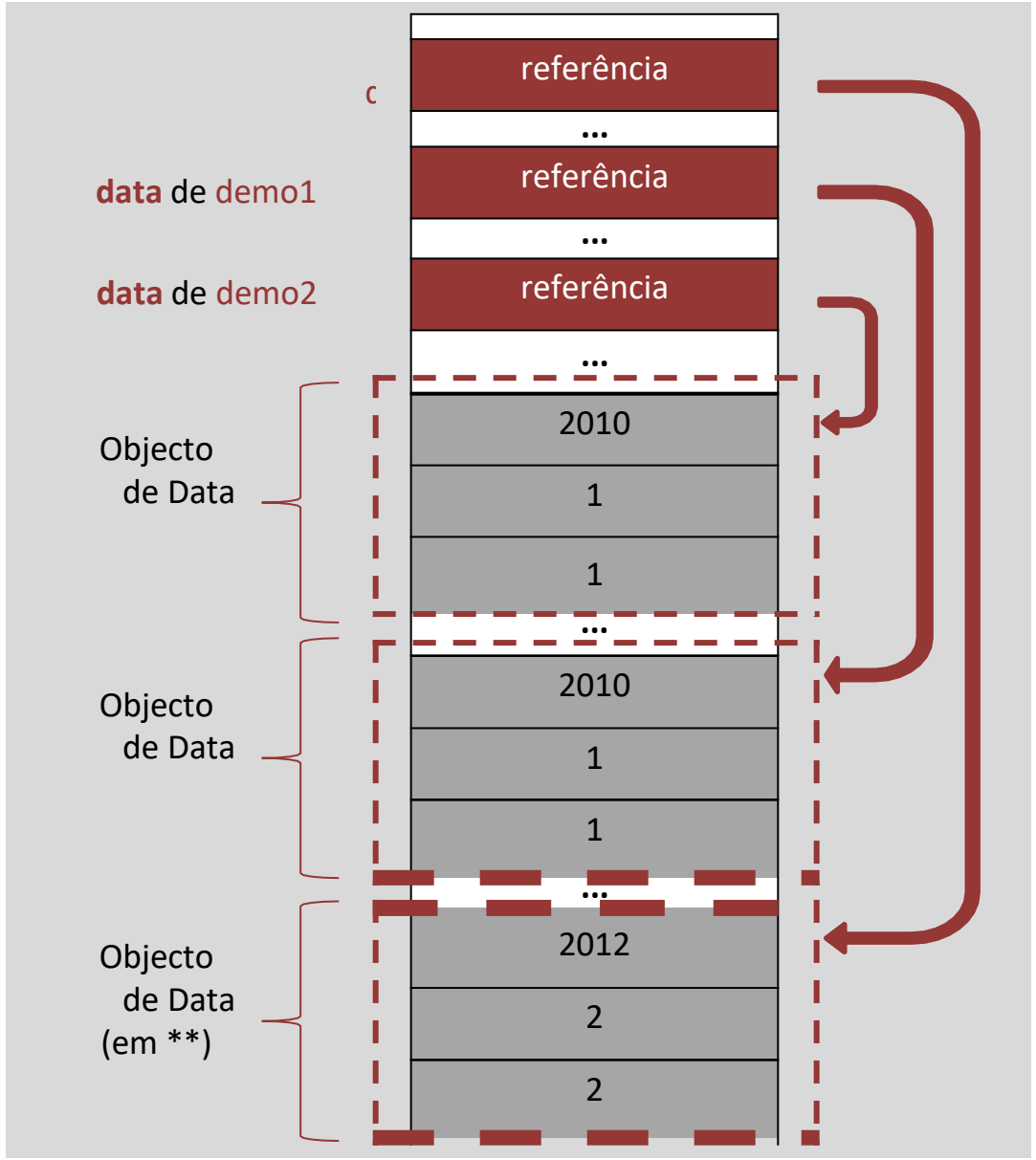
Classe Agregadora não permite Referências Partilhadas de Objetos Agregados

- **Exemplo**

Dados de objetos Demo **não** partilhados

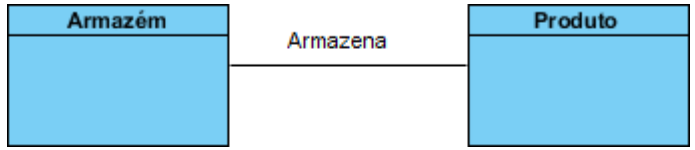
```
public class TesteDemo {
    public static void main( String[ ] args ) {
        Data d = new Data(2010, 1, 1);
        Demo demo1 = new Demo( ..., d );
        System.out.println( demo1.getData() );          // 2010-1-1
        Demo demo2 = new Demo ( ..., d );
        System.out.println( demo2.getData() );          // 2010-1-1
(**)    d.setData(2012, 2, 2);
        System.out.println( demo1.getData() );          // 2010-1-1
        System.out.println( demo2.getData() );          // 2010-1-1
        //      ------------------------------------------------------
        d = demo1.getData();            // não modifica demo2
        d.setData(1998,5,5);            // não modifica demos 1 e 2
}
}
```

Modelo de Memória RAM (em Execução)



Relação de Associação

- **Indica**
  - Alguma ligação relevante entre instâncias das classes
    - Navegação de um objeto de uma classe para outro objeto da outra classe
- **Exemplo**
  - Notação UML



Relação

- Mais **genérica** que as relações de agregação e composição
  - Relação de associação **fraca** (significado vago)
- Identificada
  - Numa fase inicial da análise e desenho
  - Na descoberta de dependências genéricas entre abstrações
- Refinada frequentemente
  - Numa relação mais concreta (agregação ou composição)
  - Numa fase mais avançada da análise

Threads e Concorrência em Java .

INTRODUÇÃO

Java é uma linguagem de programação multithread , o que significa que podemos desenvolver programas multithread usando Java. Um programa multithread contém duas ou mais partes que podem ser executadas simultaneamente e cada parte pode lidar com uma tarefa diferente ao mesmo tempo, fazendo uso ideal dos recursos disponíveis, especialmente quando seu computador tem várias CPUs .

Por definição, multitarefa é quando vários processos compartilham recursos de processamento comuns, como uma CPU. Multithreading estende a ideia de multitarefa para aplicativos onde você pode subdividir operações específicas dentro de um único aplicativo em threads individuais. Cada um dos threads pode ser executado em paralelo. O SO divide o tempo de processamento não apenas entre diferentes aplicativos, mas também entre cada thread dentro de um aplicativo.

Multithreading Java

O multithreading permite que você escreva de uma forma em que múltiplas atividades podem prosseguir simultaneamente no mesmo programa. Para atingir o multithreading (ou escrever código multithreaded), você precisa da classe java.lang.Thread .

Ciclo de vida de um thread em Java multithreading

Um thread passa por vários estágios em seu ciclo de vida. Por exemplo, um thread nasce, é iniciado, executado e então morre. O diagrama a seguir mostra o ciclo de vida completo de um thread.

Introdução- Continuação

1. **O A maioria dos programas são escritos de modo seqüencial com um ponto de início (método `main()`), uma seqüência de execuções e um ponto de término.**

   1.1 Em qualquer dado instante existe apenas uma instrução sendo executada.

1. **O O que são threads ?**

   2.1 É um simples fluxo seqüencial de execução que percorre um programa.

1. **O Multithreading: o programador especifica que os aplicativos contêm *fluxos de execução* (threads), cada thread designando uma parte de um programa que pode ser executado simultaneamente com outras threads.**

Programação concorrente

- O Todo programa, sendo *monothread* ou *multithread*, inicia com a execução da thread principal.

- Mecanismos de sincronização e prioridade podem ser usados para controlar a ordem de execução de threads independentes e colaboradas.