Metaclasses

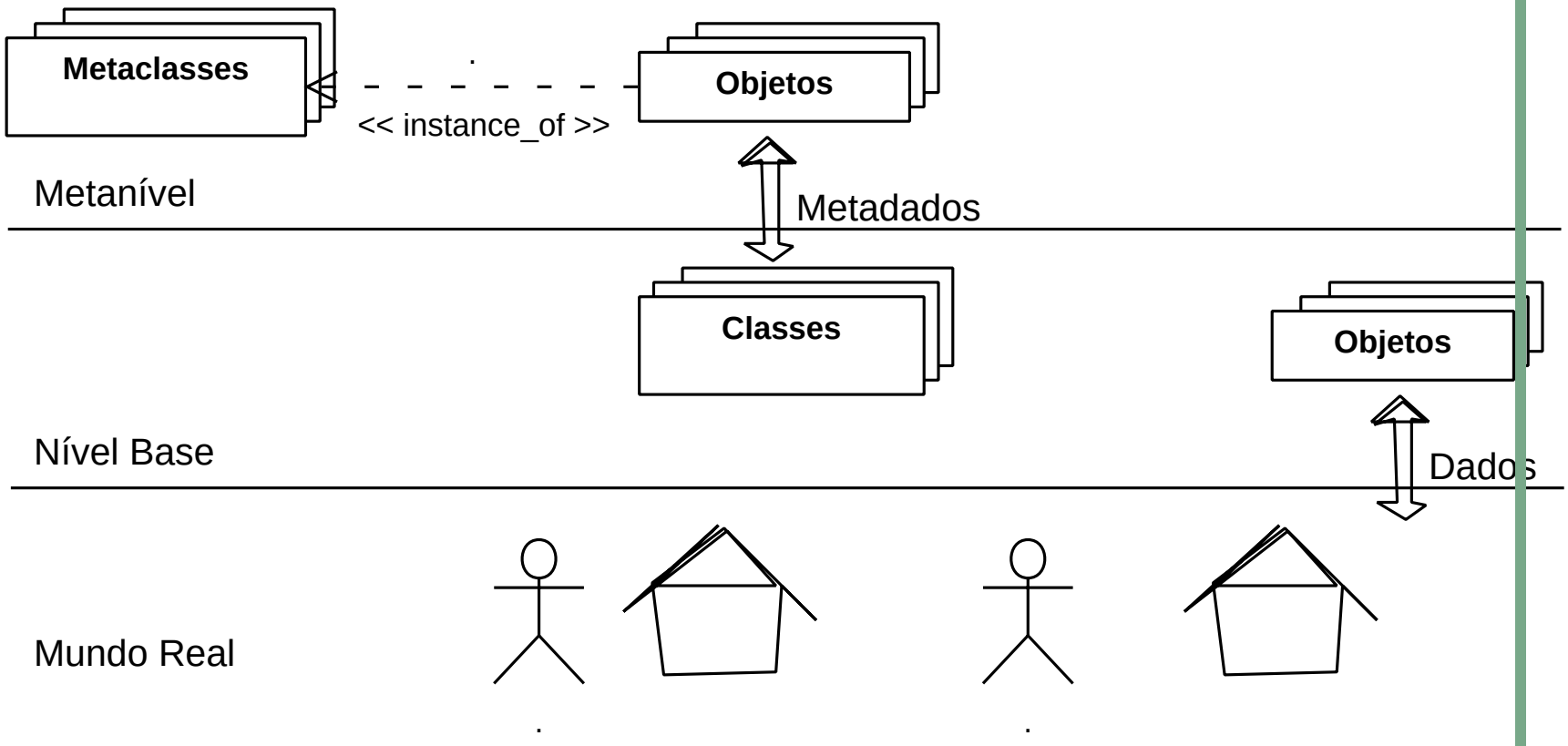
Metainformação

- **Metainformação** é um termo genérico aplicado a qualquer dado que descreve outro dado
- Em particular, na maioria das linguagens de progração orientadas a objetos, classes são vistas como "fábricas" que criam e inicializam instâncias (i.e. objetos)
- Em algumas linguagens orientadas a objetos, tal como Smalltalk-76, existem dois tipos de elementos geradores no modelo de objetos:
  - Metaclasses que geram classes (classes são objetos, instâncias de metaclasses)
  - Classes que geram objetos (objetos terminais são instâncias de classes)

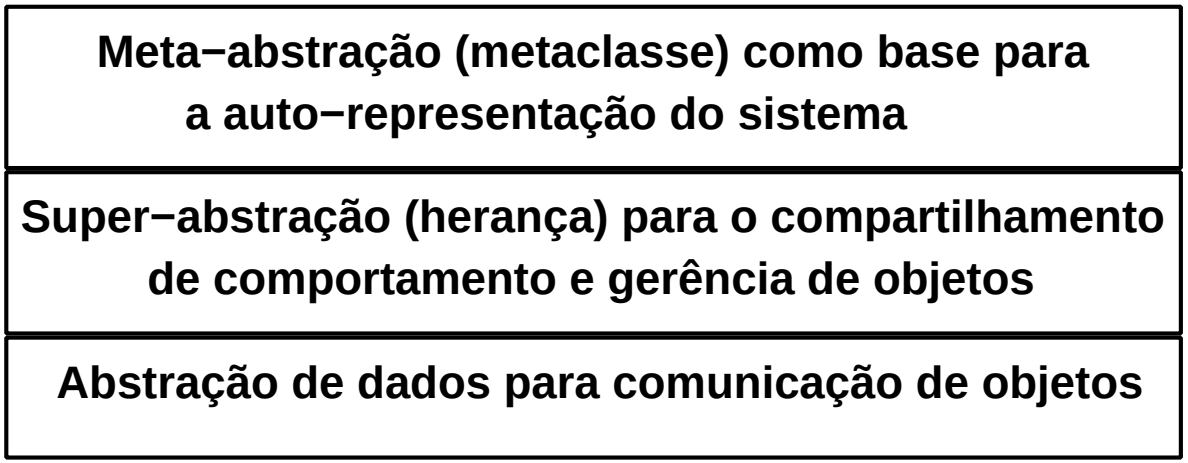
Metaclasses (I)

- Mais especificamente, **metaclasse** é uma classe que descreve outra classe, isto é, ela é uma classe cujas instâncias são classes
- Uma metaclasse guarda a metainformação de uma classe

Metaclasses (II)



Três Tipos de Abstrações no Modelo de Objetos

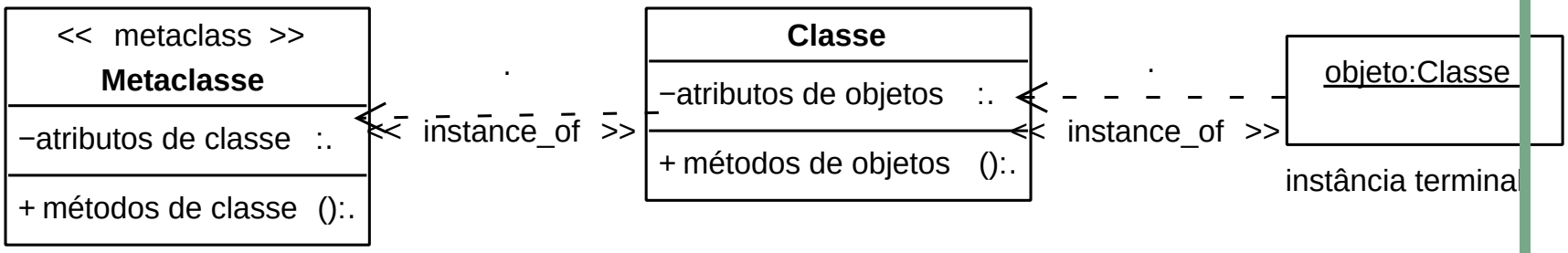


Vantagens

Os principais benefícios da representação de classes como objetos:

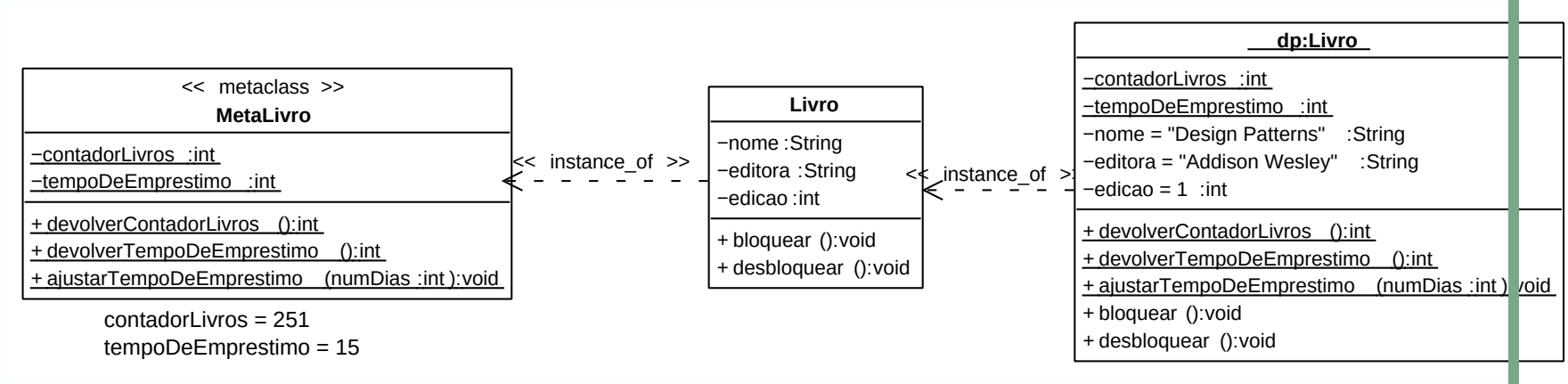
- informações globais relativas a todos os objetos de uma classe podem ser armazenadas nos **"atributos de classe"** (terminologia de Smalltalk)
- métodos associados com a metaclasse (chamados de **"métodos de classe"**) podem ser usados para recuperar e atualizar os valores desses atributos
- flexibilidade alcançada pela possibilidade de criação/iniciação dinâmica de novas classes, a partir das definições das metaclasses

Metaclasses



Exemplo 1

Metaclasse “MetaLivro” em UML



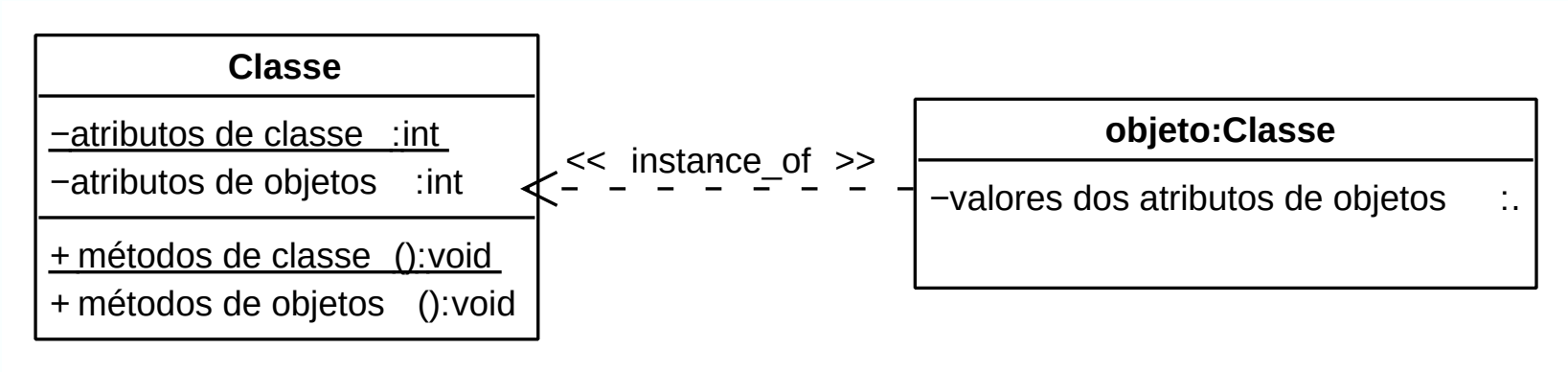
Notação UML para Metaclasses

- Metaclasses são representadas através de classes com o estereótipo $<< metaclass >>$
- UML também permite a especificação de métodos e atributos de classe. Estes aparecem sublinhados em um diagrama de classes comum

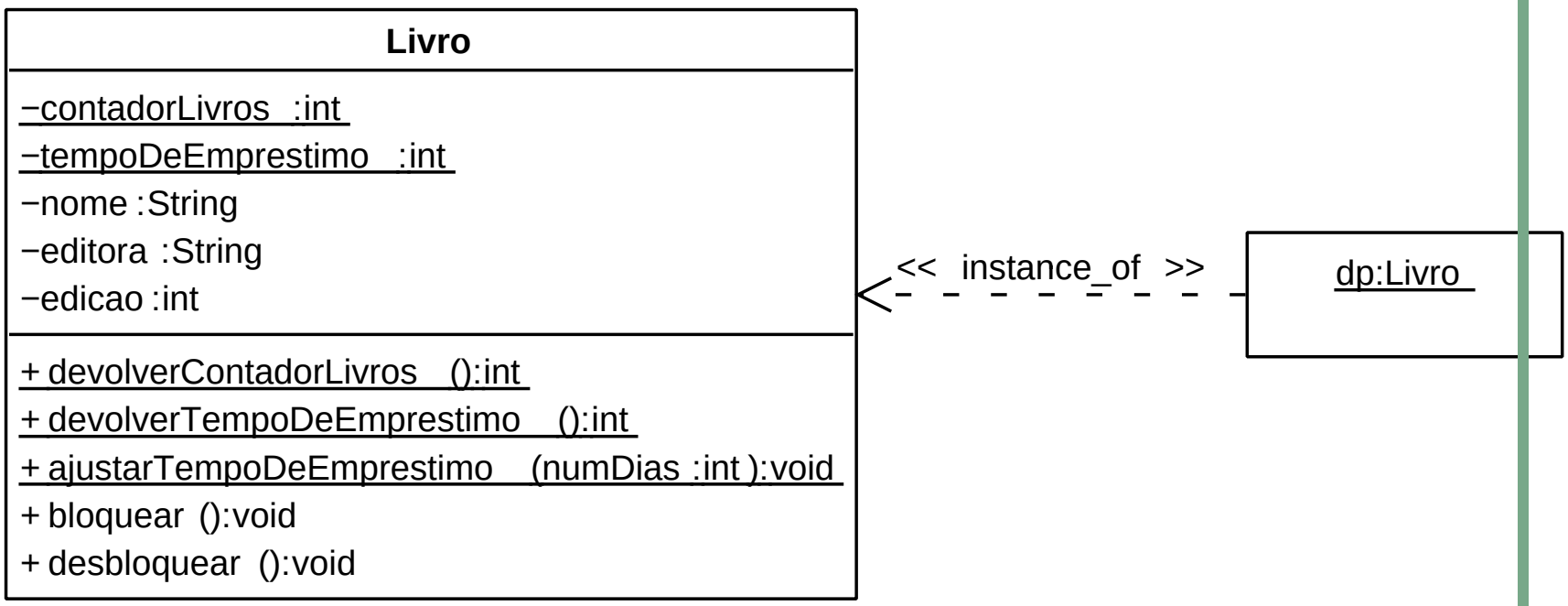
Metaclasses e Linguagens de Programação

- Smalltalk é uma das linguagens mais conhecidas que adotam o conceito de metaclasse
- Algumas linguagens orientadas a objetos, como C++ e Java, apesar de não darem apoio direto para a abordagem de "classes como instâncias de metaclasses", dão apoio para as noções de **"atributos e métodos de classes"** através de uma "fusão" dos conceitos de metaclasse e classe.

A “Fusão de Metaclasses e Classes”



Exemplo 2


Livro
−contadorLivros :int
−tempoDeEmprestimo :int
−nome :String
−editora :String
−edicao :int
+ devolverContadorLivros ():int
+ devolverTempoDeEmprestimo ():int
+ ajustarTempoDeEmprestimo (numDias :int):void
+ bloquear ():void
+ desbloquear ():void
<< instance_of >>
dp:Livro

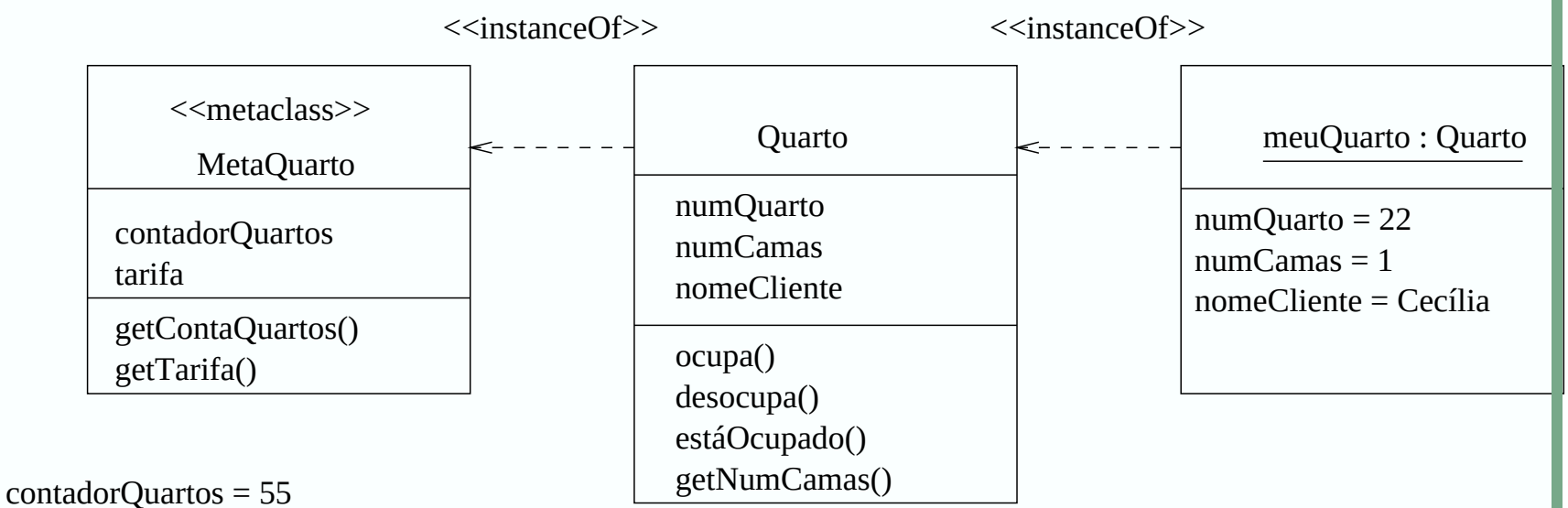
Metaclasses em Java

- Em Java, ao definirmos uma nova classe, é possível criarmos atributos da classe, que ficam associados à classe como um todo, e não aos objetos da classe.

```
static  <definição do método ou atributo>
```

- Há sempre um único valor para um atributo static, independente do número de objetos da classe, mesmo que não exista nenhuma instância da classe.
- Uma classe em Java que contenha apenas atributos static e métodos static, define uma metaclasse.

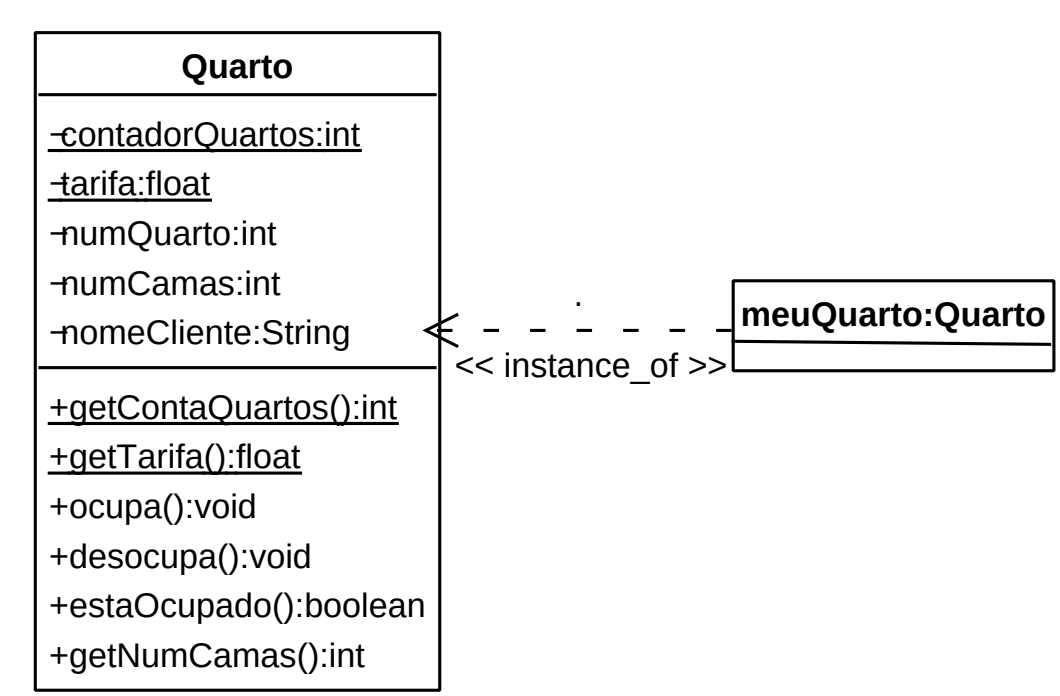
Metaclasse “MetaQuarto” - Exemplo 3(I)



contadorQuartos = 55

tarifa = R$ 150,00

Fusão MetaQuarto e Quarto - Exemplo 3(II)



Exemplo 3(III)

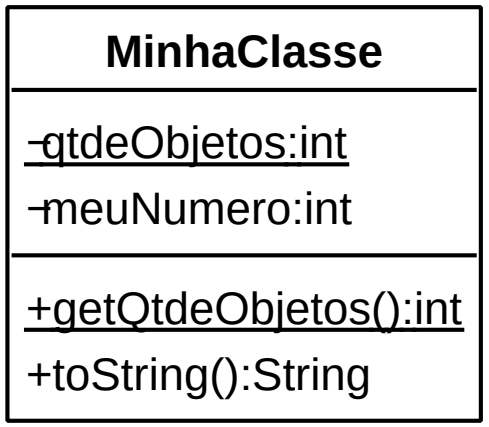
```
public class Quarto{
//atributos ‘‘de classe’’
   private static int contadorQuartos;
   private static float tarifa;
//atributos ‘‘de objetos’’
   private int numQuarto;
   private int numCamas;
   private String nomeCliente;

   ...
```

Exemplo 3 (IV)

```
   ...
//métodos ‘‘de classe’’
   public static int getContaQuartos(){
      return contadorQuartos;
   }
   public static float getTarifa(){
      return tarifa;
   }
//métodos de objeto
   public void ocupa(){...}
   public void desocupa(){...}
   public boolean estaOcupado(){...}
   public int getNumCamas(){...}
}//fim da classe Quarto
```

Exemplo 4 (I)

- Construção de uma classe chamada "`MinhaClasse`", com um atributo de classe `qtdeObjetos` para contar o total de objetos criados, e um método de classe `getQtdeObjetos`, que retorna esse valor.



Exemplo 4 (II)

```
class MinhaClasse {
    private static int qtdeObjetos = 0; //qtdeObejtos é
                                        //atrib.de classe
    private int meuNumero; // meuNumero  é atrib.de objeto
    public MinhaClasse( ) { // construtor
        qtdeObjetos++;
        meuNumero = qtdeObjetos; }
    static public int getQtdeObjetos ( ) {
        return qtdeObjetos; }
    public String toString ( ) {
        return ( ‘‘Objeto número’’+meuNumero); } }
```

```
class Exemplo {
    public static void main (String Arg [ ] ) {
        System.out.println(‘‘Qtde de objetos =’’
                          +MinhaClasse.getQtdeObjetos( ));
        MinhaClasse obj1 = new MinhaClasse ( );
        MinhaClasse obj2 = new MinhaClasse ( );
        MinhaClasse obj3 = new MinhaClasse ( );
        System.out.println(obj1.toString());
        System.out.println(obj2);
        System.out.println(obj3);
        System.out.println(‘‘Qtde de objetos =’’
                          +obj2.getQtdeObjetos( )) ;}
    } }
```

Exemplo 4 (III)

- Saída do Programa:

```
Qtde de objetos = 0
Objeto número 1
Objeto número 2
Objeto número 3
Qtde de objetos = 3
```

Construtor: Alternativa I

```
public MinhaClasse ( ) { // construtor
  qtdeObjetos++; // atributo de classe
  meuNumero = qtdeObjetos; // atributo de objeto
}
```

- O atributo qtdeObjetos pertence a classe MinhaClasse enquanto meuNumero pertence ao objeto que está sendo criado, contudo o código acima não permite notar esta diferença.

Construtor: Alternativa II

```
public MinhaClasse ( ) { // construtor
  MinhaClasse.qtdeObjetos++; //Atrib. de Classe
  meuNumero = MinhaClasse.qtdeObjetos;
                          //Atrib. de Instância
}
```

- Observe que os atributos são tratados como se pertencessem a um mesmo objeto.

Construtor: Alternativa III

```
public class MinhaClasse {
   int meuNumero;
   public MinhaClasse(int n) {meuNumero = n;}
   public String toString(){
      return (``Objeto número''+meuNumero);} }
```

- A variável `qtdeObjetos` é removida da classe `MinhaClasse`, e colocada numa nova classe chamada `MinhaFabrica`.
- `MinhaClasse` passa a ter apenas atributos e métodos de objetos (i.e. não estáticos).

Nova Classe “MinhaFabrica”

```
public final class MinhaFabrica {
   static private int qtdeObjetos=0 ;
   private MinhaFabrica () { } // não pode ser instanciada
   static public MinhaClasse createMinhaClasse() {
     //chama construtor MinhaClasse( );
     qtdeObjetos++;
     return new MinhaClasse (qtdeObjetos);
   }
   static public int getQtdeObjetos ( ) {
     return qtdeObjetos; }
}
```

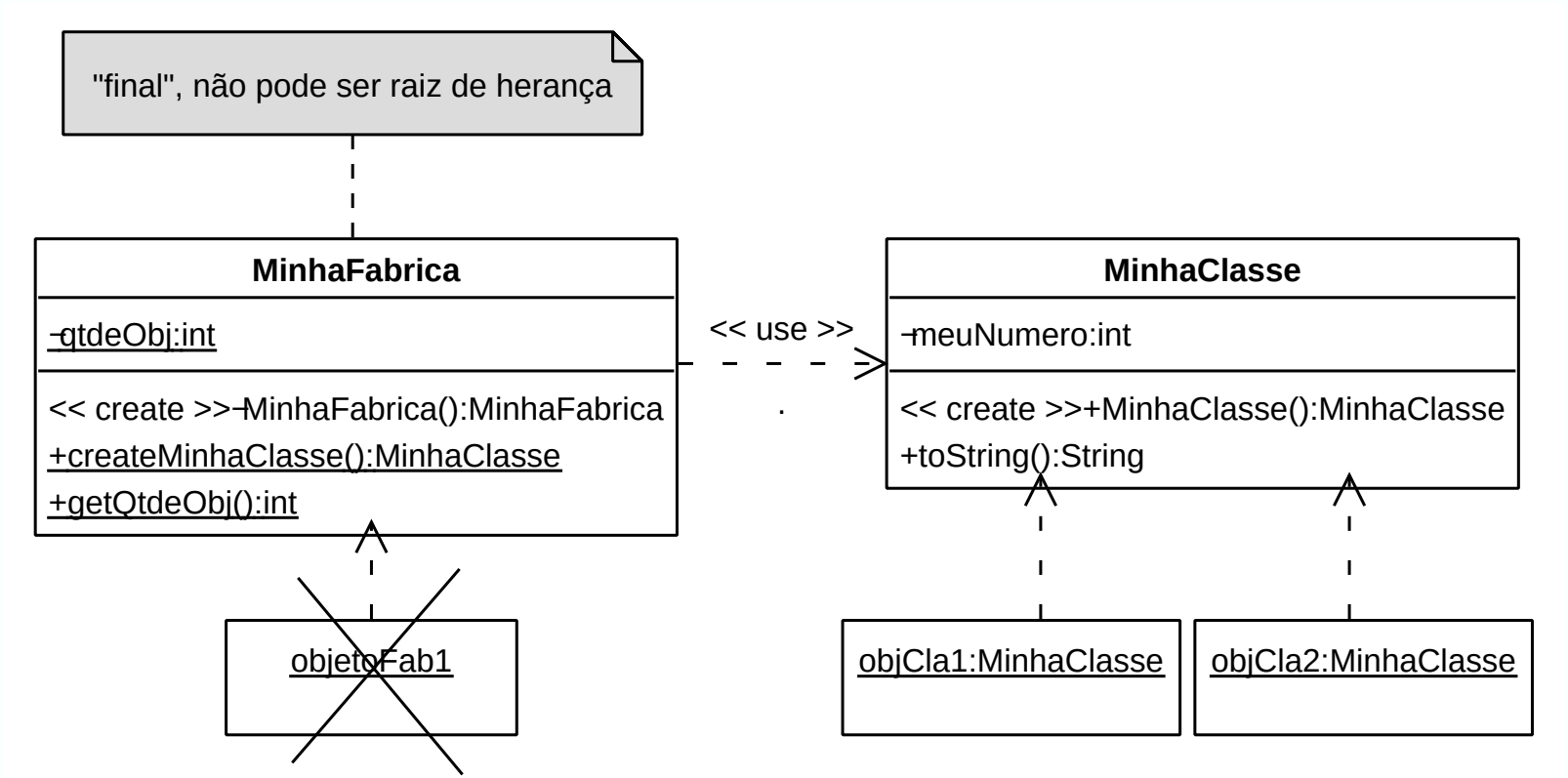
Classe "AlternativaIII"

```
class AlternativaIII {
  public static void main (String Arg [ ] ) {
    System.out.println (‘‘Qtde de objetos =’’
            +MinhaFabrica.getQtdeObjetos( )) ;
    MinhaClasse obj1 = MinhaFabrica.createMinhaClasse ( );
    MinhaClasse obj2 = MinhaFabrica.createMinhaClasse ( );
    MinhaClasse obj3 = MinhaFabrica.createMinhaClasse ( );
    System.out.println(obj1);
    System.out.println(obj2);
    System.out.println(obj3);
    System.out.println (‘‘Qtde de objetos =’’
            +MinhaFabrica.getQtdeObjetos( )) ;
  } }
```

Projeto da Classe "MinhaFabrica" (I)

- **MinhaFabrica:** Papel de uma fábrica de objetos do tipo `MinhaClasse`, tendo como função controlar o número seqüencial de cada objeto criado através do método `createMinhaClasse()`.
- **MinhaFabrica:** Só contém atributos e métodos `static`.
- **MinhaClasse:** Não possui métodos ou atributos `static`, trabalhando unicamente com os objetos da aplicação.
- Para a criação de um objeto do tipo `MinhaClasse`, se faz necessário o envio de uma mensagem extra para a fábrica de objetos; o que é feito através da chamada `MinhaFabrica.createMinhaClasse()`, o que não garante que o construtor da classe `MinhaClasse` não possa ser chamado diretamente.

Projeto da Classe “MinhaFabrica” (II)



- Como corrigir o problema da visibilidade pública do construtor de `MinhaClasse`? MinhaClasse obj4 = new MinhaClasse();

Refactoring: “MinhaClasse”

```
package meupacote;

public class MinhaClasse {
    int meuNumero;

    MinhaClasse(int n) {
        meuNumero = n;
    }

    public String toString(){
        return ("Objeto n??mero"+meuNumero);
    }

}
```

Refactoring: “MinhaFabrica”

```
package meupacote;

public final class MinhaFabrica {
    static private int qtdeObjetos=0 ;
    private MinhaFabrica () { } // n??o pode ser instanciada
    static public MinhaClasse createMinhaClasse() {
        //chama construtor MinhaClasse( );
        qtdeObjetos++;
        return new MinhaClasse (qtdeObjetos);
    }
    static public int getQtdeObjetos ( ) {
        return qtdeObjetos;
    }
}
```

Refactoring: Classe "AlternativaIII"

```
import meupacote.MinhaClasse;
import meupacote.MinhaFabrica;
class AlternativaIII {
public static void main (String Arg[ ] ) {
System.out.println ("Qtde  =" +MinhaFabrica.getQtdeObjetos( )) ;
MinhaClasse obj1 = MinhaFabrica.createMinhaClasse ( );
MinhaClasse obj2 = MinhaFabrica.createMinhaClasse ( );
MinhaClasse obj3 = MinhaFabrica.createMinhaClasse ( );
System.out.println(obj1);
System.out.println(obj2);
System.out.println(obj3);
System.out.println ("Qtde =" +MinhaFabrica.getQtdeObjetos( )) ;
//erro: MinhaClasse obj4 = new MinhaClasse();
    } }
```

Refactoring2: “MinhaClasse”

```
package meupacote;
public interface IMinhaClasse{
public String toString();

}
// ---- arquivo minhaclasse.java ---------
package meupacote;
class MinhaClasse implements IMinhaClasse {
    int meuNumero;

    MinhaClasse(int n) {
        meuNumero = n;
    }

    public String toString(){
```

```
        return ("Objeto n??mero"+meuNumero);
    }

}
```

Refactoring2: “MinhaFabrica”

```
package meupacote;

public final class MinhaFabrica {

    static private int qtdeObjetos=0 ;

    private MinhaFabrica () { } // não pode ser instanciada

    static public IMinhaClasse createIMinhaClasse() {
        //chama construtor MinhaClasse( );
        qtdeObjetos++;
        return new MinhaClasse (qtdeObjetos);
    }

    static public int getQtdeObjetos ( ) {
```

```
        return qtdeObjetos;
    }
}
```

Refactoring2: Classe "AlternativaIII"

```
import meupacote.IMinhaClasse;
import meupacote.MinhaFabrica;
class AlternativaIII {
public static void main (String Arg[ ] ) {
System.out.println ("Qtde de objetos =" +MinhaFabrica.getQtdeObje
IMinhaClasse obj1 = MinhaFabrica.createIMinhaClasse ( );
IMinhaClasse obj2 = MinhaFabrica.createIMinhaClasse ( );
IMinhaClasse obj3 = MinhaFabrica.createIMinhaClasse ( );
System.out.println(obj1);
System.out.println(obj2);
System.out.println(obj3);
System.out.println ("Qtde de objetos ="
                        +MinhaFabrica.getQtdeObjetos( )) ;
    }
}
```

Padrão de Projeto "MinhaFabrica"

- A Alternativa III ilustra uma maneira sistematizada de separar atributos e métodos de classe, de atributos e métodos de objetos.
- Assim, sugere-se a seguinte "disciplina de programação" na utilização do modificador `static`:
  1. Caso seja necessário utilizar o modificador static para atributos de variáveis, isolar os mesmos numa classe que não seja instanciável;
  2. Caso o Item (1) seja custoso, sendo necessária a inclusão de um atributo static numa classe instanciável, qualquer referência a este atributo deve conter, obrigatoriamente, o nome da classe, apesar de isto não ser exigido pelo compilador Java (Alternativa II).

Metaclasses em Ruby

```
class Teste
  @@contador = 0  # variável de classe

  # chamado por Teste.new
  def initialize(nome)
   @@contador += 1 #variável de classe,só existe na classe
   @nome = nome # variável de objeto,só existe em objetos
   n = 1    # variável de método,só existe em initialize
  end

  # self. denota método de classe
  def self.get_contador
    @@contador
  end
```

```
  def get_nome
    @nome
  end

  def get_n
    n
  end
end

puts Teste.get_contador    # método de classe, imprime 0
t = Teste.new('meu nome')  # cria instância de Teste
puts Teste.get_contador    # método de classe, imprime 1
puts t.get_nome   # método de objeto, imprime 'meu nome'
#puts t.get_n     # método de objeto, não funciona pois
          # o 'n' de get_n não é o mesmo 'n' do initialize
#puts t.get_contador # não funciona pois
```

```
                        # o método get_contador é de classe
#puts Teste.get_nome # não funciona pois
                        # o método get_nome é de objeto
```

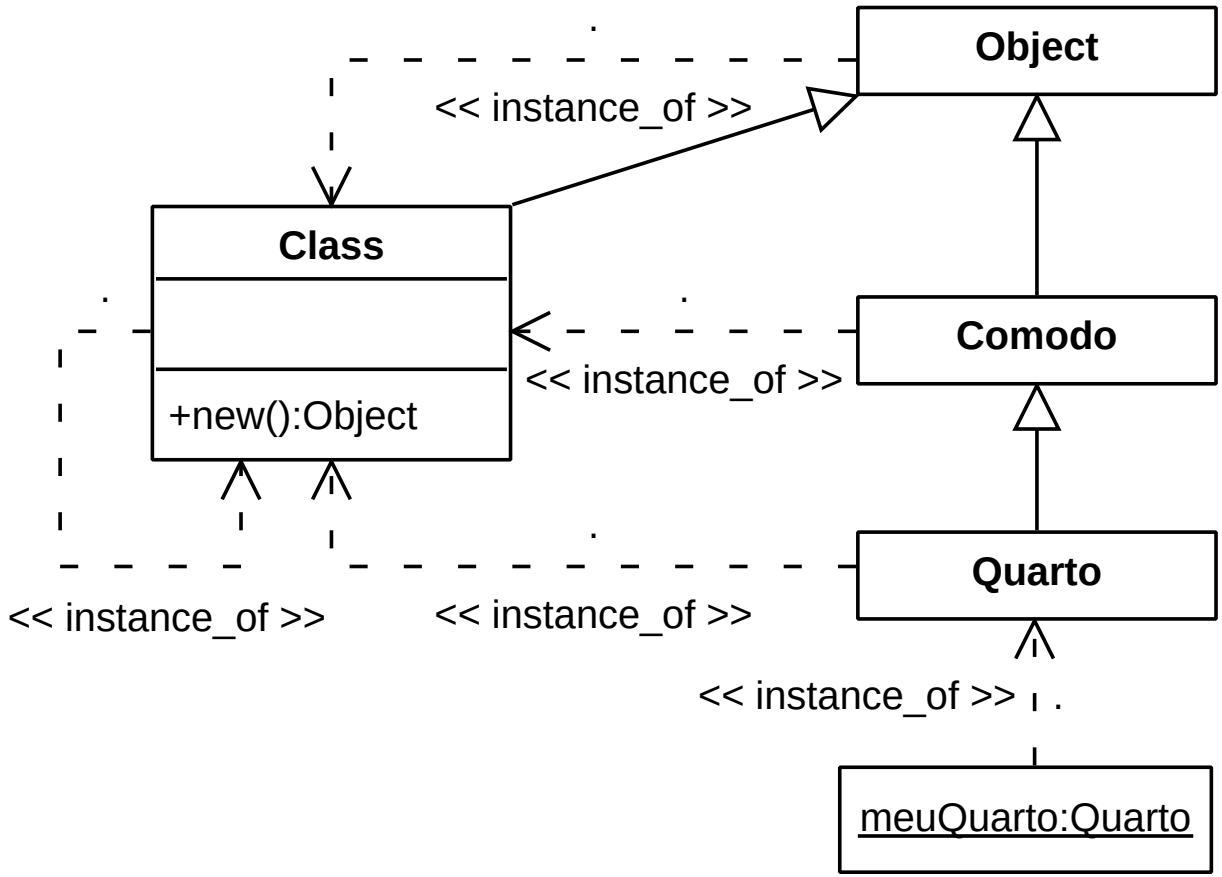
Reflexão Computacional

- **Reflexão Computacional** é definida como sendo a habilidade de observar e manipular o comportamento computacional de um sistema através de um processo chamado de materialização (*reification*).

Reflection de Java

- A biblioteca `java.lang.reflect` contém outras metaclasses que também são referenciadas pela classe `Class`. Através dessas classes pode-se obter informações a respeito da classe de um objeto qualquer, como o nome da classe e o nome dos seus métodos.

Modelo de Smalltalk-76 / Java (I)



Modelo de Smalltalk-76 / Java (II)

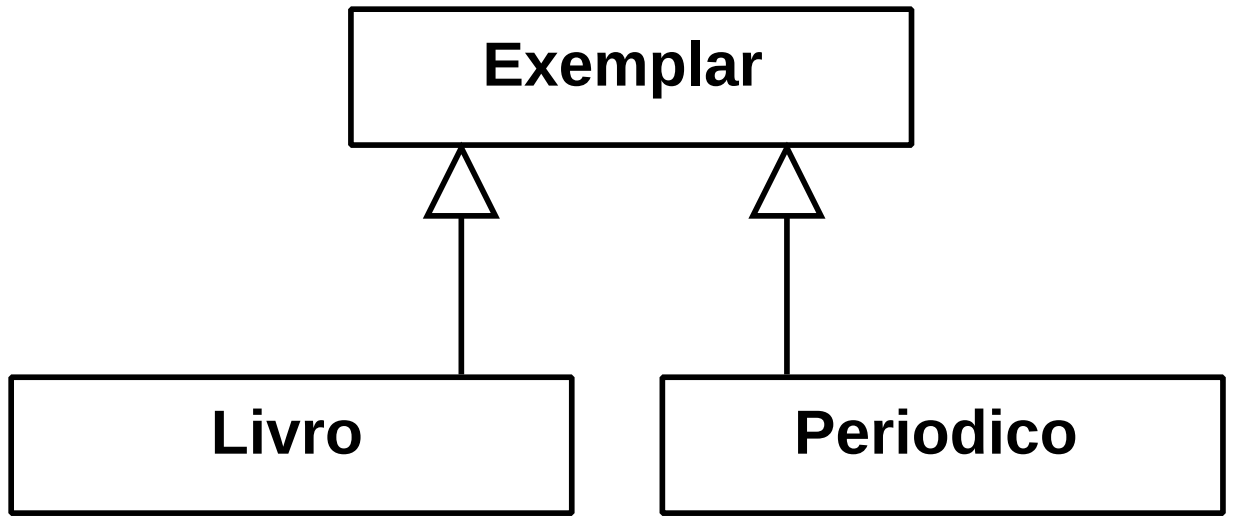
- Os relacionamentos de instanciação ligam cada instância à sua classe geradora e organiza os objetos numa **hierarquia de instanciação**.
- A hierarquia de instanciação é ortogonal à hierarquia de herança.
- A classe `Class` é a única metaclasse, e portanto é a raiz da hierarquia de instanciação.
- A classe `Object` é a raiz da árvore de herança.

Modelo de Smalltalk-76 / Java (III)

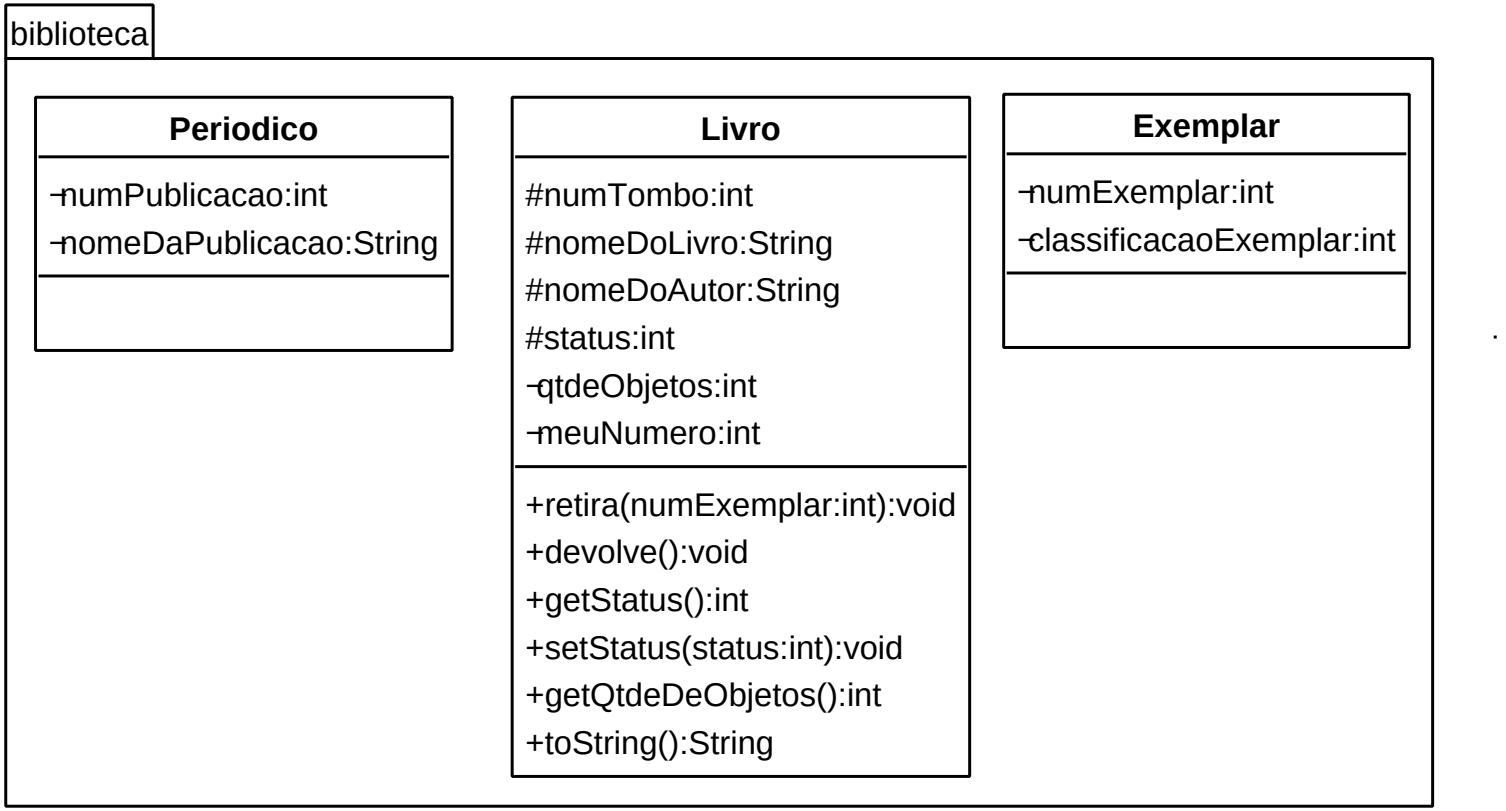
- Como qualquer classe, a classe `Class` herda seu comportamento da classe `Object`.
- A classe `Class` define o comportamento comum a todas as classes.
  Ex.: imprimir o conteúdo de uma classe, instanciar uma classe (`new`).

Exemplo

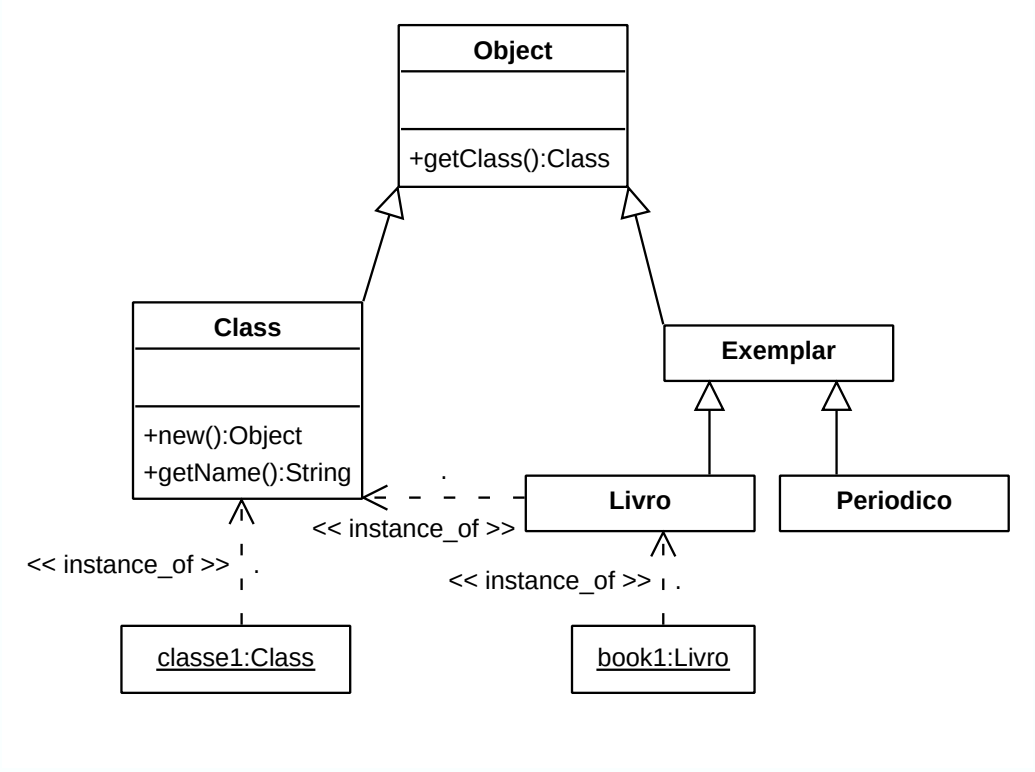
- Uma biblioteca apresenta diversos exemplares que podem ser classificados como livros ou periódicos



Interfaces das Classes da Biblioteca (I)



Interfaces das Classes da Biblioteca (II)



Trabalhando com Metainformação (I)

- getClass( ) / getName( ):

```
//Testando busca pelo nome da classe de determinado objeto
   String nomeDaClasse;
   biblioteca.Livro book1 = new biblioteca.Livro
                  (new Integer(1), ‘‘Menino de Engenho ’’,
                  ‘‘José Lins do Rego’’);
   Class classe1 = book1.getClass();
   nomeDaClasse = classe1.getName();
   System.out.println(‘‘Nome da classe:’’+nomeDaClasse);
```

SAÍDA:

O nome da classe é: biblioteca.Livro

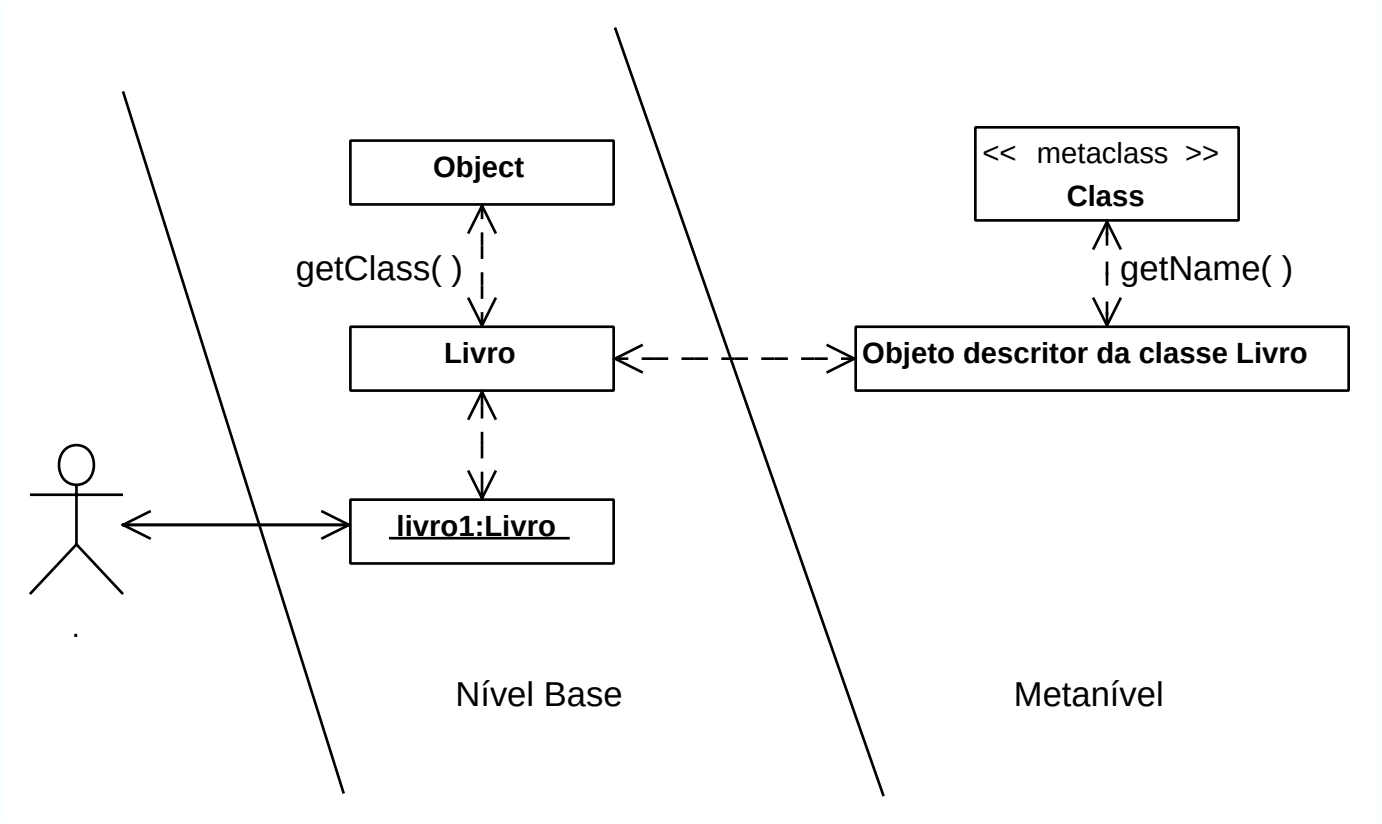
Trabalhando com Metainformação (II)

- **getClass( ):** é um método de Object que retorna o descritor da classe referente ao objeto.

- **getName( ):** é um método de Class que dado o descritor da classe, retorna o nome desta.

Método getClass()

- $\mathbf{getClass(\ )}$ - Devolve a referência para o objeto descritor da Classe.



Método getMethod()

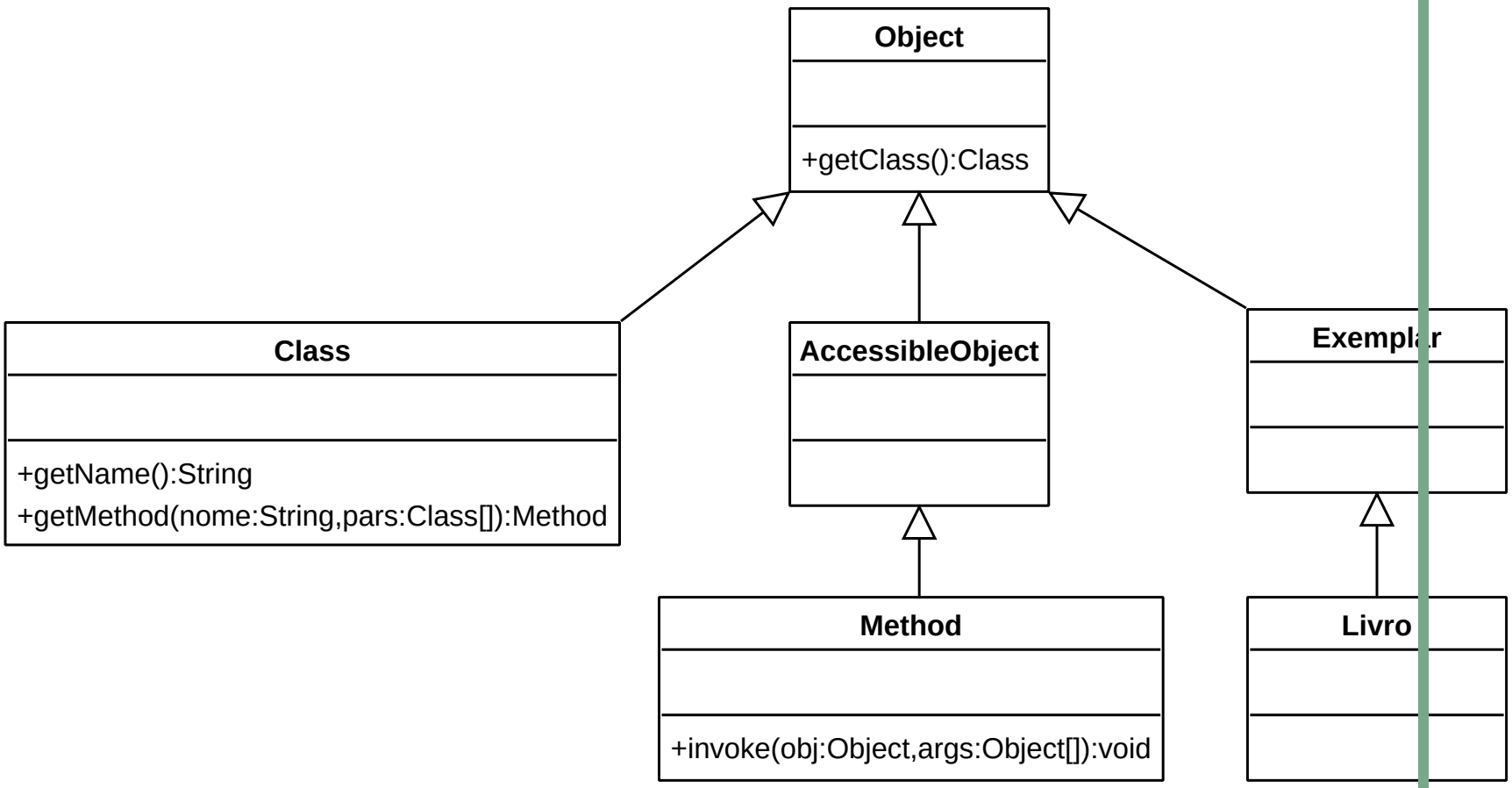
```
//Criação e execução de um método de uma classe
    biblioteca.Livro book2 = new biblioteca.Livro
                (new Integer(1), ‘‘Memorial do Convento’’,
                ‘‘José Saramago’’);
    Class classe2 = book2.getClass(); // classe2 recebe o
          //descritor da classe referente ao objeto book2
    try{
       Integer inteiro = new Integer(1);
       Class classe3 = inteiro.getClass();
           //recebe o descritor da classe referente
           //ao objeto inteiro
       Method metodo = classe2.getMethod(‘‘retira’’,
                                  new Class[] {classe3});
       System.out.println(metodo.toString());
       metodo.invoke(book2, new Object[]{inteiro});
```

```
} catch (Exception e){
    e.printStackTrace();
    //- Assinatura do retira:
    //   retira(int i);
    //- Assinatura do getMethod:
    //   retira(String nome, Class[] tiposParametros);
}
```

SAÍDA:

public void biblioteca.Livro.retira(java.lang.Integer)

Livro 1 retirado na data: Mon Oct 01



Método getSuperclass ()

```
// Busca da superclasse de determinado objeto
  biblioteca.Livro book4 = new biblioteca.Livro
      (new Integer(1), "Lusíadas", "Luis de Camões");
  Class classe = book4.getClass();
  if (classe != null){
   System.out.println("Imprimindo a hierarquia de
                                classes de livro : ");
   for (Class s = classe.getSuperclass(); s != null;
                                s = s.getSuperclass())
     System.out.println(s.getName() + " é  superclasse ");
  }
```

SAÍDA:

Imprimindo a hierarquia de classes de livro :

biblioteca.Exemplar é superclasse

java.lang.Object é superclasse

Manipulação de Metainformação através de "getSuperclass()"

- **getSuperclass( ):** é um método de Class que retorna o descritor da superclasse referente a uma classe.

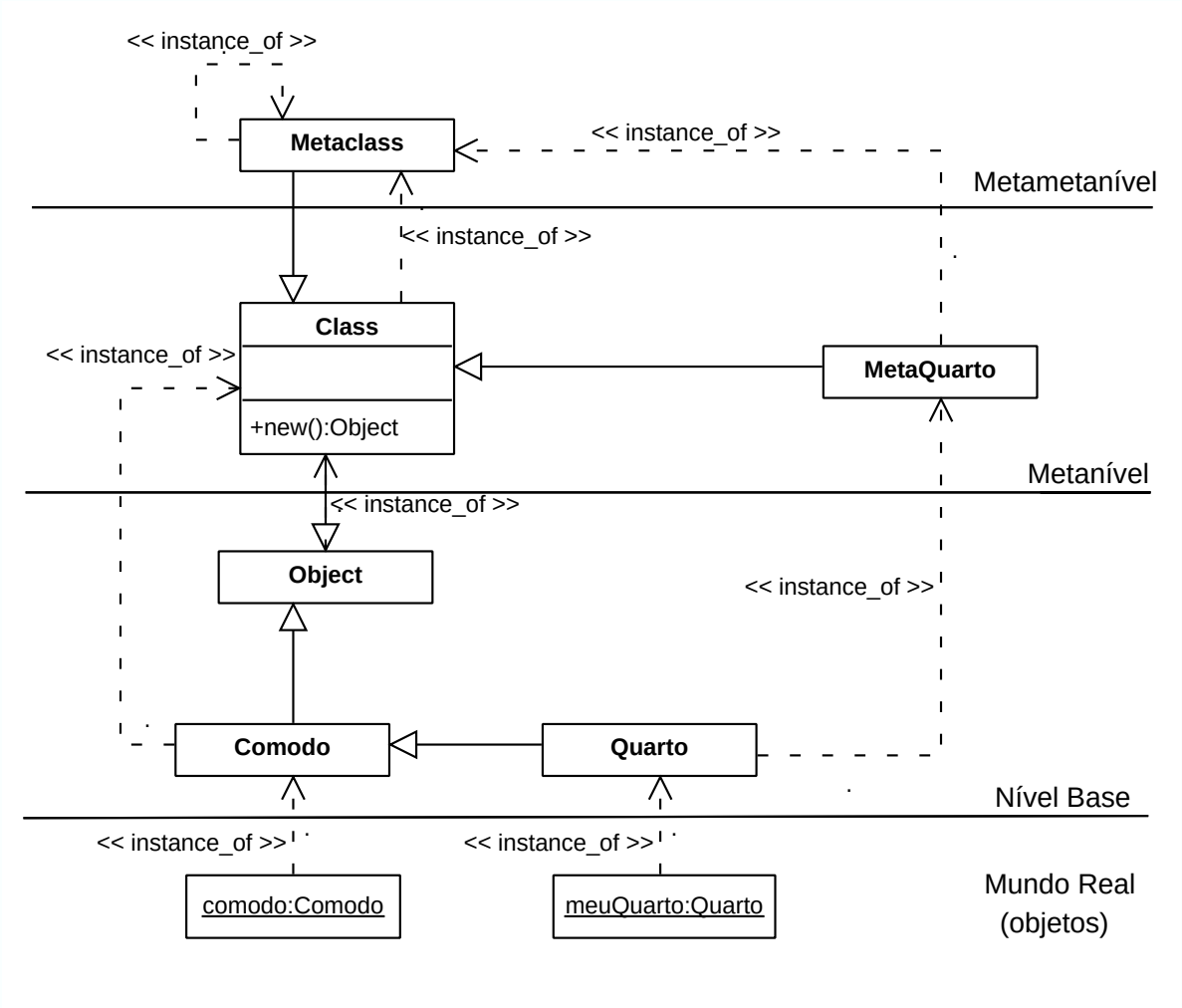
```
(Class s = classe.getSuperclass();  // s obtém o descritor
                                    //da superclasse de classe
System.out.println(s.getName() + " é  superclasse ");
          // utilizando o método getName( ) e tendo ‘‘s’’,
          //que é o descritor da superclasse, é possível
          //obter o nome da superclasse
```

Localização dos Métodos em Java

- ‘‘`getClass()`’’ são definidos na classe `Object`.
- ‘‘`getName()`’’, ‘‘`getMethod()`’’ e ‘‘`getSuperclass()`’’ são definidos na classe `Class`.

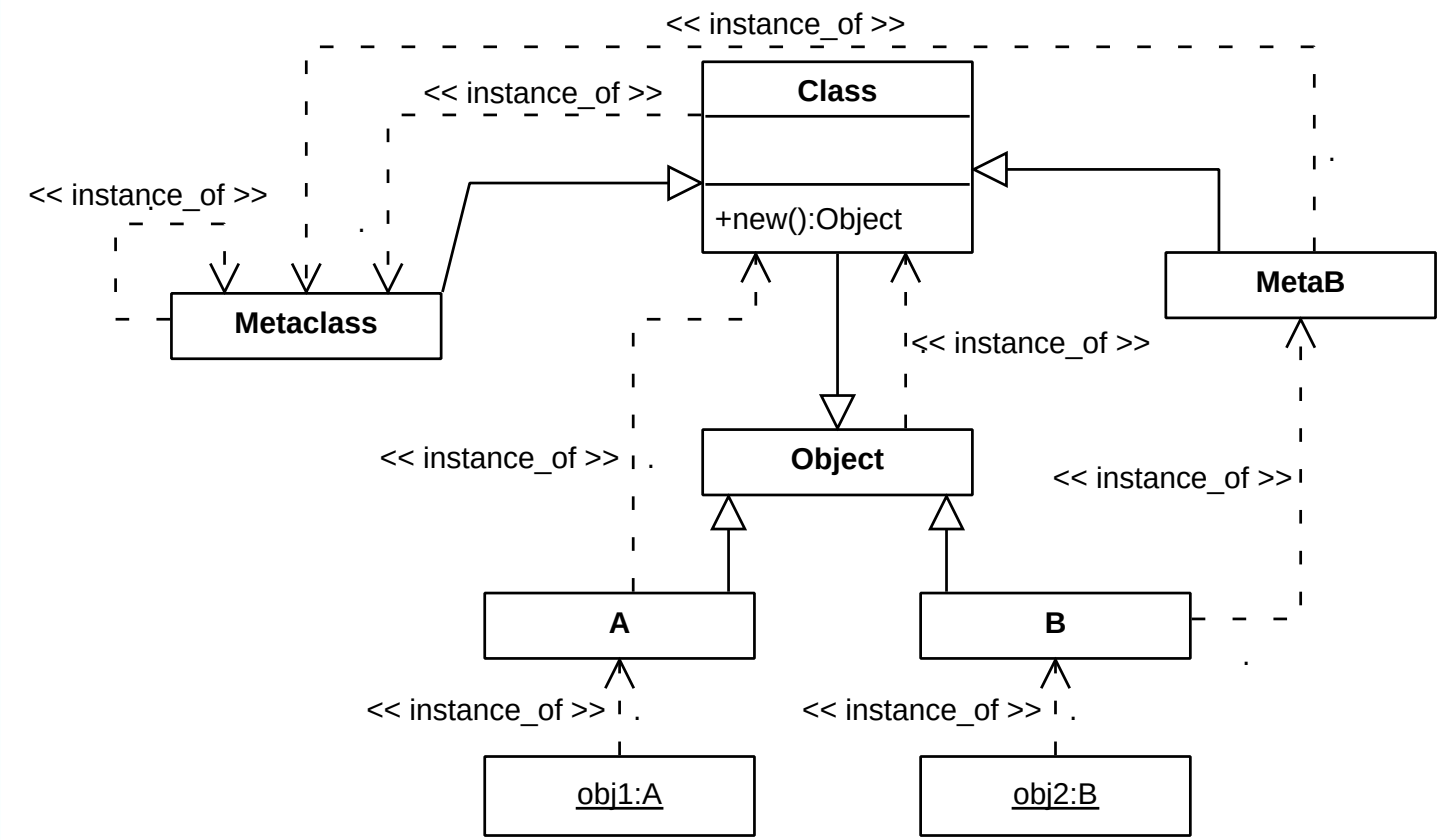
Modelo de Loops (I)

Hierarquia de Instanciação



Modelo de Loops (II)

Hierarquia de Herança



Modelo de Loops (III)

- A classe `Metaclass` define um comportamento "default" para as metaclasses. Ex.: `new` para criar classes.
- Ela é a metaclasse das outras metaclasses, e portanto é sua própria metaclasse.
- A classe `Class` define um comportamento "default" para as classes. Ex.: `new` para criar objetos.
- Ela é a metaclasse "default".
- A classe `Object` define o comportamento dos objetos. Ela é a raiz da árvore de herança. Portanto, a classe `Class` é subclasse de `Object`.