fusberti@ic.unicamp.br



# Associações, Agregações e Composições

prof. Fábio Luiz Usberti

MC322 - Programação Orientada a Objetos

Instituto de Computação - UNICAMP - 2014


# Sumário

# Referência `this`

## Referenciando as entidades de um objeto com a palavra-chave `this`

- Dentro de um método de instância ou um construtor, a palavra-chave `this` consiste em uma referência ao objeto corrente, ou seja, o objeto associado ao método ou construtor chamado.
- Normalmente a referência `this` é utilizada para diferenciar um atributo de instância de uma variável local ou parâmetro com o mesmo nome do atributo.
- Uma boa prática de programação é evitar adotar nomes de parâmetros ou de variáveis locais conflitantes com nomes de atributos. Isso ajuda a prevenir erros sutis, difíceis de localizar.

# Referência `this`

## Referenciando as entidades de um objeto com a palavra-chave `this`

- No escopo de um método não estático, a palavra-chave `this` é usada implicitamente para referenciar atributos e métodos de instância.
- Java conserva em memória uma única cópia de cada método declarado, mesmo que seja um método de instância.
- Desse modo, o uso implícito da referência `this` é necessária para determinar qual objeto está chamando o método.
- Também é possível utilizar a referência `this` em um construtor para chamar outro construtor da mesma classe. Isso é denominado chamada explícita de construtor e será visto no tópico de sobrecarga de construtores.

# Referência this

## Exemplo de uso da referência this

```
// SimpleTimeTest.java
// Demonstração de uso explícito e implícito da referência "this"
public class SimpleTimeTest {
    public static void main(String args[]) {
        SimpleTime time = new SimpleTime(15, 30, 19);
        System.out.println(time.buildString());
    } // fim main
} // fim classe ThisTest

// classe SimpleTime demonstra o uso da referência "this"
class SimpleTime {
    private int hour, minute, second;

    // A referência "this" distingue os parâmetros das variáveis de instância
    public SimpleTime(int hour, int minute, int second) {
        this.hour = hour;
        this.minute = minute;
        this.second = second;
    } // fim construtor SimpleTime

    // uso explícito e implícito da referência "this" na chamada do método toUniversalString
    public String buildString() {
        return String.format("%24s: %s\n%24s: %s", "this.toUniversalString()",
                this.toUniversalString(), "toUniversalString()", toUniversalString());
    } // fim método buildString

    // uso da referência "this" é indiferente neste método
    public String toUniversalString() {
        return String.format("%02d:%02d:%02d", this.hour, this.minute, this.second);
    } // fim método toUniversalString
} // fim classe SimpleTime
```

# Sobrecarga de Construtores

## Múltiplas maneiras de inicializar um objeto

- A sobrecarga de construtores tem o mesmo princípio de sobrecarga de métodos, bastando declarar múltiplos construtores com listas de argumentos distintas.
- Quando pelo menos um construtor é declarado, o compilador não cria o construtor default.
- Se o programador deseja criar seu próprio construtor e manter um construtor default (sem argumentos), torna-se necessário declarar explicitamente um construtor sem argumentos.
- É permitido o uso da referência `this` dentro de um construtor para chamar outro construtor, reaproveitando seu código de inicialização de atributos (chamada explícita de construtor).
- Chamar um construtor dentro de um método com a referência `this` consiste em um erro de compilação.

# Sobrecarga de Construtores

## Exemplo

```
// Time.java
// Classe Time exemplifica o uso de construtores sobrecarregados.
public class Time {
    private int hour; // 0 – 23
    private int minute; // 0 – 59
    private int second; // 0 – 59

    // CONSTRUTORES
    public Time() {
        this(0, 0, 0); // chama construtor com três argumentos
    } // fim construtor sem argumentos

    public Time(int h) {
        this(h, 0, 0); // chama construtor com três argumentos
    } // fim construtor com 1 argumento int

    public Time(int h, int m) {
        this(h, m, 0); // chama construtor com três argumentos
    } // fim construtor com 2 argumentos int

    public Time(int h, int m, int s) {
        setTime(h, m, s); // chama método setTime para validar a entrada
    } // fim construtor com 3 argumentos int

    public Time(Time time) {
        this(time.getHour(), time.getMinute(), time.getSecond()); // chama construtor com três argumentos
    } // fim construtor com 1 argumento Time
    /* continua na próxima página... */
```

# Sobrecarga de Construtores

## Exemplo

```
/* ... continua da página anterior */

// MÉTODOS ACESSORES
public void setHour(int h) {
    hour = ((h >= 0 && h < 24) ? h : 0);
} // fim método setHour

public void setMinute(int m) {
    minute = ((m >= 0 && m < 60) ? m : 0);
} // fim método setMinute

public void setSecond(int s) {
    second = ((s >= 0 && s < 60) ? s : 0);
} // fim método setSecond

public int getHour() {
    return hour;
} // fim método getHour

public int getMinute() {
    return minute;
} // fim método getMinute

public int getSecond() {
    return second;
} // fim método getSecond
/* continua na próxima página... */
```

# Sobrecarga de Construtores

## Exemplo

```
    /* ... continua da página anterior */
    // configura um horário assegurando que os valores de entrada são consistentes
    public void setTime(int h, int m, int s) {
        setHour(h); // configura as horas
        setMinute(m); // configura os minutos
        setSecond(s); // configura os segundos
    } // fim método setTime

    // converte para uma String no formato universal (HH:MM:SS)
    public String toUniversalString() {
        return String.format("%02d:%02d:%02d", getHour(), getMinute(),
                getSecond());
    } // fim método toUniversalString

    // converte para uma String no formato padrão (H:MM:SS AM ou PM)
    public String toString() {
        return String.format("%d:%02d:%02d %s",
                ((getHour() == 0 || getHour() == 12) ? 12 : getHour() % 12),
                getMinute(), getSecond(), (getHour() < 12 ? "AM" : "PM"));
    } // fim método toString
} // fim classe Time
```

# Sobrecarga de Construtores

## Exemplo

```
// TimeTest.java
// Construtores sobrecarregados instanciam objetos do tipo Time.
public class TimeTest {
    public static void main(String args[]) {
        Time t1 = new Time(); // 00:00:00
        Time t2 = new Time(2); // 02:00:00
        Time t3 = new Time(21, 34); // 21:34:00
        Time t4 = new Time(12, 25, 42); // 12:25:42
        Time t5 = new Time(27, 74, 99); // 00:00:00
        Time t6 = new Time(t4); // 12:25:42

        System.out.println("Construtores utilizados :\n");
        System.out.println("t1: todos os argumentos default");
        System.out.printf(" Universal: %s -- Padrão: %s\n\n", t1.toUniversalString(), t1);

        System.out.println("t2: horas especificadas; minutos e segundos default");
        System.out.printf(" Universal: %s -- Padrão: %s\n\n", t2.toUniversalString(), t2);

        System.out.println("t3: horas e minutos especificados; segundos default");
        System.out.printf(" Universal: %s -- Padrão: %s\n\n", t3.toUniversalString(), t3);

        System.out.println("t4: horas, minutos e segundos especificados");
        System.out.printf(" Universal: %s -- Padrão: %s\n\n", t4.toUniversalString(), t4);

        System.out.println("t5: testando uma entrada inválida");
        System.out.printf(" Universal: %s -- Padrão: %s\n\n", t5.toUniversalString(), t5);

        System.out.println("t6: objeto Time passado como argumento");
        System.out.printf(" Universal: %s -- Padrão: %s\n\n", t6.toUniversalString(), t6);
    } // fim main
} // fim classe TimeTest
```

# Sobrecarga de Construtores

## Exemplo: impressão no terminal

```
Construtores utilizados:

t1: todos os argumentos default
 Universal: 00:00:00 -- Padrão: 12:00:00 AM

t2: horas especificadas; minutos e segundos default
 Universal: 02:00:00 -- Padrão: 2:00:00 AM

t3: horas e minutos especificados; segundos default
 Universal: 21:34:00 -- Padrão: 9:34:00 PM

t4: horas, minutos e segundos especificados
 Universal: 12:25:42 -- Padrão: 12:25:42 PM

t5: testando uma entrada inválida
 Universal: 00:00:00 -- Padrão: 12:00:00 AM

t6: objeto Time passado como argumento
 Universal: 12:25:42 -- Padrão: 12:25:42 PM
```

# Sobrecarga de Construtores

## Múltiplas maneiras de inicializar um objeto

- Um construtor pode chamar outros métodos normalmente, mas dado que o construtor está no processo de instanciar o objeto, é possível que variáveis de instâncias ainda não tenham sido inicializadas.
- É um erro comum incluir um tipo de retorno em um método construtor.
- Java permite que métodos tenham o mesmo nome do construtor e tenham um tipo de retorno. No entanto, esses métodos não são construtores e portanto não serão chamados em uma instanciação.

# Verificação de consistência em métodos acessores

## Validação de dados

- A integridade de dados não é obtida automaticamente após declarar atributos com visibilidade `private`.
- É necessário que o código verifique a consistência do estado de um objeto quando novos valores são atribuídos aos seus atributos.
- Um local conveniente para realizar a verificação de consistência é no método acessor `set`.
- O método `set` pode retornar um valor indicando se a atribuição é válida.
- Alternativamente, é possível utilizar tratamentos de exceção para indicar tentativas inválidas de atribuição.

# Associações, Agregações e Composições

## Relação entre objetos

- Associações, composições e agregações correspondem a relações entre objetos. Elas ocorrem quando uma classe faz referência a objetos de outras classes (ou até da mesma classe).
- Em geral, essas relações entre objetos expressam o significado "tem um" (*has a*), ou seja, um objeto possui e tem alguma ligação com outro objeto.
- A diferença entre associações, composições e agregações encontra-se na força da ligação.

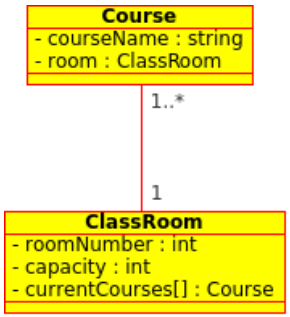
# Associações, Agregações e Composições

## Associação

- Uma associação representa uma ligação entre objetos do tipo "tem um" que não implica em qualquer dependência ou posse de um objeto pelo outro.
- A associação pode ser unidirecional (um objeto faz referência a outro) ou bidirecional (os dois objetos se referenciam).
- A associação em geral reflete algum tipo de uso ou interação temporária entre os objetos.

# Associações, Agregações e Composições

## Diagrama de classes com uma associação




Associação: relação entre classes `Course` e `ClassRoom`.

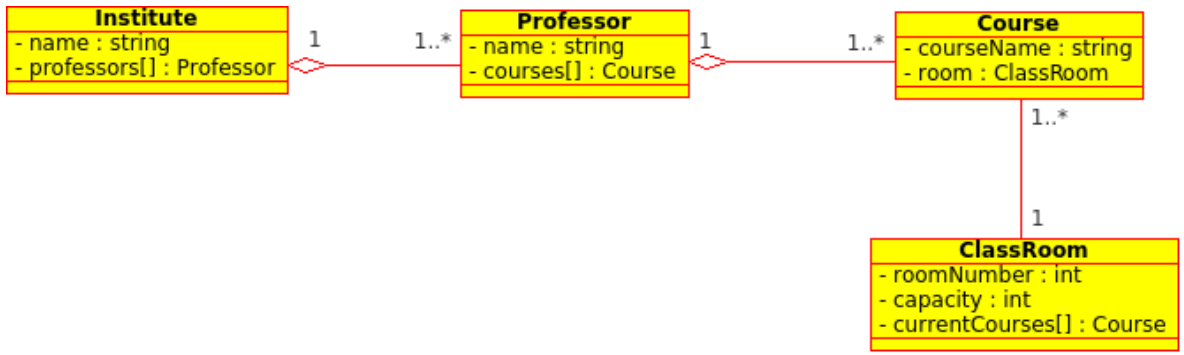
# Associações, Agregações e Composições

## Agregação

- Uma agregação é uma variante da relação "tem um", mais específica do que associação, pois trata de uma relação parte-todo.
- Na agregação, uma classe A é um recipiente de outra classe B, ou seja, B está contido em A.
- Em uma agregação o ciclo de vida de objetos das classes A e B são independentes, ou seja, um objeto A pode existir sem B e vice-versa.

# Associações, Agregações e Composições

## Diagrama de classes com agregação e associação




Agregação: relação entre classes `Institute` e `Professor`.

Agregação: relação entre classes `Professor` e `Course`.

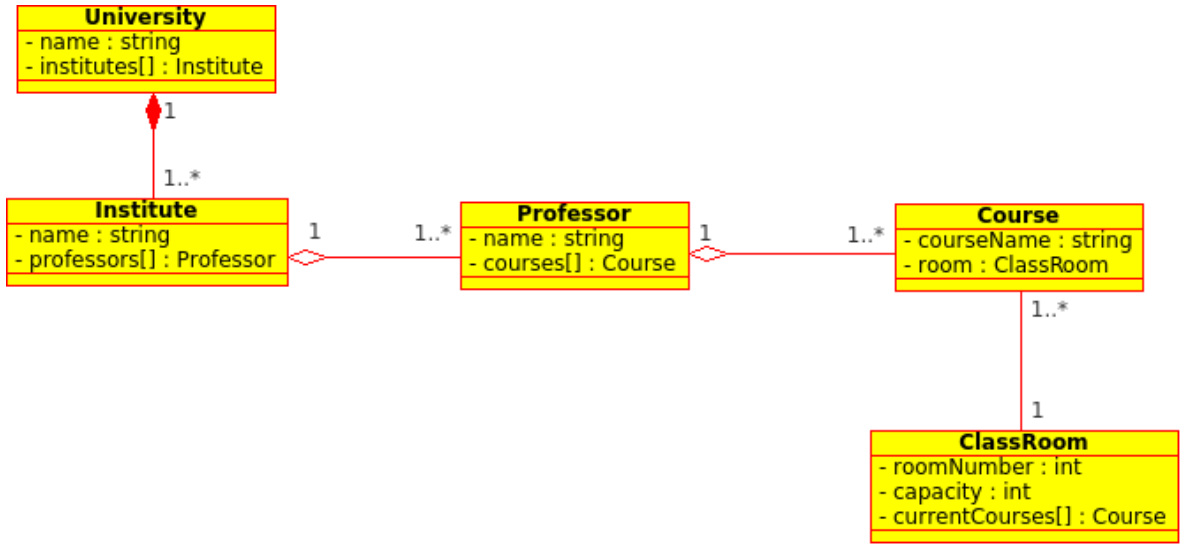
# Associações, Agregações e Composições

## Composição

- Uma composição é a relação do tipo "tem um" (*has a*) mais forte entre dois objetos.
- Como em uma agregação, há uma classe A recipiente de outra classe B. No entanto, a classe B, além de estar contida é uma parte indissociável da classe A.
- A relação presente em uma composição faz com que seus objetos tenham ciclos de vida dependentes. Se o objeto recipiente for destruído (liberados da memória), então todos os objetos contidos também serão destruídos.

# Associações, Agregações e Composições

## Diagrama de classes com composição, agregação e associação




Composição: relação entre classes `University` e `Institute`.

# Associações, Agregações e Composições

## Exemplo

```
// Date.java
// Classe Date representa uma data do calendário

public class Date {
    private int day; // 1—31 depende do mês
    private int month; // 1—12
    private int year; // qualquer

    // construtor: inicializa com um mês, dia e ano.
    // O dia e o mês passam por análise de consistência
    public Date(int theDay, int theMonth, int theYear) {
        month = checkMonth(theMonth); // valida mês
        year = theYear;
        day = checkDay(theDay); // valida dia

        System.out.printf("Construtor para a data %s\n", this);
    } // fim construtor

    // método que avalia a consistência do valor passado para o mês
    private int checkMonth(int testMonth) {
        if (testMonth > 0 && testMonth <= 12) // verifica o mês
            return testMonth;
        else // mês inválido
        {
            System.out.printf("Mês inválido (%d), configurando para 1.", testMonth);
            return 1; // retorna um valor consistente para o mês
        } // fim else
    } // fim método checkMonth
    /* continua na próxima página... */
```

# Associações, Agregações e Composições

## Exemplo

```
    /* ... continua da página anterior */
    // método que avalia a consistência do valor passado para o dia e o mês
    private int checkDay(int testDay) {
        int daysPerMonth[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30,
                31 };

        // verifica se o dia está dentro do intervalo válido para o mês
        if (testDay > 0 && testDay <= daysPerMonth[month - 1])
            return testDay;

        // verifica ano bissexto
        if (month == 2 && testDay == 29
                && (year % 400 == 0 || (year % 4 == 0 && year % 100 != 0)))
            return testDay;

        System.out.printf("Dia inválido (%d), configurando para 1.", testDay);
        return 1; // em caso de valor inválido, retorna o número 1 (válido para
                  // qualquer mês)
    } // fim método checkDay

    // imprime a data no formato dia/mês/ano
    public String toString() {
        return String.format("%d/%d/%d", day, month, year);
    } // fim método toString
} // end class Date
```

# Associações, Agregações e Composições

## Exemplo

```
// Employee.java
// Classe Employee faz referência a objetos da classe Date (agregação)
public class Employee {
    private String firstName;
    private String lastName;
    private Date birthDate;
    private Date hireDate;

    // construtor inicializa objeto com nome e datas de nascimento e contratação
    public Employee(String first, String last, Date dateOfBirth, Date dateOfHire) {
        firstName = first ;
        lastName = last;
        birthDate = dateOfBirth;
        hireDate = dateOfHire;
    } // fim construtor

    // imprime informação do objeto Employee
    public String toString() {
        return String.format("Nome: %s, %s. Contratação: %s Nascimento: %s",
                lastName, firstName, hireDate, birthDate);
    } // fim método toString

    public static void main(String args[]) {
        Date birth = new Date(24, 7, 1949);
        Date hire = new Date(12, 3, 1988);
        Employee employee = new Employee("Fulano", "Silva", birth, hire);
        System.out.println(employee);
    } // fim main
} // fim classe Employee
```

# Composições

## Exemplo: impressão no terminal

```
Construtor para a data 24/7/1949
Construtor para a data 12/3/1988
Nome: Silva, Fulano.  Contratação: 12/3/1988  Nascimento: 24/7/1949
```

# 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.