Classes e Objetos em Java

Prof. Dr. Lucas C. Ribas

Disciplina: Programação Orientada a Objetos

Departamento de Ciências de Computação e Estatística

unesp
UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"

IBILCE / UNESP - CÂMPUS DE SÃO JOSÉ DO RIO PRETO

Agenda

- Estrutura das classes
  - Campos, Métodos e Construtores
- Objetos
- Passagem de parâmetros para métodos
- Modificadores de acesso
- Membros de classe (estáticos)
- Pacotes
- Javadoc
- Classes aninhadas
- Tipo Enum

unesp

Classes

Estrutura das classes

- Declaração minimalista de uma classe

```
class MyClass   {
  // fields
  // constructors
  // methods
}
```

unesp

Estrutura das classes

- Em geral, a declaração de classes pode conter os seguintes componentes, nesta ordem
  - Modificador de acesso (public ou *ausente*)
  - Nome da classe, com a **primeira letra em maiúscula** por convenção
  - O nome da sua classe pai (se houver), precedido pela palavra chave *extends*
    - Uma classe só pode herdar de uma classe pai (ou superclasse)
  - Uma lista de nomes de interfaces que a classe implementa (se houver), separadas por vírgula, precedida da palavra chave *implements*
    - Uma classe pode implementar várias interfaces
  - O corpo da classe, cercado por chaves { }

unesp

Estrutura das classes

- Exemplo

```
class MinhaClasse extends MySuperClass implements MyInterface {
// fields
// constructors
// methods
}
```

unesp

Campos

- Como vimos anteriormente, variáveis de instância são também chamados de campos (ou atributos)
- Cada campo em uma classe é composto de três componentes
  - Modificador de acesso (public, private, protected, *ausente*)
  - Tipo do campo
  - Nome do campo

Campos

- Suponha a classe Bicicleta

```
public class Bicicleta {
  // fields
    public int cadence;
    public int marchas;
    public int velocidade;
    // constructors
    // methods
    }
```

- Os campos definidos são todos do tipo *int*, mas poderiam ser objetos ou arrays.

unesp

Campos

- Pela ideia do encapsulamento, é melhor definir os campos como privados e prover métodos de acesso a eles

```
public class Bycicle {
    // fields
    private int cadence;
    private int gear;
    private int speed;

    // methods
    public int getCadence() {
        return cadence;
    }
    public void setCadence(int newValue) {
        cadence = newValue;
    }
    public int getGear() {
        return gear;
    }
    public void setGear(int newValue) {
        gear = newValue;
    }
    public int getSpeed() {
        return speed;
    }
    public void applyBrake(int decrement) {
        speed -= decrement;
    }
    public void speedUp(int increment) {
        speed += increment;
    }
}
```

Métodos

- De forma geral, a declaração de métodos dentro de uma classe possui seis componentes, nesta ordem:
  - Modificador de acesso (public, private, ...)
  - Tipo de retorno (qualquer tipo) ou *void*
  - Nome do método
    - Assim como para campos e nomes das classes, também há convenções para nomes dos métodos (adiante)
  - Lista de parâmetros entre parênteses, separados por vírgula. Cada parâmetro deve ser precedido pelo seu tipo. Se não há parâmetros, deve haver parênteses vazio.
  - Lista de exceções que o método pode lançar
  - Corpo do método, entre chaves {}

unesp

Métodos

- Apesar de podermos dar qualquer nome para os métodos, é sensato seguir algumas convenções
  - A convenção de maiúsculas e minúsculas segue a mesma convenção para nome de variáveis
    - Primeira palavra minúscula e demais com primeira letra maiúscula
  - A primeira palavra do nome do método deve ser um verbo. As demais palavras podem ser adjetivos, substantivos, etc.
  - Exemplos
    - getValue()
    - compareTo(Object obj)
    - isEmpty()

unesp

Métodos

- A assinatura de um método é determinada pelo nome do método e sua lista de parâmetros
  - Tipo de retorno e nome dos parâmetros não importa
  - Importante, pois Java **suporta sobrecarga de métodos**
- Exemplos de assinaturas diferentes de um mesmo método
  - sort(int [ ]v)
  - sort(double [ ]v)
  - sort(double [ ]v, char []v)

Construtores

- Construtores são utilizados para inicializar os objetos da classe
- São declarados de forma similar aos métodos
  - Contudo, devem ter o nome da classe
  - **Não possuem tipo de retorno (nem void)**
- Por exemplo, a classe Bicycle poderia ter o seguinte construtor

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```

unesp

Construtores

- Para criar um novo objeto (instância) da classe Bicycle, devemos usar o operador new

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```

```
Bycicle myBike = new Bicycle(30, 0, 8);
```

- O comando *new Bicycle(30, 0, 8)* aloca a posição de memória necessária ao objeto e inicializa os campos

unesp

Construtores

- A classe Bicycle poderia ter mais de um construtor, desde que as assinaturas sejam diferentes

```
public Bicycle() {
    gear = 1;
    cadence = 10;
    speed = 0;
}
```

```
Bycicle myBike = new Bicycle();
```

- Neste caso, o comando *new Bicycle( )* utiliza o construtor sem argumentos da classe

unesp

Construtores

```
public class Bycicle {
    // fields
    private int cadence;
    private int gear;
    private int speed;
    // constructors
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }
    public Bicycle() {
        gear = 1;
        cadence = 10;
        speed = 0;
    }
    // methods (get, set, ...)
}
```

Construtores

- É possível definir uma classe sem construtores, mas é preciso ficar atento
  - Neste caso, o compilador irá prover um construtor padrão para a classe, sem argumentos
  - Esse construtor padrão irá chamar o construtor **sem argumentos** da sua superclasse
    - Se a superclasse não tiver um construtor sem argumentos, o compilador acusará o problema
    - E se a classe não tiver uma superclasse?
    - Lembre-se que TODAS as classes herdam de Object
    - Object TEM um construtor sem argumentos

unesp

Parâmetros de Métodos e Construtores

- Os parâmetros dos métodos e construtores de uma classe podem ser de qualquer tipo
  - Primitivos (int, double, float, etc.)
  - Referências (objetos e arrays)
- Exemplo

```
public Polygon createPolygon(Point[] corners) {
    // method body goes here
 }
```

unesp

Parâmetros de Métodos e Construtores

- Quando não se sabe o número exato de parâmetros de um mesmo tipo que o método deve ter, podemos utilizar a notação de *varargs*
  - Permite que o método invocador passe os argumentos separados por virgula e não dentro de um array
- Notação
  - tipo... variável

```
public PrintStream printf(String format, Object... args)
```

```
System.out.printf("%s: %d, %s", name, idnum, address);
System.out.printf("%s: %d, %s, %s, %s", name,idnum,address,phone,email);
```

unesp

Objetos

Objetos

- Um programa típico em Java cria diversos objetos, de vários tipos
- A interação entre os objetos se dá pela chamada dos métodos
  - Chamada de métodos caracteriza a troca de mensagens entre os objetos
- As interações entre os objetos é responsável pela execução das tarefas do programa

unesp

Objetos

- A criação de um objeto envolve três passos
  - **Declaração**: associação de um nome de variável a um tipo de objeto
  - **Instanciação**: a palavra-chave new cria uma nova instância (objeto)
  - **Inicialização**: a operação new é seguida de uma chamada a um dos construtores da classe, que inicializa o objeto
- Exemplos

```
Point originOne = new Point(23, 94);
Retangulo rectOne = new Retangulo(originOne, 100, 200);
Retangulo rectTwo = new Retangulo(50, 100);
```

unesp

Objetos

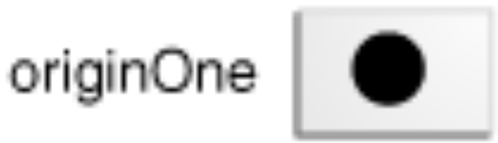
- No exemplo anterior, as três etapas foram feitas de uma só vez
- Porém, é possível declarar um objeto sem instanciá-lo
- A declaração de variáveis primitivas já alocam a quantidade de memória necessária para aquele tipo
- Isso não acontece para variáveis do tipo objeto ou arrays
  - Variáveis do tipo referência

```
int x;
Point originOne;
```

Objetos

- A simples declaração de um objeto não cria o objeto
- Ao tentar usar um objeto não criado, ocorre um erro de compilação
- Variáveis que não foram inicializadas são como ponteiros (implícitos) que não referenciam nenhum objeto

```
Point originOne;
```



Objetos

- Considere a classe Point

```java
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

- A chamada abaixo cria um objeto Point

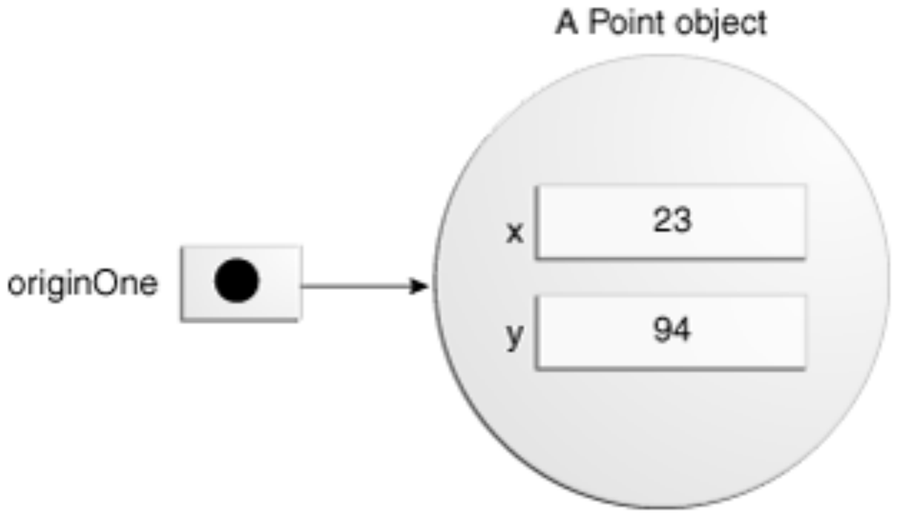
```java
Point originOne = new Point(23, 94);
```

Objetos

- Ao criar uma instância da classe, através do operador new, a memória é alocada e a referência do objeto criado é retornada para a variável

- Operador também chama o construtor da classe

```
Point originOne = new Point(23, 94);
```



Objetos

- Não é necessário associar a referência do objeto alocado a uma variável
  - Podemos utilizar o retorno do operador new diretamente, da maneira como for conveniente
  - Note, porém, que o programa não tem a referência para o objeto criado

```
int height = new Retangulo().height;
```

  - Pelo código acima, o que é *height* na classe Rectangle e qual o tipo de acesso?

Objetos

- Considere a classe Retangulo

```
public class Retangulo {
    public int width = 0;
    public int height = 0;
    public Point origin;
    // 4 construtores
    public Retangulo() {
        origin = new Point(0, 0);
    }
    public Retangulo(Point p) {
        origin = p;
    }
    public Retangulo(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
    public Retangulo(Point p, int w,
    int h) {
        origin = p;
        width = w;
        height = h;
    }
    // método para mover retangulo
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }
    // método para calcular
        area do triangulo
    public int getArea() {
        return width * height;
    }
}
```

Objetos

- O retângulo é representado por um ponto de origem, um valor de altura e um valor de largura
- Cada construtor é capaz de criar um retângulo com informações diferentes
- O compilador Java sabe qual construtor deve ser chamado pela lista de parâmetros que é passada

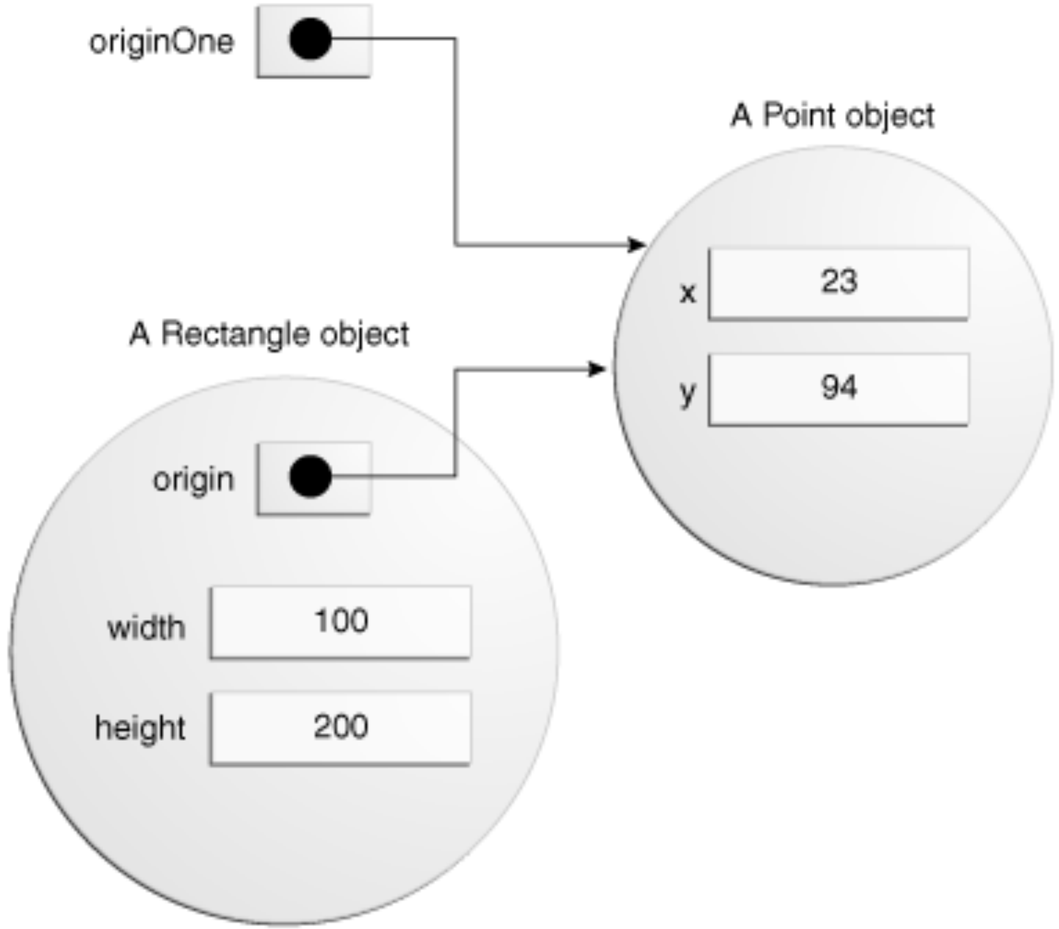
unesp

Objetos

- Considere as situações abaixo
  - O que acontece na memória?
  - Quantas referências existem para o primeiro objeto criado (do tipo Point)?

```
Point originOne = new Point(23, 94);
Retângulo rectOne = new Rectangle(originOne, 100, 200);
```

Objetos




```
Point originOne = new Point(23, 94);
Retangulo rectOne = new Rectangle(originOne, 100, 200);
Retangulo rectTwo = new Rectangle(50, 100);
```

Objetos

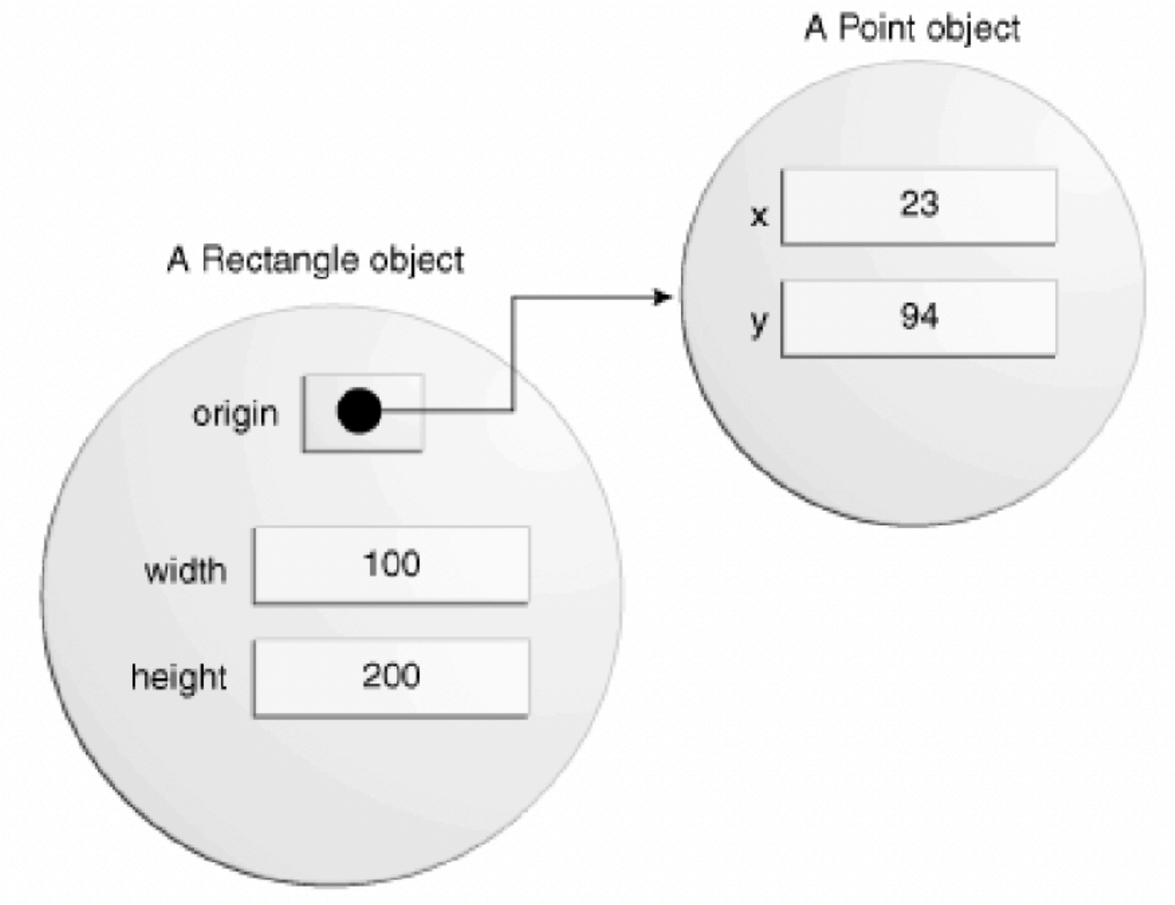
- Alteramos o código para o que está abaixo
- O retângulo representa a mesma informação do anterior?
- Há alguma diferença na memória?

```
Retangulo rectOne = new Rectangle(100, 200);
rectOne.move(23, 94);
```

- Não temos mais a referência externa para o objeto do tipo Point

unesp

Objetos


A Point object
x
23
y
94
A Rectangle object
origin
width
100
height
200

Destruição de Objetos

- Como vimos, o GC do Java libera a memória de um objeto quando não há mais referências para este objeto
- Em geral, isso ocorre quando uma referência fica fora de escopo
- Porém, podemos fazer o desreferenciamento explicitamente utilizando **null**
  - É preciso atentar se TODAS as referências para o objeto foram removidas

```
rectOne = null;
```

unesp

Passagem de Parâmetros para Métodos

- Passagem de tipos primitivos para métodos e construtores são sempre por valor
  - Cópia dos valores é colocada nos parâmetros
- Passagem dos tipos de referência também é feita por valor
  - Isso porque as variáveis deste tipo possuem como valor a referência para um objeto/array
  - Porém, com a referência ao objeto/array é possível alterar os campos do objeto, caso se tenha acesso suficiente
  - Mas ao retornar do método, a referência nunca é perdida

unesp

Passagem de Parâmetros

unesp

Passagem de Parâmetros para Métodos

- Exemplo
  - Qual o efeito do código?
  - O que acontece na memória?

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

```
Circle myCircle = new Circle(10,20);
moveCircle(myCircle, 23, 56);
```

Palavra-chave *this*

- Dentro de métodos de instâncias e construtores, o *this* representa o objeto (instância) atual
  - Não faz sentido para métodos de classe (static)
- Uma aplicação muito comum do *this* é para diferenciar os campos de um objeto dos parâmetros de um método ou construtor
  - Como ele, é possível acessar qualquer membro da instância atual, de modo a não gerar ambiguidade

Palavra-chave *this*

- Lembre-se do construtor da classe Point

```
public class Point {
    public int x = 0;
    public int y = 0; //constructor
    public Point(int a, int b) {
        x = a;
        y = b;
    }
}
```

Palavra-chave *this*

- O que aconteceria se o nome dos parâmetros do construtores fossem os mesmos nomes dos campos da classe?

```
public class Point {
    public int x = 0;
    public int y = 0; //constructor
    public Point(int x, int y) {
        x = x;
        y = y;
    }
}
```

unesp

Palavra-chave *this*

- O que aconteceria se o nome dos parâmetros do construtores fossem os mesmos nomes dos campos da classe?
  - Com o this, é possível acessar o campo do objeto, mesmo quando uma variável local ou parâmetro obscurece o campo

```
public class Point {
    public int x = 0;
    public int y = 0; //constructor
    public Point(int x, int y) {
          this.x = x;
           this.y = y;
    }
 }
```

unesp

Palavra-chave *this*

- Outra aplicação do *this* é para chamar um construtor dentro de outro construtor da mesma classe
- IMPORTANTE: a chamada de outro construtor com o *this* deve ser a primeira coisa dentro de um construtor
  - Primeira linha

Palavra-chave *this*

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(0, 0, 1, 1);
    }
    public Rectangle(int width, int height) {
        this(0, 0, width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
    ...
```

Relembrando

```
private Point p1, p2;
private Rectangle r1;
Private Circle c1;

r1 = new Rectangle(p1, 10, 10);
c1 = new Circle(p2, 5);

p1 = new Point(1,1);
p2 = new Point(2,2);

c1.setPoint(p2);
p2 = p1;
r1.setPoint(p2);

p1 = null;
p2 = null;
```

Modificadores de Acesso

Modificadores de Acesso

- Existem quatro tipos de modificadores de acesso
  - *public*: todas as classes tem acesso
  - *private*: apenas a classe atual tem acesso
  - *protected*: classes do seu pacote e subclasses (que podem estar fora do seu pacote) tem acesso
- *ausente* (package-private): classes do seu pacote tem acesso
- Classes podem ser declaradas apenas como public ou package-private
- Membros da classe (campos e métodos) podem ter qualquer um dos quatro tipos de acesso

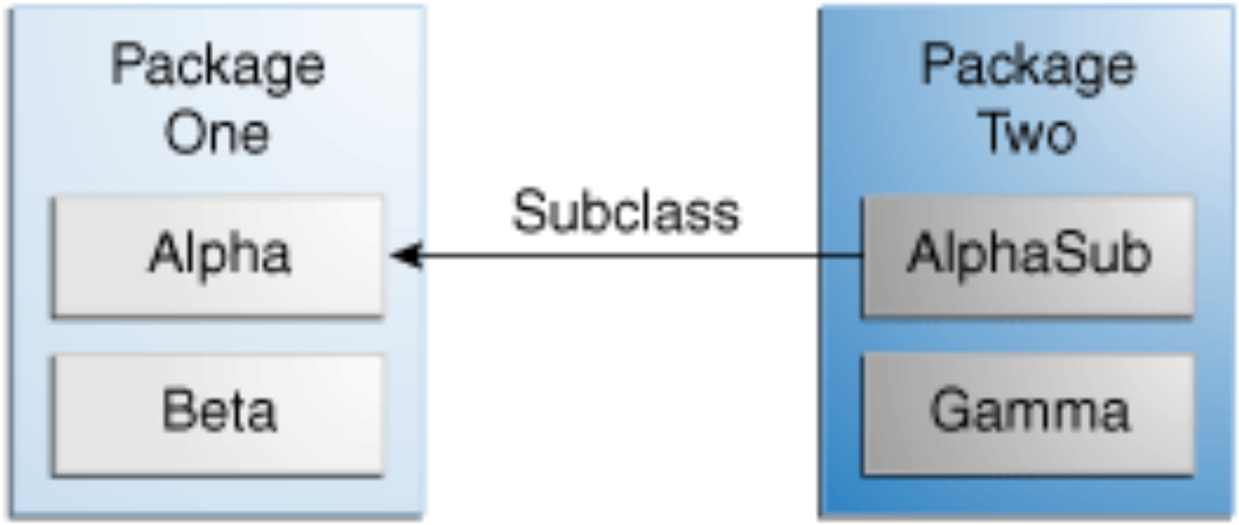
unesp

Modificadores de Acesso

- Os tipos de acesso são importantes em duas situações para o programador
  - Saber quais membros das classes externas ao seu projeto (API Java, por exemplo) suas classes poderão acessar
  - Quando escrevemos uma classe, precisamos definir os níveis de acesso para as outras classes

Modificadores de Acesso

- Exemplo



Visibilidade de membros da classe Alpha				
Modificador	**Alpha**	**Beta**	**AlphaSub**	**Gamma**
public	S	S	S	S
protected	S	S	S	N
no modifier	S	S	N	N
private	S	N	N	N

Modificadores de Acesso

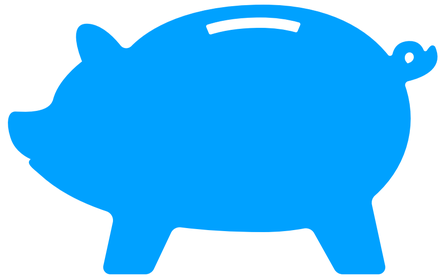
- Dicas para a escolha do nível de acesso
  - Use o nível mais restrito possível (private), a menos que você tenha uma boa razão para não fazê-lo
  - Evite campos públicos, exceto para constantes
    - Campos públicos limitam a flexibilidade do código, deixando a implementação mais presa a um contexto
    - Quando não permitimos o acesso direto às variáveis, podemos alterar mais facilmente alguma funcionalidade
- Uma boa escolha dos níveis de acesso evita erros de **mau uso**

unesp

Você é o novo contratado do banco UNESP

- Crie uma classe **Conta** que tenha os seguintes campos privados:
  - Nome
  - CPF (pode usar long)
  - Saldo R$
  - Número da conta (poderíamos para gerar automaticamente?)
  - Agência
- Para cada um dos campos disponibilize um método ***set*** e ***get***
- Faça um método para depósito (que recebe um valor) e outro método para saque da conta (que recebe um valor e retorna ele)

- Depois na *main:*
  - Crie várias contas (objetos) e realize as operações para teste.



unesp

Bibliografia

- DEITEL, H. M. & DEITEL, P.J. "Java : como programar", Bookman, 2017.
- Material baseado nos slides:
  - Luiz E. Virgilio da Silva. Notas de Aula de Programação Orientada a Objetos (ICMC/USP).