SCC0504 – Programação Orientada a Objetos

Classes e Objetos

Luiz Eduardo Virgilio da Silva
ICMC, USP

Sumário

- 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

Estrutura das classes

- Declaração minimalista de uma classe

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

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 {}

Estrutura das classes

- Exemplo

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

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 Bicycle

```
public class Bicycle {
    // fields
    public int cadence;
    public int gear;
    public int speed;

    // constructors
    // methods
}
```

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

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 {}

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()

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
    - draw(int)
    - draw(double)
    - draw(String, int)

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;
}
```

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

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

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

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
}
```

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... variavel

```
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);
```

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

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);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
Rectangle rectTwo = new Rectangle(50, 100);
```

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

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

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

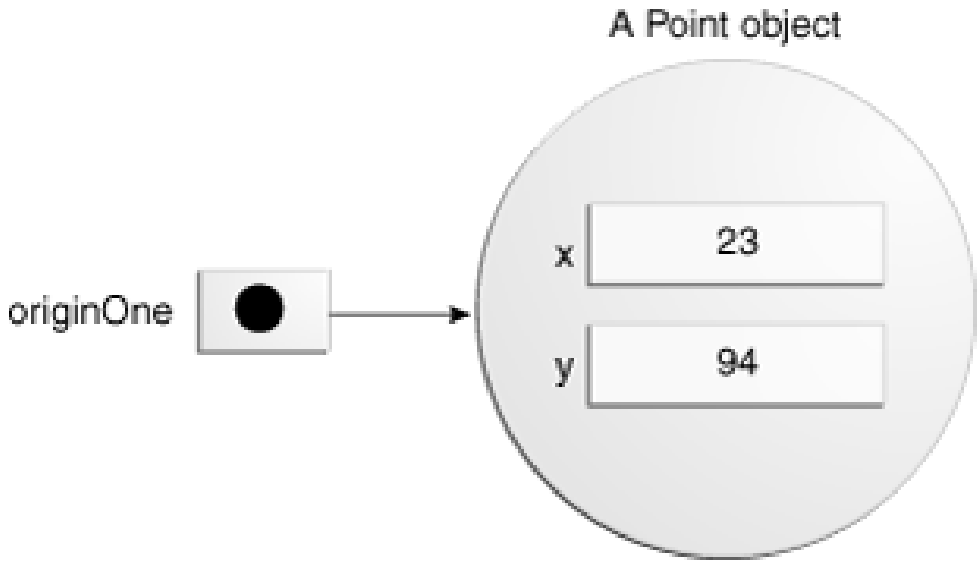
- A chamada abaixo cria um objeto Point

```
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 Rectangle().height;
```

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

Objetos

- Considere a classe Rectangle

```
public class Rectangle {
    public int width = 0;
    public int height = 0;
    public Point origin;

    // four constructors
    public Rectangle() {
        origin = new Point(0, 0);
    }
    public Rectangle(Point p) {
        origin = p;
    }
    public Rectangle(int w, int h) {
        origin = new Point(0, 0);
        width = w;
        height = h;
    }
```

Objetos

- Considere a classe Rectangle

```
    public Rectangle(Point p, int w, int h) {
        origin = p;
        width = w;
        height = h;
    }
    // a method for moving the rectangle
    public void move(int x, int y) {
        origin.x = x;
        origin.y = y;
    }
    // a method for computing the area of the rectangle
    public int getArea() {
        return width * height;
    }
}
```

Objetos

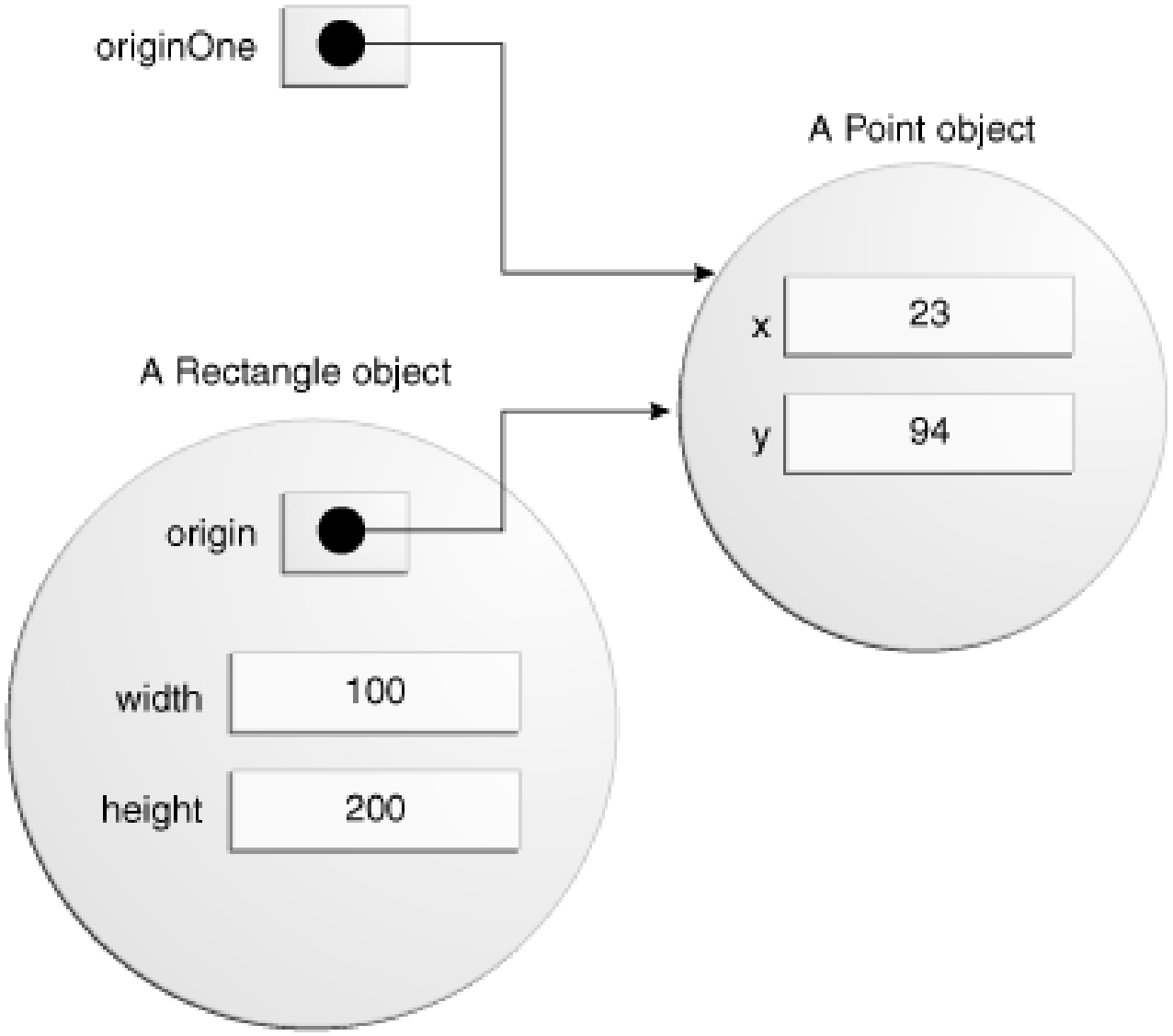
- O retângulo é representado por um ponto de origem, um valor de altura e um valor de largura
- Cada construtor é capaz de criar um retângulo com informações diferentes
- O compilador Java sabe qual construtor deve ser chamado pela lista de parâmetros que é passada

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);
Rectangle rectOne = new Rectangle(originOne, 100, 200);
```

Objetos


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

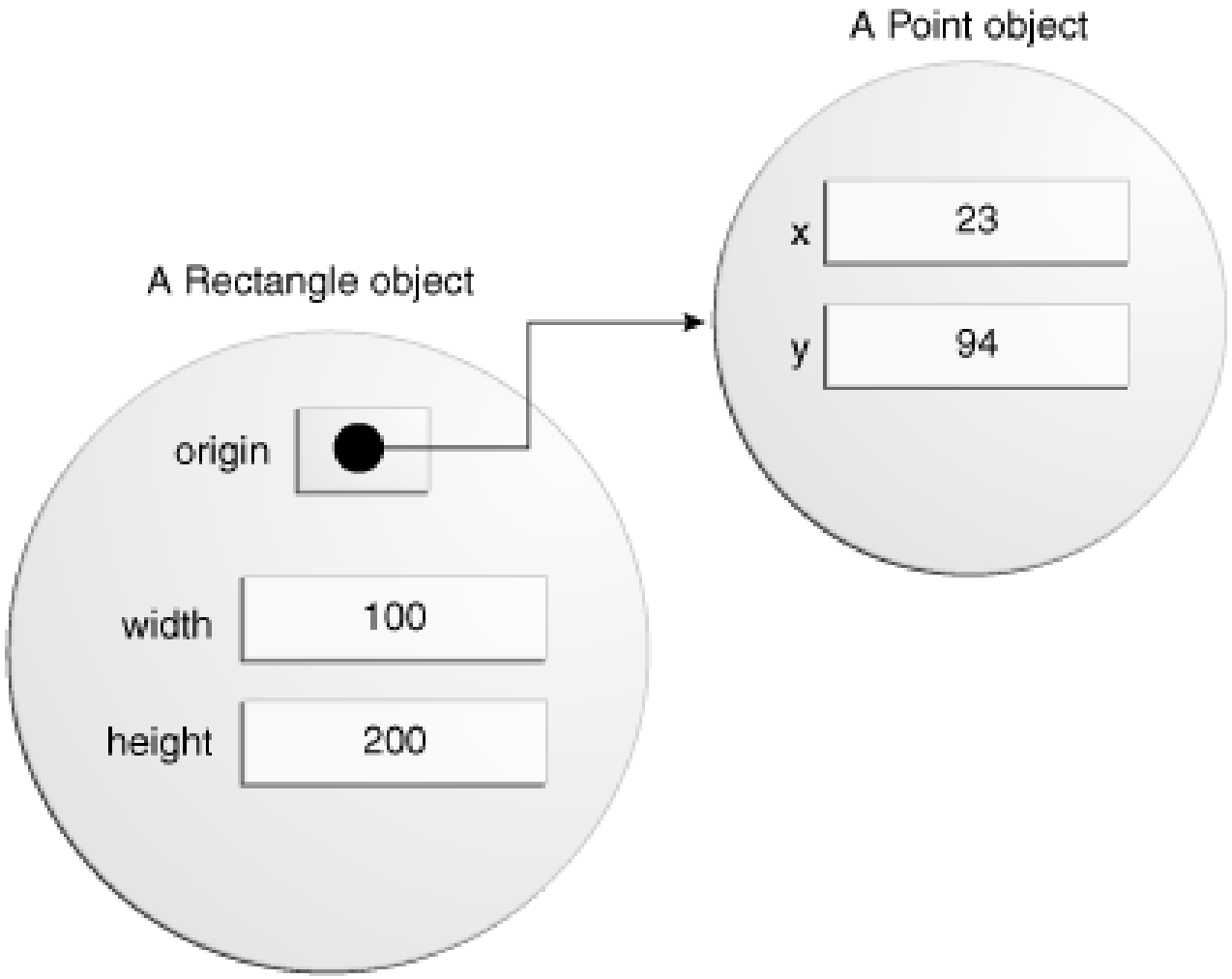
Objetos

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

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

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

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**

```
rectOne = null;
```

  - É preciso atentar se TODAS as referências para o objeto foram removidas

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

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;
    }
}
```

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;
    }
}
```

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

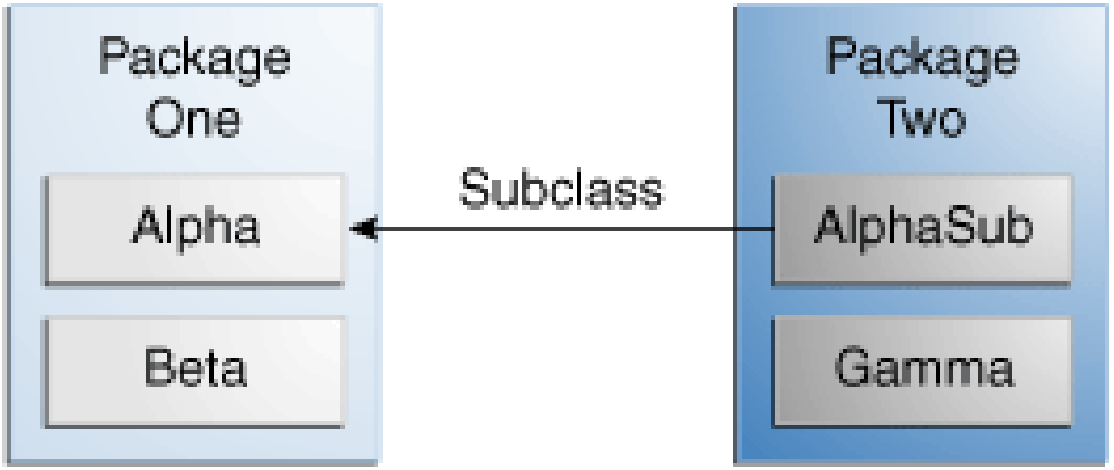
- Existem quatro tipos de modificadores de acesso
  - public: todas as classes tem acesso
  - private: apenas a classe atual tem acesso
  - protected: classes do seu pacote e subclasses (que podem estar fora do seu pacote) tem acesso
  - *ausente* (package-private): classes do seu pacote tem acesso
- 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

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**

Membros de Classe

- Quando um objeto é criado (instanciado), cada um terá seu conjunto de variáveis de instância
  - Essas variáveis serão alocadas para cada objeto diferente
- As variáveis de classe são comuns a todos os objetos
  - Uma única variável na memória, cuja referência é compartilhada por todas as instâncias
- Variáveis de classe são declaradas utilizando a palavra chave *static*
  - Campos estáticos

Membros de Classe

- Variáveis de classe podem ser manipuladas por qualquer instância da classe, mas não há a necessidade de ter uma instância
  - Ela existe independente de qualquer instância

myObject.staticField;
MyClass.staticField; ← preferível

- Exemplo de aplicação
  - Suponha que queiramos associar um ID para cada objeto Bycicle que for criado (em qualquer lugar)
  - Queremos fazer isso de forma serial, ou seja: 1, 2, 3, ...

Membros de Classe

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;

    public Bicycle(int cadence, int speed, int gear){
        this.gear = gear;
        this.cadence = cadence;
        this.speed = speed;
        id = ++numberOfBicycles;
    }

    public int getID() {
        return id;
    }
    ...
}
```

Membros de Classe

- Assim como campos, é possível definir métodos de classe (ou métodos estáticos)
- Valem as mesmas regras para campos
  - Métodos estáticos existem independentemente das instâncias
  - A chamada à métodos estáticos deve ser feita pelo nome da classe (convenção)
- Uma aplicação comum de métodos estáticos é o acesso à campos estáticos

```
public static int getNumberOfBicycles() {
    return numberOfBicycles;
}
```

Membros de Classe

- Considerações importantes sobre acessos entre membros de instância e de classes
  - Métodos de instância PODEM acessar variáveis e métodos de instância diretamente
  - Métodos de instância PODEM acessar variáveis e métodos de classe diretamente
  - Métodos de classe PODEM acessar variáveis e métodos de classe diretamente
  - Métodos de classe **NÃO PODEM** acessar variáveis e métodos de instância diretamente
  - Métodos de classe **NÃO PODEM** usar a palavra-chave *this*, pois não há instância que *this* deva representar

Membros de Classe

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;
    private int id;
    private static int numberOfBicycles = 0;

    public Bicycle(int cadence, int speed, int gear){
        this.gear = gear;
        this.cadence = cadence;
        this.speed = speed;
        id = ++numberOfBicycles;
    }

    public static void printBikeId() {
        System.out.println("Bike number " + id);
    }
    ...
}
```

Membros de Classe

- Constantes
  - A combinação dos modificadores *static* e *final* é usada para criar constantes
  - Modificador *final* indica que a variável não pode ser alterada
    - Seu valor deve ser definido junto com a declaração
    - Tentar alterar uma constante gera erro de compilação
  - Por que usar modificar *static* para definir constantes?
    - Cada instância teria uma variável constante

Inicializando Variáveis de Classe

- A inicialização de variáveis de instâncias pode ser feita de duas formas
  - Associando valores à variável no mesmo comando de sua declaração
  - Dentro de um construtor

```
public class Bicycle {
    private int cadence = 10;
    private int gear = 0;
    private int speed = 0;

    public Bicycle() {}
}
```

```
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;

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

Inicializando Variáveis de Classe

- A inicialização de variáveis de instâncias pode ser feita de duas formas
  - Associando valores à variável no mesmo comando de sua declaração
  - Dentro de um construtor
- Para variáveis de classe (*static*), só a primeira opção é possível, já que construtores só fazem sentido para instâncias
- O que fazer quando uma variável de classe exige uma inicialização um pouco mais complexa?
  - Por exemplo, copiar elementos de um array

Inicializando Variáveis de Classe

- Java provê um bloco de inicialização estático

```
static {
    // whatever code is needed for initialization goes here
}
```

- Uma classe pode conter vários blocos deste tipo
- Eles podem estar em qualquer posição no corpo da classe
- O compilador irá chamar todos os blocos estáticos, na ordem em que foram definidos

Inicializando Variáveis de Classe

- Uma forma alternativa ao bloco estático é usar um método privado e estático
  - A vantagem é que o método pode ser chamado novamente em alguma situação futura, para reinicializar as variáveis de classe

```
class Whatever {
    public static varType MY_VAR = initializeClassVariable();

    private static varType initializeClassVariable() {
        // initialization code goes here
    }
}
```

Início do Programa

- Todo projeto em Java precisa de um método público e estático chamado **main**
- Este método deve ser declarado dentro de alguma classe
- Contudo, por ser estático, não há necessidade de uma instância para chamá-lo
  - No início do programa, não há instâncias de nenhuma classe

```
public class Principal {
    public static void main(String[] args) {
        // the program starts here
    }
}
```

Pacotes

- A estrutura de pacotes é importante para organizar classes relacionadas de alguma forma
  - Lembre-se que os modificadores protected e *package-private* estão relacionados aos pacotes
  - Facilita para o programador na hora de buscar por classes que realizam determinada tarefa
  - Exemplos:
    - java.lang
    - java.io
  - Duas (ou mais) classes podem ter o mesmo nome se estiverem em pacotes diferentes

Pacotes

- Convenção para nomes
  - Todas as letras minúsculas
  - Empresas devem criar uma estrutura de pacotes que use o endereço web ao contrário
  - Exemplo: www.example.com
    - Se a empresa acima criar um pacote chamado meupacote, ele seria criado em com.example.meupacote
  - Todos os pacotes da API java estão em java. ou javax.

Pacotes

- O pacote do qual a classe faz parte precisa ser declarado no início do código-fonte que define a classe
- Um pacote também reflete a estrutura de diretórios do código fonte
  - Essa estrutura deve ser respeitada, caso contrário o código não compila

```
package vehicle;

public class Bicycle {
    // class body
}
```

Importação de Pacotes e Classes

- Para usar uma classe é preciso importá-la, colocando todo seu caminho na estrutura dos pacotes
  - Importações vem depois do comando package

```
package vehicle;

import java.util.Scanner;

public class Bicycle {
    Scanner sc = new Scanner(System.in);
    …
}
```

Importação de Pacotes e Classes

- Para usar uma classe é preciso importá-la, colocando todo seu caminho na estrutura dos pacotes
  - Importações vem depois do comando package
  - Também é possível importar TODAS as classe de um pacote de uma vez

```
package vehicle;

import java.util.*;

public class Bicycle {
    Scanner sc = new Scanner(System.in);
    …
}
```

JavaDoc

- A documentação do código é identificada por um tipo especial de comentário
  - /** */
- Cada entrada deve ser colocada logo antes de
  - Definição da classe
  - Campos
  - Construtores
  - Métodos
- As entradas do JavaDoc são compostas por
  - Descrição
  - Bloco de tags

JavaDoc

```
/**
 * Returns an <code>Image<code> object that can then be painted on
 * the screen. The <code>url</code> argument must specify an absolute
 * {@link URL}. The <code>name</code> argument is a specifier that
 * is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
 public Image getImage(URL url, String name) {
     try {
         return getImage(new URL(url, name));
     } catch (MalformedURLException e) {
         return null;
     }
 }
```

JavaDoc

Method Detail

getImage

```
public java.awt.Image getImage(java.net.URL url,
                               java.lang.String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute `URL`. The `name` argument is a specifier that is relative to the url argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

Parameters:

`url - an absolute URL giving the base location of the image`

`name - the location of the image, relative to the url argument`

Returns:

`the image at the specified URL`

See Also:

`Image`

JavaDoc

- A primeira linha da entrada JavaDoc será usada na tabela que resume os métodos
- Tags de HTML podem ser usadas dentro do JavaDoc, uma vez que o texto será convertido para HTML
- A primeira linha que começar com “@” indica que o bloco de descrição terminou
- Separe a descrição das tags finais por uma linha vazia

JavaDoc

Tag & Parameter	Usage	Applies to	Since
@author *John Smith*	Describes an author.	Class, Interface, Enum	
@version *version*	Provides software version entry. Max one per Class or Interface.	Class, Interface, Enum	
@since *since-text*	Describes when this functionality has first existed.	Class, Interface, Enum, Field, Method	
@see *reference*	Provides a link to other element of documentation.	Class, Interface, Enum, Field, Method	
@param *name description*	Describes a method parameter.	Method	
@return *description*	Describes the return value.	Method	
@exception *classname description* **@throws** *classname description*	Describes an exception that may be thrown from this method.	Method	

JavaDoc

@deprecated *description*	Describes an outdated method.	Class, Interface, Enum, Field, Method	
{**@inheritDoc**}	Copies the description from the overridden method.	Overriding Method	1.4.0
{**@link** *reference*}	Link to other symbol.	Class, Interface, Enum, Field, Method	
{**@value** *#STATIC_FIELD*}	Return the value of a static field.	Static Field	1.4.0
{**@code** *literal*}	Formats literal text in the code font. It is equivalent to <code> {@literal}</code>.	Class, Interface, Enum, Field, Method	1.5.0
{**@literal** *literal*}	Denotes literal text. The enclosed text is interpreted as not containing HTML markup or nested javadoc tags.	Class, Interface, Enum, Field, Method	1.5.0

JavaDoc

```
// import statements

/**
 * This class is intended to illustrate the JavaDoc structure only.
 *
 * @author Firstname Lastname <address @ example.com>
 * @version 1.6 (current version number of program)
 * @since 1.2 (the version of the package this class was first added to)
 */
public class Test {
    /**
     * Description of the variable here.
     */
    private int debug = 0;

    // remaining class body
}
```

Classes Aninhadas

- Em Java, é possível declarar uma classe dentro de outra classe

```
class OuterClass {
    ...

    class InnerClass {
        ...
    }
}
```

- A classe aninhada é um membro da classe externa
  - Pode ser declarada com qualquer tipo de acesso
  - A classe aninhada tem acesso a todos os membros da classe externa, mesmo os privados

Classes Aninhadas

- Por que usar classes aninhadas?
  - Maneira lógica de agrupar classes que são usadas apenas por uma outra classe
    - Classe auxiliar
  - Aumenta o encapsulamento
    - As classes aninhadas tem acesso total aos membros da classe que a possui
    - Assim, não é preciso relaxar o acesso ao mundo externo
  - Melhor manutenção e entendimento do código

Classes Aninhadas

- Objetos de classes aninhadas existem dentro de uma instância da classe externa
  - Lembre-se que a classe aninhada é considerada um membro da classe externa
- Para instanciar uma classe aninhada é preciso utilizar uma instância (objeto já alocado) da classe externa

```
OuterClass outerObj = new OuterClasse()
OuterClass.InnerClass innerObj = outerObj.new InnerClass();
```

```
public class Class1 {
    protected InnerClass1 ic;

    public Class1() {
        ic = new InnerClass1();
    }
    public void displayStrings() {
        System.out.println(ic.getString() + ".");
        System.out.println(ic.getAnotherString() + ".");
    }
    public static void main(String[] args) {
        Class1 c1 = new Class1();
        c1.displayStrings();
    }

    protected class InnerClass1 {
        public String getString() {
            return "InnerClass1: getString invoked";
        }
        public String getAnotherString() {
            return "InnerClass1: getAnotherString invoked";
        }
    }
}
```

Classes Aninhadas

- Existem outros tipos de classes aninhadas, que não trataremos aqui
  - Classes aninhadas estáticas
  - Classes locais
    - Declaradas no corpo de um método
  - Classes anônimas
    - Também declaradas no corpo de um método, mas sem definição de nome

Tipo *enum*

- Tipo especial de dado, que define os possíveis valores associados ao tipo
  - Variáveis de um tipo *enum* só pode assumir os valores pré-definidos
  - Por se tratarem de constantes, são declarados em maiúscula
  - Valores de *enum* podem ser usados em *switch*

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

Tipo *enum*

- O tratamento de *enum* no Java é poderoso
  - O corpo de um *enum* podem conter campos e métodos
  - O compilador Java adiciona alguns métodos especiais quando um *enum* é criado
  - Ex: método estático **values()**, que retorna um array com todos os valores definidos no *enum*
  - Contudo, diferentemente de classes normais, todo enum herda a classe java.lang.Enum implicitamente

Resumo

- 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

Dúvidas?


TIME COST
STRATEGY A
STRATEGY B
ANALYZING WHETHER STRATEGY A OR B IS MORE EFFICIENT
THE REASON I AM SO INEFFICIENT