

Orientação a Objetos em Java

Classe x Objetos



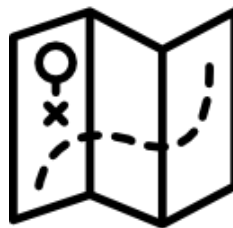
Classe
(Abstração)

Objetos (Instâncias)
da Classe Carro



Onde Encontrar Objetos?

- O sistema de software é baseado na **interpretação do mundo**
 - Os objetos que compõem o software devem ser a representação dos objetos relevantes que constituem o mundo exterior
 - Devem mapear os objetos reais em objetos computacionais e escrever programas que dão vida a estes objetos em um sistema computacional
- Resumindo:
 - “os objetos estão por aí; é só pegá-los.”



Como Descrever Objetos?

- Objetos são descritos a partir de **classes**
- Uma classe contém a definição das **características** que os objetos daquela classe terão:
 - Definição dos atributos
 - Definição dos métodos



Classe

- Classes definem um **novo tipo de dado e como ele é implementado**
- Definem as características que os objetos daquela classe terão
- Definições:



Atributos

- *São os dados, as variáveis relacionadas àquela classe*



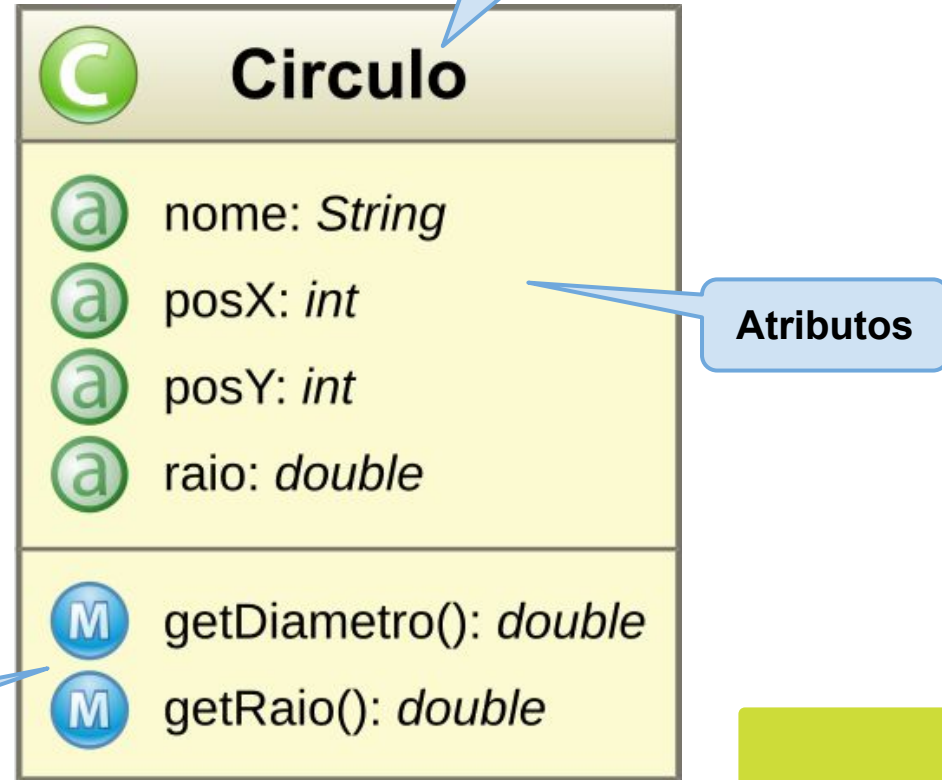
Métodos

- *são as funções, operações que os objetos daquela classe poderão executar*



Diagrama de Classes

- Diagrama de Classes é um diagrama da **UML** (*Unified Modeling Language*) usado para representar a **estrutura das classes** de um aplicativo e as relações entre elas



Coesão e Acoplamento



- **Uma Classe** descreve as características e o comportamento dos seus objetos. **É uma pequena parte do sistema**, um módulo
- Como todo módulo, uma característica importante de uma classe é que ela tenha uma única responsabilidade
 - Esta característica é conhecida como **coesão**
 - Classes coesas não assumem responsabilidades que não são suas. Ou seja, não possuem métodos ou atributos não relacionadas diretamente ao seu objetivo
- Além disso, uma classe deve, na medida do possível, depender o mínimo de outras classes
 - Esta característica é conhecida como **acoplamento**
 - Classes fortemente acopladas dificultam o reuso e a manutenção, pois mudanças em uma delas afetarão as outras



Classes em Java

- Sintaxe para descrever uma classe em Java

```
class NomeDaClasse {  
  
    // Atributos  
  
    // Métodos  
  
}
```

Nomes de classes começam com letra maiúscula e seguem este padrão, conhecido como CamelCase





Classes em Java

■ Sintaxe para descrever uma classe em Java

```
class Circulo {  
    String nome;  
    int posX, posY;  
    double raio;  
  
    double getDiametro() {  
        return 2 * raio;  
    }  
  
    double getArea() {  
        return 3.14159 * raio * raio;  
    }  
}
```

Classe Circulo

Atributos

Método getDiametro()

Método getArea()

Objetos

- Objetos são as **instâncias de uma classe**
- São criados em **tempo de execução**
 - Eles **ocupam** espaço na **memória**
 - Esses espaços são alocados quando um novo objeto de uma classe é criado
 - São **alocados dinamicamente**, assim como a função *malloc* do C
 - São **desalocados automaticamente** quando o objeto não está mais em uso
- Um objeto é criado (e alocado) apenas através do operador `new`

O Objetos

■ Instanciando a Classe Circulo (i.e., criando um objeto)

```
class Principal {  
    public static void main(String args[]) {  
        Circulo circ = new Circulo();  
        circ.nome = "FN-2187";  
        circ.posX = 7;  
        circ.posY = 3;  
        circ.raio = 2.5;  
  
        System.out.println("O circulo circ está em ("  
            + circ.posX + ", " + circ.posY  
            + "). Possui raio " + circ.raio  
            + " e area " + circ.getArea());  
    }  
}
```

Instanciando a classe
Circulo e criando o
objeto circ

Modificando atributos
do objeto circ

Acessando o atributo
posY

Executando o método
getArea()

Objetos

- A seguinte linha realiza **quatro operações**:

```
Circulo circ = new Circulo();
```

- Declaração
 - *circ* será um objeto da classe *Circulo*
 - Declarações não criam objetos, apenas declaram
- Instanciação
 - *new* é um operador que cria dinamicamente um novo objeto na memória
- Inicialização
 - Chamada ao construtor da classe *Circulo* (veremos mais adiante)
- Atribuição
 - A referência do objeto retornada pelo operador *new* é atribuída à variável *circ*



Atributos



Atributos

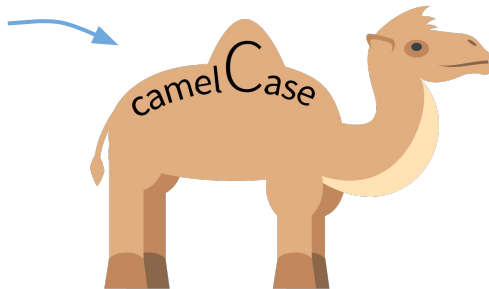
- Atributos são os **dados** relacionados à classe
 - São variáveis dentro do **escopo de uma classe**
 - Servem para armazenar informações relacionadas
- Cada atributo possui um **tipo**, um **nome** e, opcionalmente, **modificadores** de acesso (este último será visto em outras aulas)
- Exemplos:

```
int anoPublicacao;  
double raio;  
float notaFinal = 9.8f;  
String nomeMestre = "Yoda";
```

Atributos

■ Padrão usado nos **nomes dos atributos**

- Na especificação do Java (*Cap. 6, Names*) há algumas **convenções** de nomes
- Usar **nomes completos** ou mesmo **frases**
- Separar palavras compostas usando **camelCase**
 - *A primeira palavra começa com letra minúscula*
 - *As outras palavras começam com letra maiúscula*



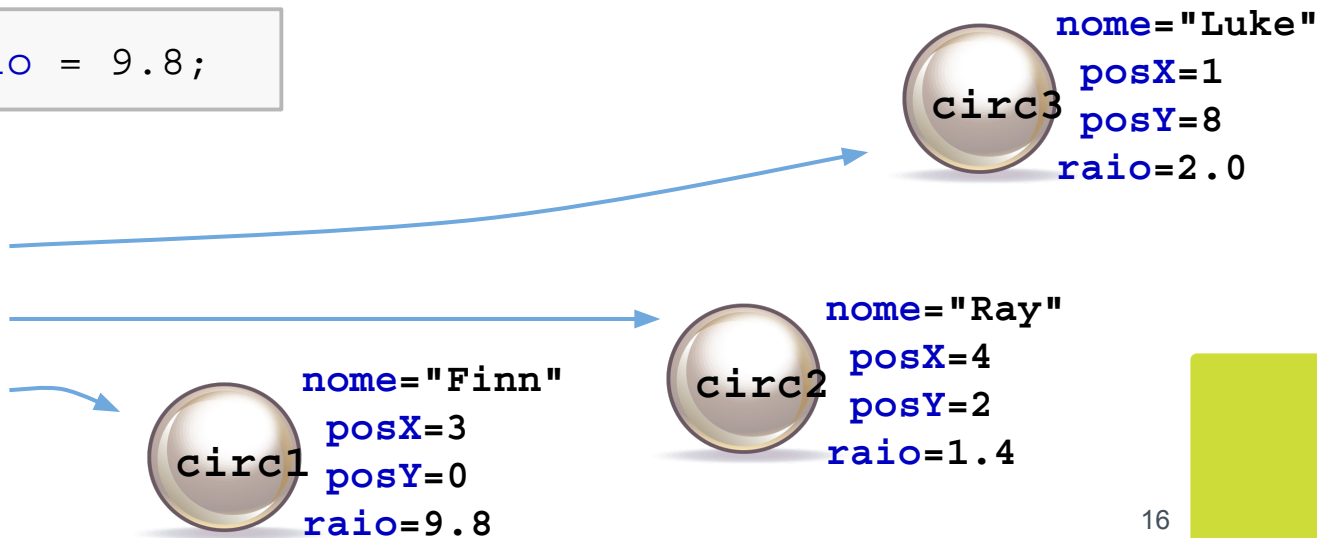
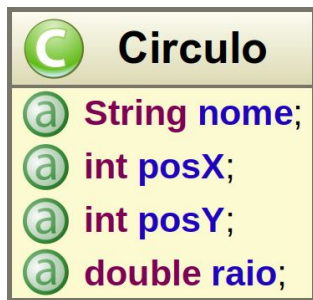
■ Atributos podem ser

- Atributos de **Instância** (padrão, próximo slide)
- Atributos **Estáticos** (ou Atributos de Classe, vistos mais adiante)

Atributos de Instância

- O atributo pertence ao **objeto** (instância da classe)
 - Cada **objeto** instanciado terá **uma região na memória** para o atributo
 - Cada objeto poderá ter um valor diferente (próprio) para o atributo
 - São acessados usando o nome do objeto

```
circ1.raio = 9.8;
```





Atributos Estáticos

- O atributo pertence à **classe** (atributo de classe)
 - Tem apenas **uma posição na memória**
 - **Todos** os objetos da classe **enxergam** e acessam **o mesmo valor**
 - Se um objeto muda o valor do atributo, ele muda em todos os objetos da classe
 - São declarados usando o modificador `static`

```
static int numCirculos;
```

- São acessados usando o nome da classe

```
Circulo.numCirculos = 3;
```

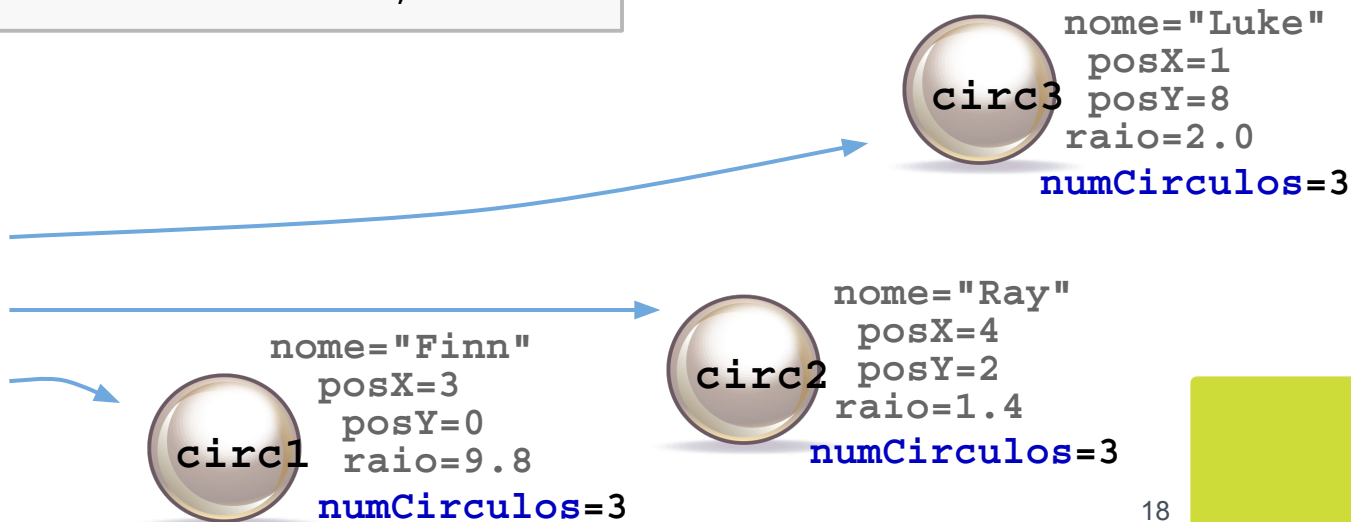
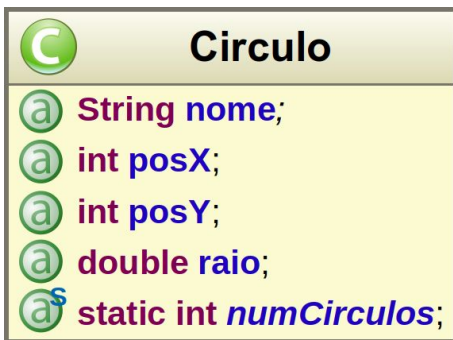


Atributos Estáticos

Exemplo

- Ao atribuir o valor 3 para o atributo estático `numCirculos`, todos os objetos dessa mesma classe enxergam esse mesmo valor

```
Circulo.numCirculos = 3;
```



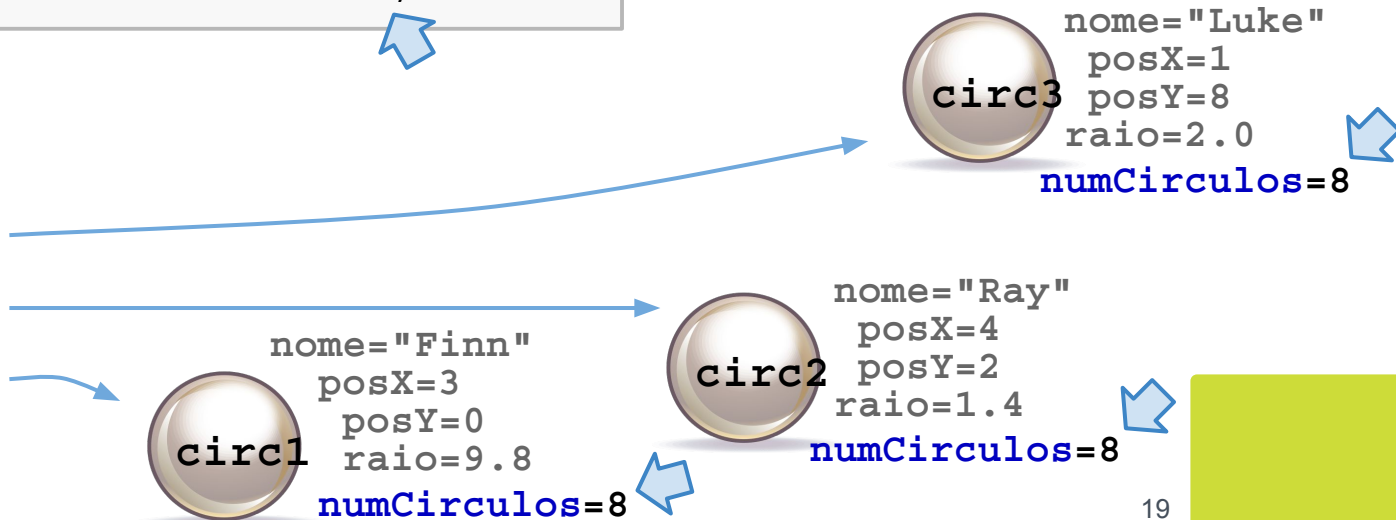
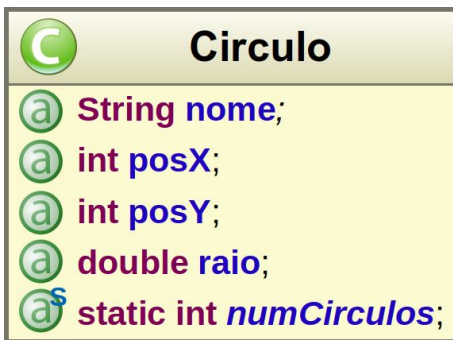


Atributos Estáticos

Exemplo

- Ao mudar o valor do atributo estático `numCirculos`, para 8, o valor desse atributo muda em todos os objetos, pois todos enxergam o mesmo, único, valor na memória

```
Circulo.numCirculos = 8;
```





Atributos Constantes

- Uma constante é um atributo cujo **valor não muda**, após ser atribuído
 - São declarados usando o modificador de acesso `final`
 - Não precisam, necessariamente, serem inicializados durante a declaração
 - Mas uma vez inicializados, seu valor não muda
- Em geral, constantes são também atributos estáticos (de classe) e públicos (veremos futuramente). Isso facilita o acesso à constante.
- São normalmente utilizados para armazenar **valores utilitários**
- Convenção dos nomes:
 - Nomes de constantes são escritos todos em **letras maiúsculas**
 - E usando underscores (`_`) para separar palavras compostas



Atributos Constantes

Exemplos:

- Algumas constantes do código-fonte da classe `Math`

```
public static final double E = 2.7182818284590452354;  
public static final double PI = 3.14159265358979323846;
```

- Algumas constantes do código-fonte da classe `Integer`

```
public static final int MIN_VALUE = 0x80000000;  
public static final int MAX_VALUE = 0x7fffffff;  
public static final int SIZE = 32;  
public static final int BYTES = SIZE / Byte.SIZE;
```

Acessando o atributo
constante `SIZE` da
classe `Byte`



Atributos e o Estado de um Objeto

- O estado de um objeto são os valores atuais de seus atributos
- Por exemplo:
 - Um objeto da classe `Circulo` possui `raio = 8`.
 - O valor do atributo `raio` (8), em conjunto com os valores dos outros atributos do círculo, indicam o **estado atual do objeto**
 - Se mudarmos o raio deste objeto para 42, **mudamos o estado do objeto** que passa a ser este valor (42) em conjunto com o valor dos outros atributos.
- O estado de um objeto pode mudar quando
 - Alteramos o valor de um atributo diretamente; ou quando
 - Executamos um método do objeto (que altera o valor de algum atributo)



Métodos



Métodos

- Métodos são as operações que os objetos da classe podem executar
 - São funções ou procedimentos
 - Executam um algoritmo
 - Acessam e modificam atributos
 - Retornam um valor ou `void`
 - Podem possuir variáveis locais
 - *Os nomes das variáveis locais seguem as mesmas regras dos nomes dos atributos, mas em geral são menores (acrônimos, abreviações, etc)*
- Convenções de nomes
 - Os nomes dos métodos possuem as mesmas convenções de nomes dos atributos
 - Entretanto, em geral, são verbos



Métodos

- ▣ Cada método possui: **modificadores de acesso** (opcional), um **retorno**, um **nome**, **parâmetros** e uma **implementação**

- ▣ Exemplo da classe `Circulo`

```
double getArea() {  
    return 3.14159 * raio * raio;  
}
```

- ▣ Exemplo da classe `String` (código-fonte do Java)

```
public char charAt(int index) {  
    if ( (index < 0) || (index >= value.length) ) {  
        throw new StringIndexOutOfBoundsException(index);  
    }  
    return value[index];  
}
```



Métodos Estáticos (de Classe)

- Assim como os atributos, também existem **Métodos de Instância** (usados até o momento) e os **Métodos Estáticos** (de Classe)
- Métodos Estáticos
 - São declarados utilizando o modificador de acesso `static`
 - Não precisam de uma instância (objeto) para serem executados
 - *Podem ser executados diretamente usando o nome da classe*
 - Não podem acessar atributos de instância
 - *Só podem acessar atributos estáticos*
 - *Porque? Imagine que não exista nenhuma instância daquela classe na memória, como o atributo será acessado? E se tiver várias instâncias? Qual seria usada?*



Métodos Estáticos (de Classe)

- Exemplos de Métodos Estáticos
 - Chamando o método estático `sqrt` da classe `Math`:

```
// O ponto (a,b) pertence ao círculo?  
boolean pertence(double a, double b) {  
    double dx = a - posX;  
    double dy = b - posY;  
    double dist = Math.sqrt(dx*dx - dy*dy);  
    return (dist <= raio);  
}
```



Métodos Estáticos (de Classe)

Exemplos de Métodos Estáticos

- Declaração do método `sqrt` (raiz quadrada) da classe `Math` (código-fonte do Java)

```
public static double sqrt(double a) {  
    return StrictMath.sqrt(a); // delegates to StrictMath  
}
```

- Declaração do método `sqrt` da classe `StrictMath`

```
public static native double sqrt(double a);
```

- O modificador `native` indica que o método é implementado em C/C++
- No código do OpenJDK, é possível encontrar a chamada à função `sqrt` do C
- No C, `sqrt` é implementado usando uma instrução do processador



Sobrecarga de Métodos

- Em Java, dois ou mais métodos podem ter o mesmo nome!
 - Entretanto, precisam ter parâmetros de tipos/quantidades diferentes
 - Isso é conhecido como **sobrecarga de métodos**
 - *Em inglês, method overloading*
 - O Java diferencia um método do outro observando os **parâmetros**
 - Exemplo:
 - *O método `println` que você tem usado para imprimir texto, classe `PrintStream`*

```
public void println()           { /* ... */ }
public void println(boolean x) { /* ... */ }
public void println(char x)    { /* ... */ }
public void println(float x)   { /* ... */ }
public void println(int x)     { /* ... */ }
public void println(long x)    { /* ... */ }
public void println(Object x)  { /* ... */ }
public void println(String x)  { /* ... */ }
```

Este método é o que
você está usando,
provavelmente



Métodos Especiais

- Em Java, temos dois métodos especiais
 - Método Construtor
 - *Executado quando um novo objeto daquela classe é criado. Método que atribui valores padrões para os atributos de um objeto (dentre outras coisas)*
 - Método Destrutor
 - *Executado quando o coletor de lixo vai remover o objeto da memória. Pouco usado, uma vez que não há garantias de quando/se ele será executado*



Métodos Construtores

- Executado quando um novo objeto daquela classe é criado
 - Utilizado para inicializar os atributos do novo objeto, dentre outras coisas
- Possui o mesmo nome da classe
 - Como nomes de classes começam com letra maiúscula, este método também começará com letra maiúscula
- Permite sobrecarga
 - Podemos ter diversos construtores (com o mesmo nome da classe), desde que estes possuam parâmetros diferentes
- Não retornam valor
 - Estes são os únicos métodos que não retornam valor (nem mesmo `void`)



Métodos Construtores

- Criando construtores para a classe `Circulo`

```
class Circulo {  
    int posX, posY;  
    double raio;  
  
    Circulo() {  
        posX = 0;  
        posY = 0;  
        raio = 0.0;  
    }  
  
    Circulo(double raio) {  
        posX = 0;  
        posY = 0;  
        this.raio = raio;  
    }  
  
    Circulo(int posX, int posY, double raio) {  
        this.posX = posX;  
        this.posY = posY;  
        this.raio = raio;  
    }  
  
    // Métodos getDiametro, getArea ...  
}
```

Este construtor não possui parâmetros. Para criar um objeto usando este construtor:

```
Circulo circ = new Circulo();
```

Este construtor possui um parâmetro do tipo `double`. Para criar um objeto:

```
Circulo circ = new Circulo(1.2);
```

Este `this` serve para diferenciar o atributo `raio` do parâmetro (variável local) `raio`.

Este construtor possui três parâmetros do tipo `double`. Para criar um objeto:

```
Circulo circ =  
    new Circulo(1, 3, 9.8);
```




Encadeamento de Construtores

- Para simplificar e reutilizar código, um construtor pode chamar outro
 - Isso é conhecido como **Encadeamento de Construtores**
 - É a forma **recomendada** de se criar construtores

Este construtor é o principal. Inicializa todos os atributos e será usado pelos outros construtores

```
class Circulo {  
    int posX, posY;  
    double raio;  
  
    Circulo() {  
        this(0, 0, 0.0);  
    }  
  
    Circulo(double raio) {  
        this(0, 0, raio);  
    }  
  
    Circulo(int posX, int posY, double raio) {  
        this.posX = posX;  
        this.posY = posY;  
        this.raio = raio;  
    }  
    // Métodos getDiametro, getArea ...  
}
```

Este construtor usa a palavra reservada `this` para chamar o construtor que possui três parâmetros do tipo `double` (último)

Mesma coisa, mas passando o `raio`



Método Destruitor

■ Método Destruitor

- Executado quando o objeto é removido da memória
 - Chamado pelo coletor de lixo (próximo slide)

■ É implementado através do método `finalize`

```
protected void finalize() { /*...*/ }
```

- ## ■ Na prática, é muito raramente utilizado e deve ser evitado
- Motivo: java não garante se ou quando o método será chamado
 - Se você deixar para fechar um arquivo nele, por exemplo, o arquivo pode não ser devidamente fechado

Coletor de Lixo

- O Coletor de Lixo é uma das **tecnologias chave** do Java
 - Em Java, não podemos liberar memória
 - Tudo é feito automaticamente pelo Coletor de Lixo
- Como o Coletor de Lixo **sabe o que pode** ou não ser **liberado**?
 - Por uma regra bem simples: se um objeto não possui mais referências para ele, então ele pode ser liberado
 - Isso funciona porque Java não permite acesso direto à memória
 - Se um objeto perdeu todas as referências a ele, é impossível acessá-lo e, portanto, sabe-se que ele nunca mais será necessário

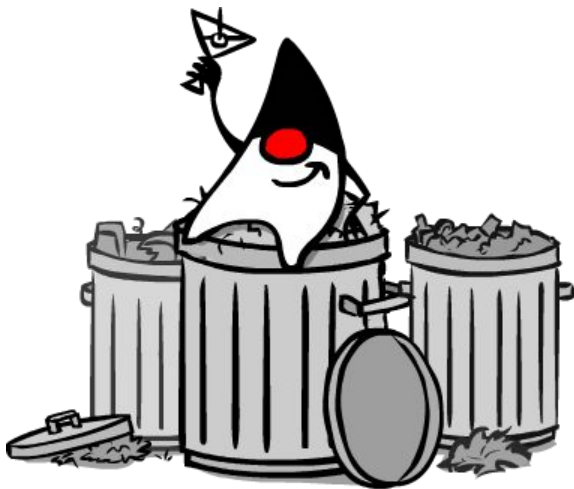


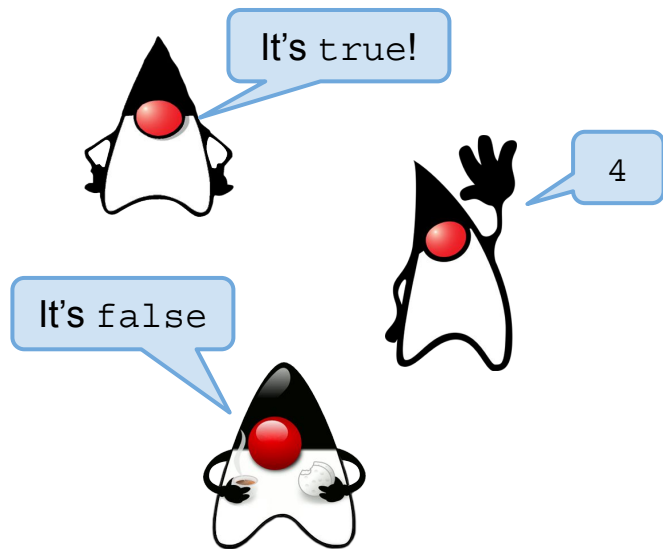
Coletor de Lixo

- Entretanto, se uma **referência é mantida** para um objeto que não é mais necessário, **ele continuará ocupando memória**
 - Para evitar isso, basta atribuir `null` às variáveis que apontam para o objeto
 - No exemplo abaixo, **removemos a referência** que `circ` tinha para o objeto e, na próxima coleta de lixo, ele será liberado

```
circ = null;
```

- Note que, na maioria dos casos, isso não é necessário, pois grande parte dos objetos são alocados dentro de métodos (escopo local) e suas referências são perdidas após o término do método
 - No caso de atributos de um objeto (que são também objetos), estes serão automaticamente desalocados quando o objeto for desalocado*





Tipos de Dados em Java

Tipos de Dados em Java



- Java é uma linguagem com **tipagem estática**
 - Toda variável e toda expressão possui um **tipo** que é **conhecido em tempo de compilação**
- Java é uma linguagem com **tipagem forte**. Tipos:
 - **Limitam os valores** que uma variável pode armazenar
 - Limitam os valores que uma expressão pode produzir
 - **Limitam as operações** suportadas pelos valores
 - Determinam o **significado das operações**

Tipos de Dados em Java

- Tipos em Java são divididos em **duas categorias**:
 - Tipos **Primitivos** e Tipos **Referência**
- Tipos Primitivos:
 - Tipos numéricos inteiros
 - *byte, short, int, long e char*
 - Tipos numéricos de ponto flutuante
 - *float e double*
 - Tipo booleano
 - *boolean*
- Tipos Referência (ou simplesmente “Objetos”)
 - Classes
 - Interfaces
 - Vetores

Tipos Primitivos

- São os tipos “**nativos**” do hardware

<code>byte</code>	Inteiro, 8 bits
<code>short</code>	Inteiro, 16 bits
<code>int</code>	Inteiro, 32 bits
<code>long</code>	Inteiro, 64 bits
<code>float</code>	Ponto flutuante, 32 bits
<code>double</code>	Ponto flutuante, 64 bits
<code>boolean</code>	<code>true</code> ou <code>false</code>
<code>char</code>	Caractere Unicode, 16 bits, sem sinal

- Com exceção do `char` e do `boolean`, todos os tipos primitivos **possuem sinal**
 - Não é possível criar números “unsigned” em Java

Tipos Primitivos: Operadores

- Em Java, podemos usar operadores nas variáveis

Multiplicativos	* / %
Aditivos	+ -
Relacionais	> < >= <=
Igualdade/Diferença	== !=
E Lógico	&&
OU Lógico	
Seleção	e ? e : e
Atribuição	=
Atribuição+Operação	+= -= *= /=
Incremento/Decremento	++ --
Operadores de Bits	>> << & ^ ~

Tipos Primitivos: Inicialização

- **Atributos** de tipos primitivos são automaticamente **inicializados para zero**

- O que acontece no código abaixo?

```
class Pessoa {  
    int    anoNascimento;  
    boolean solteiro;  
  
    public static void main(String args[]) {  
        Pessoa fulano = new Pessoa();  
        System.out.println("AnoNasc = " + fulano.anoNascimento  
                             + ", Solteiro? " + fulano.solteiro);  
    }  
}
```

```
$ javac Pessoa.java  
$ java Pessoa  
AnoNasc = 0, Solteiro? false
```

Tipos Primitivos: Inicialização

- Entretanto, variáveis locais de tipos primitivos **não são inicializados**
 - O que acontece no código abaixo?

```
class Pessoa {  
  
    public static void main(String args[]) {  
        int anoNasc;  
        System.out.println("AnoNasc = " + anoNasc);  
    }  
  
}
```

```
$ javac Pessoa.java  
Pessoa.java:5: variable anoNasc might not have been initialized  
    System.out.println("AnoNasc = " + anoNasc);  
                                ^  
1 error
```

Tipos Referência



- Também chamados de:
 - referências, tipos não-primitivos, ou simplesmente “objetos”
- Armazenam uma referência para um objeto
 - Uma referência é um endereço de memória onde o objeto está armazenado
 - Por usar uma máquina virtual, o endereço armazenado em uma referência não corresponde ao endereço na memória física do computador

Tipos Referência



- ▣ Uma referência a um objeto pode ser do tipo
 - ▣ Classe
 - *Classes já existentes no Java*
 - *Classes criadas pelo usuário*
 - ▣ Interface (veremos futuramente)
 - ▣ Vetor
 - *Em java, um vetor é um objeto*

Tipos de Referências x Tipos Primitivos



- ▣ Variáveis de Tipos Primitivos
 - ▣ São sempre utilizadas “**por valor**”
 - ▣ Quando usadas como parâmetros de métodos ou em atribuições, elas sempre serão uma **cópia** da variável original
 - ▣ Não é possível criar uma referência (ponteiro) para uma variável primitiva
 - ▣ Por isso não existem os operadores `&` ou `*` como na linguagem C/C++

Tipos de Referências x Tipos Primitivos



- ▣ Variáveis de Tipos Referência (objetos de classes)
 - ▣ São sempre utilizadas “**por referência**”
 - ▣ Quando usadas como parâmetros de métodos ou em atribuições, apenas a referência é passada. Portanto, **mudando-se seus atributos dentro do método, mudará os atributos do objeto “original”**
 - ▣ Para se “duplicar” um objeto, deve-se usar o método `clone`


Tipos de Referências x Tipos Primitivos

Exemplo

```
int a = 3;  
char b = 't';  
Circulo c = new Circulo();
```

Var.	Endereço	Conteúdo
a	0200	3
b	0300	't'
c	0400	0900

	0900	posX = 2, posY = 5, raio = 6.0



Tipos de Referências x Tipos Primitivos

- Qual a saída do código?


```
class A {  
    int i = 1;  
  
    public static void main(String[] args) {  
        A a1 = new A();  
        A a2 = new A();  
        System.out.println("a1.i = " + a1.i);  
        System.out.println("a2.i = " + a2.i);  
        System.out.println("a1 == a2 ? " + (a1 == a2) );  
        System.out.println("a1.i == a2.i ? " + (a1.i == a2.i));  
    }  
}
```

```
$ javac A.java  
$ java A  
a1.i = 1  
a2.i = 1  
a1 == a2 ? false  
a1.i == a2.i ? true
```

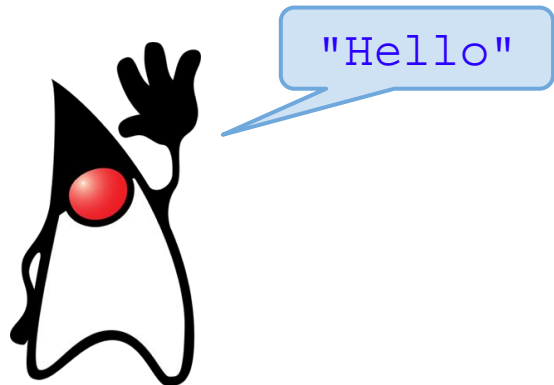
Tipos de Referências x Tipos Primitivos

- Qual a saída do código?

```
class A {  
    int i = 1;  
  
    public static void main(String[] args) {  
        A a1 = new A();  
        A a2 = a1;  
        System.out.println("a1.i = " + a1.i);  
        System.out.println("a2.i = " + a2.i);  
        System.out.println("a1 == a2 ? " + (a1 == a2) );  
        System.out.println("a1.i == a2.i ? " + (a1.i == a2.i));  
    }  
}
```



```
$ javac A.java  
$ java A  
a1.i = 1  
a2.i = 1  
a1 == a2 ? true  
a1.i == a2.i ? true
```



Classe String, um Tipo Especial

String, um Tipo Especial

- Os objetos da classe `String` são especiais e são **tratados de forma diferente**
 - Motivo: performance e facilidade (são frequentemente utilizadas)
- Strings podem ser inicializadas de **duas** formas:
 - Explicitamente através do operador `new` (como qualquer outro objeto):

```
String c = new String("Something, something dark side");
```

- Implicitamente através de uma "**string literal**" (mais comum, o recomendado):

```
String c = "Something, something dark side";
```

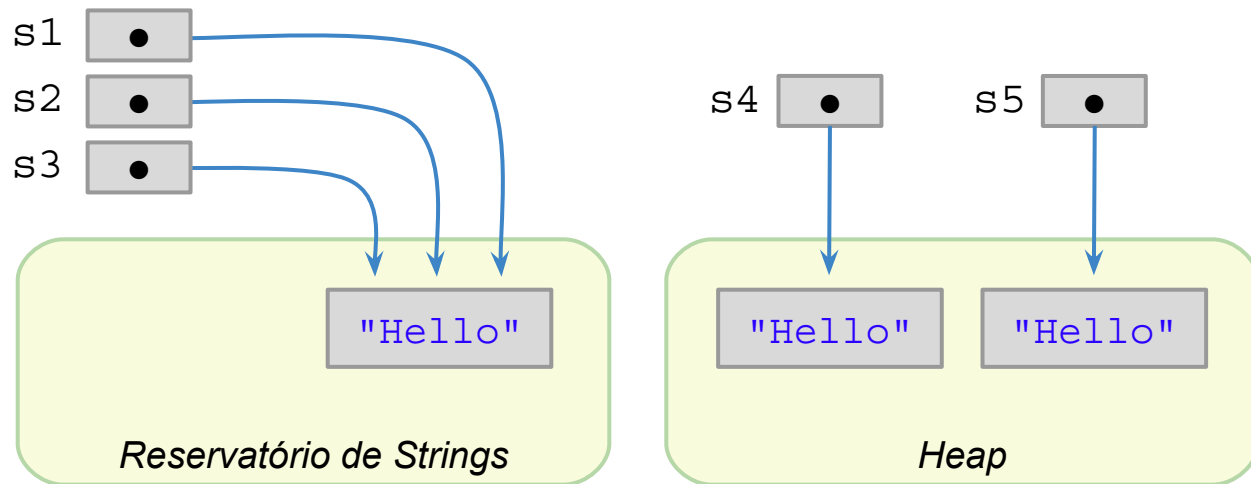
Reservatório de Strings

- **Strings** criadas com o operador `new` são armazenadas junto com os outros objetos no *heap* e se comportam como qualquer outro objeto
- **Strings literais** são armazenadas em um **reservatório de strings**:
 - Diferença: **Strings literais com o mesmo conteúdo, compartilham o mesmo endereço de memória**
- Em ambos os casos:
 - O operador "+" é usado para concatenar strings
 - As **strings são imutáveis**, ou seja, não tem como alterar seu conteúdo
 - *Para “mudar” uma string, deve-se atribuí-la a uma nova string contendo o novo valor. A string antiga, se não estiver mais em uso, será liberada pelo coletor de lixo.*

Reservatório de Strings

Exemplo:

```
String s1 = "Hello";           // String literal
String s2 = "Hello";           // String literal
String s3 = s1;                 // Mesma Ref.
String s4 = new String("Hello"); // Objeto da classe String
String s5 = new String("Hello"); // Objeto da classe String
```



Reservatório de Strings

Exemplo:

```
String s1 = "Hello";           // String literal
String s2 = "Hello";           // String literal
String s3 = s1;                 // Mesma Ref.
String s4 = new String("Hello"); // Objeto da classe String
String s5 = new String("Hello"); // Objeto da classe String
```

```
s1 == s1;           // true, mesma referência
s1 == s2;           // true, mesma referência
s1 == s3;           // true, mesma referência
s1.equals(s3);      // true, mesmo conteúdo
s1 == s4;           // false, referências diferentes
s1.equals(s4);      // true, mesmo conteúdo
s4 == s5;           // false, referências diferentes
s4.equals(s5);      // true, mesmo conteúdo
s1 == "Hello";      // true, mesma referência
s4 == "Hello";      // false, referências diferentes
```

Concatenação de Strings

- Strings podem ser **concatenadas** usando o operador +

```
String p1 = "Omicron";  
String p2 = "Persei";  
String w = p1 + " " + p2;    // Omicron Persei  
int num = 8;  
String w = w + " " + num;    // Omicron Persei 8
```

- O operador + gera uma **nova string** na memória e retorna a referência para ela
- Sempre que o Java encontra o operador + e um dos parâmetros é uma `String`, ele faz a concatenação
- Note que, na última linha, a `String` antiga apontada por `w` não foi “modificada”
 - O que acontece é que, agora, `w` apontará para a nova `String` criada pelo operador +.

Métodos da Classe String

- A classe String possui vários métodos úteis. Por exemplo:

```
boolean equals(Object anObject)
```

```
boolean equalsIgnoreCase(String anotherString)
```

```
int length()
```

```
int indexOf(int ch)
```

```
char charAt(int index)
```

```
boolean contains(CharSequence s)
```

```
boolean startsWith(String suffix)
```

```
boolean endsWith(String suffix)
```

```
String replace(CharSequence target, CharSequence replacement)
```

```
public String[] split(String regex, int limit)
```

```
String toLowerCase()
```

```
String toUpperCase()
```

Laboratório

- Disponível no ColabWeb
 - bit.ly/pp-colabweb

