CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS

ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

TIPOS

- Um tipo é um conjunto de valores e suas operações.
- Um tipo pode ser um *tipo primitivo* ou um *tipo de referência*

Tipos primitivos

byte	8-bits $[-128; 127]$
short	16-bits $[-32.768; 32.767]$
int	32-bits $[-2^{31}; 2^{31} - 1]$
long	64-bits $[-2^{63}; 2^{63} - 1]$
float	ponto flutuante de precisão simples de 32-bits
double	ponto flutuante de precisão dupla de 64-bits
boolean	`true` ou `false`
char	1 caractere Unicode de 16-bits

TIPOS PRIMITIVOS “DEFICIENTES”

- `byte`, `short` — use `int` sempre que possível
  - `byte` deveria ter sido definido como `unsigned`
- `float` — use `double` sempre que possível
  - float fornece precisão muito baixa
  - há poucos casos onde usar float realmente vale a pena, ex: vetores muito grandes em sistemas com recursos muito restritos

- Um tipo é um conjunto de valores e suas operações.
- Um tipo pode ser um *tipo primitivo* ou um *tipo de referência*

Tipos de referência
Um tipo de referência pode ser:

- um *tipo de classe*, definido quando uma classe é declarada
- um *tipo de interface*, definido quando uma interface é declarada
- um *tipo de vetor*
- um *tipo de enum*

E pode ter como valor:

- uma referência a um objeto ou vetor
- o valor especial `null`, que representa um objeto nulo (ou a ausência de um objeto)

JAVA TEM UM SISTEMA DE TIPOS BIPARTITE

Primitivos	Tipos de referência
`int, long, byte, short, char, float, double, boolean`	Classes, interfaces, vetores, enums, anotações
Não possuem identidade, apenas valores	Tem identidade além de valor
Imutáveis	Alguns imutáveis, outros não
Na pilha, existem apenas quando usadas	No *heap*, passível de desalocação pelo coletor de lixo
Muito barato	Mais custoso

COMPARAÇÃO DE VALORES

- `x == y` compara x e y "diretamente":

  valores primitivos: devolve `true` se x e y possuírem o **mesmo valor**

  referências a objetos: devolve `true` se x e y **referirem ao mesmo objeto**

- `x.equals(y)` compara os **valores** dos objetos referenciados por x e por y

VERDADEIRO OU FALSO?

```java
int i = 17;
int j = 17;
System.out.println(i == j);
```

VERDADEIRO OU FALSO?

```java
int i = 17;
int j = 17;
System.out.println(i == j);
```

true

VERDADEIRO OU FALSO?

```java
int i = 17;
int j = 17;
System.out.println(i == j);
```

true

```java
String s = "teste";
String t = s;
System.out.println(s == t);
```

VERDADEIRO OU FALSO?

```java
int i = 17;
int j = 17;
System.out.println(i == j);
```

true

```java
String s = "teste";
String t = s;
System.out.println(s == t);
```

true

VERDADEIRO OU FALSO?

```java
int i = 17;
int j = 17;
System.out.println(i == j);
```

true

```java
String s = "teste";
String t = s;
System.out.println(s == t);
```

true

```java
String u = "Daniel";
String v = u.toLowerCase();
String w = "daniel";
System.out.println(v == w);
```

VERDADEIRO OU FALSO?

```java
int i = 17;
int j = 17;
System.out.println(i == j);
```

true

```java
String s = "teste";
String t = s;
System.out.println(s == t);
```

true

```java
String u = "Daniel";
String v = u.toLowerCase();
String w = "daniel";
System.out.println(v == w);
```

indefinido (mas falso na prática)

MORAL DA HISTÓRIA

- Sempre use `.equals()` para comparar referências a objetos!
  - (exceto no caso de enums, que são especiais)
  - O operador `==` pode falhar silenciosamente e imprevisivelmente quando aplicado a referências de objetos
  - O mesmo vale para `!=`

TIPOS DE VETORES

- `t[]`, onde `t` é um tipo de referência
- um tipo de vetor `t[]` é um tipo de referência
- ou seja, `t[]` pode ser tanto `null` quanto uma referência a um vetor cujos elementos são do tipo `t`
- o tamanho do vetor pode ser obtido no campo `length`.
  Ex: `for(int i=0; i < t.length; i++) {}`

Exemplos de variáveis com esses tipos:

- `String[] args`
- `int[] listaDeInteiros = {0, 1, 1, 2, 3, 5, 8}`
- `double[] reais = new double[7]`

BOXING

- Para cada tipo primitivo, há uma classe *wrapper* (invólucro) correspondente que, por sua vez, é um tipo de referência
- Uma instância de um objeto de uma classe *wrapper* contém um único valor do tipo primitivo correspondente
- Um *wrapper* deve ser usado quando um valor de um tipo primitivo é passado para um método que espera um tipo de referência, ou quando um valor primitivo é armazenado em uma variável ou campo de um tipo de referência
- Java automaticamente faz o *boxing* (transformação do tipo primitivo para o tipo do *wrapper*) e o *unboxing* (transformação contrária)
- Você pode fazer isso explicitamente usando operações como `new Integer(i)` ou `o.intValue()`

Tipos primitivos e seus *wrappers*

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

Exemplos:

```
Character ch = 'a';

Boolean bb1 = false, bb2 = !bb1; // Boxing para [false] [true]
Integer bi1 = 117; // Boxing para [117]
Double bd1 = 1.2; // Boxing para [1.2]
boolean b1 = bb1; // Unboxing, resulta false
if (bb1) // Unboxing, resulta false
  System.out.println("Not true");
int i1 = bi1 + 2; // Unboxing, resulta 119
// short s = bi1; // Ilegal
Integer bi2 = bi1 + 2; // Unboxing, boxing, resulta [119]
Integer[] biarr = { 2, 3, 5, 7, 11 };
int sum = 0;
for (Integer bi : biarr)
  sum += bi; // Unboxing no corpo do loop
for (int i : biarr) // Unboxing no cabeçalho do loop
  sum += i;
```

PRIMITIVOS BOXED

- Contêineres imutáveis para os tipos primitivos
- Permite você “usar” primitivos nos contextos que pedem objetos
- **Caso típico de quando você usa coleções**
- **Não os use se puder evitar!**
- Deixe a linguagem fazer o *autoboxing* e o *auto-unboxing* para você
  - esconde, mas não elimina a distinção
  - cuidado!

O QUE O FRAGMENTO DE CÓDIGO ABAIXO FAZ?

```
int[] a = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int i;
int sum1 = 0;
for (i = 0; i < a.length; i++) {
  sum1 += a[i];
}

int j;
int sum2 = 0;
for (j = 0; i < a.length; j++) {
  sum2 += a[j];
}

System.out.println(sum1 - sum2);
```

O QUE O FRAGMENTO DE CÓDIGO ABAIXO FAZ?

```
int[] a = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int i;
int sum1 = 0;
for (i = 0; i < a.length; i++) {
  sum1 += a[i];
}

int j;
int sum2 = 0;
for (j = 0; i < a.length; j++) { // Erro de copy/paste
  sum2 += a[j];
}

System.out.println(sum1 - sum2);
```

Você esperaria que fosse impresso 0, mas ele imprime 55

VOCÊ PODE CORRIGIR O CÓDIGO ASSIM:

```java
int[] a = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

int i;
int sum1 = 0;
for (i = 0; i < a.length; i++) {
  sum1 += a[i];
}

int j;
int sum2 = 0;
for (j = 0; j < a.length; j++) {
  sum2 += a[j];
}

System.out.println(sum1 - sum2); // Imprime 0, como esperado
```

MAS ASSIM É MUITO MELHOR:

Expressão idiomática de Java para um laço

```java
int sum1 = 0;
for (int i = 0; i < a.length; i++) {
  sum1 += a[i];
}

int sum2 = 0;
for (int i = 0; i < a.length; i++) {
  sum2 += a[i];
}

System.out.println(sum1 - sum2); // Imprime 0, como esperado
```

- Reduz o escopo da variável usada como índice do laço
- Mais curto e menos sujeito a erro

ASSIM É AINDA MELHOR!

Laço no estilo for-each

```
int sum1 = 0;
for (int x : a) {
  sum1 += x;
}

int sum2 = 0;
for (int x : a) {
  sum2 += x;
}

System.out.println(sum1 - sum2); // Imprime 0, como esperado
```

- Elimina completamente o escopo da variável usada como índice do laço
- Ainda mais curto e menos sujeito a erro

LIÇÕES APRENDIDAS COM A PERGUNTA

- **Minimize o escopo de suas variáveis locais**
  - declare a variável apenas no local onde for usá-la
- Inicialize as variáveis quando declará-las
- Prefira laços do tipo for-each a laços `for` tradicionais
- Use as expressões idiomáticas da linguagem
- Muito cuidado com mau cheiros de código
  - tais como uma variável de índice declarada fora do laço

O QUE É UM PACOTE?

Definição:
Um pacote é um espaço de nomes que organiza um conjunto de classes e interfaces relacionadas.

```
package bicicleta;

interface Bicicleta { ... }

class BicicletaBásica implements Bicicleta { ... }

class MountainBike extends BicicletaBásica { ... }
```

Fully qualified name de uma classe:
`bicicleta.MountainBike`

TIPOS DE ENUM

- Java tem enums orientados a objeto
- Na sua forma mais simples, são como enums de C:

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

- Mas tem muitas outras vantagens:
  - Checagem de tipo em tempo de compilação
  - Múltiplos tipos enum podem compartilhar os mesmos nomes de valores
  - Permite adicionar ou reordenar sem quebrar o código que usa o enum
  - Podem usar os métodos de `Object`
  - Algumas classes que usa enum são muito rápidas (ex: `EnumSet`, `EnumMap`)
  - Permite percorrer todas as constantes de um enum

EXEMPLO

```
public class EnumTest {
    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;

            case FRIDAY:
                System.out.println("Fridays are better.");
                break;

            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;

            default:
                System.out.println("Midweek days are so-so.");
                break;
        }
    }
```

Variáveis (estáticas) de classe

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

    // adiciona uma variável de instância para o ID do objeto
    private int id;

    // adiciona uma variável de classe para o número de
    // objetos Bicycle instanciados
    private static int numberOfBicycles = 0;
}
```

- São associadas à classe e não a cada objeto
- Podem ser referenciadas pelo nome da classe `Bicycle.numberOfBicycles` ou por uma referência a um objeto `myBike.numberOfBicycles` (desaconselhado)

Métodos de classe

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

- Java permite métodos estáticos assim como variáveis
- Devem ser invocadas com o nome da classe, sem a necessidade de criar uma instância da classe `Bicycle.getNumberOfBicycles()` (também pode ser chamado de uma instância, mas é desaconselhado)

MEMBROS DE CLASSE

Nem toda combinação de variáveis e métodos de instância e de classe podem ser usadas:

- Métodos de instância podem acessar variáveis de instância e métodos de instância diretamente
- Métodos de instância podem acessar variáveis e métodos de classes diretamente
- Métodos de classe podem acessar variáveis de classe e métodos de classe diretamente
- Métodos de classe **não podem** acessar variáveis e métodos de instância diretamente, eles precisam usar uma referência a um objeto
  - também não podem usar `this`, já que não há uma instância de objeto para ser referenciada

FINALMENTE É POSSÍVEL ENTENDER O CÓDIGO ABAIXO

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

- main é `public` porque precisa ser invocada por uma classe externa (a própria JVM)
- main é `static` porque precisa ser invocada quando ainda não há nenhuma instância objeto (na inicialização)
- `System` é uma classe, então `System.out` é uma variável de classe (estática)
- `args` é um vetor com referências a objetos do tipo `String`

CONSTANTES

- O modificador `final` indica que o valor de um campo não pode ser mudado depois de definido
- Junto com `static`, `final` é usado para definir constantes
- A convenção de nome é que o nome de constantes é escrito em letras maiúsculas, e se a palavra for composta usa-se *underscore* para separar

```
static final double MEU_PI = 3.141592653589793
```

Forma de metadados
Provê informação sobre um programa que não faz parte do programa em si; não tem efeito sobre o código que eles anotam.

Usado pelo:

Compilador para detectar erros ou suprimir *warnings*

Instalador programas de instalação podem usar a informação para gerar código, arquivos XML, etc.

Interpretador algumas anotações podem ser examinadas em tempo de execução

```
@Test
void doisMaisDoisSupostamenteSãoQuatro() { ... }

@Autor(
    nome = "Daniel Cordeiro",
    data = "02/03/2016"
)
class MinhaClasse() { ... }

@Autor(nome="Daniel Cordeiro")
@Override
void meuSuperMétodo() { ... }

@interface Autor {
    String nome() default "Desconhecido";
    String data();
```

ANOTAÇÕES NO JAVA 8

Java SE 8[1] permite anotar qualquer uso de um tipo:

- em instanciação de objetos:

```
new @Interned MeuObjeto();
```

- em conversão de tipos:

```
minhaString = (@NonNull String) str;
```

- cláusula `implements`:

```
class ListaImutável<T> implements
    @Readonly List<@Readonly T> { ... }
```

- em declarações de exceções

```
void monitorDeTemperatura() throws
    @Critical TemperaturaException { ... }
```

[1] Ver também a JSR 308

BIBLIOGRAFIA

- The Java™ Tutorials
  https://docs.oracle.com/javase/tutorial/java/concepts/index.html