

# Enumerados

# Tipos genéricos

# Java

## Tipos Enumerados

# Definição de constantes

---

- ❖ Criar uma interface para definir um conjunto de constantes é (era) prática corrente.

- Antes do JAVA 5!!

```
public static final int SPRING = 0;  
public static final int SUMMER = 1;  
public static final int FALL = 2;  
public static final int WINTER = 3;
```

- Podemos agora ter um método que aceita um destes valores

```
public setSeason(int season) { ...
```

- ❖ Este formato tem um problema!!!

- Qual?
  - Como se resolve?

# Solução

---

## ❖ Usar enum

```
public enum Season { SPRING, SUMMER, FALL, WINTER }
```

- Podemos agora ter um método que aceita um destes valores

```
public void setSeason(Season season) { ...
```

## ❖ Resolve o problema anterior. Terá outros?

# Tipos Enumerados

---

❖ Mais valia importante: “compile-time type safety”

❖ Forma mais Simples

```
public enum Color { WHITE, BLACK, RED, YELLOW, BLUE  
}
```

– Forma de referenciar  
Color.WHITE, Color.RED, etc

❖ Dentro de uma Classe

```
public class Externa{  
    public enum Color {    WHITE, BLACK, RED, YELLOW, BLUE  
    }  
}
```

– Forma de referenciar  
Externa.Color.WHITE, Externa.Color.RED, etc

# Tipos Enumerados em JAVA

---

- ❖ enum é uma classe, não um tipo primitivo.
  - São Objectos – podemos utilizar em Collections;
  - Pode implementar uma Interface.
  - Suportam comparação (== ou equals()).
- ❖ Tipos enumerados não são inteiros.
- ❖ Só têm construtores privados.
- ❖ Os valores enumerados são automaticamente public, static, final.

# Enum – uma Classe

---

- ❖ Podemos ter tipos Enumerados com dados e operações associadas:

```
public enum Color {  
    WHITE(21), BLACK(22), RED(23), YELLOW(24), BLUE(25);  
  
    // Dados  
    private int code;  
  
    // Construtor  
    private Color(int c) {    code = c; }  
  
    // Método  
    public int getCode() {    return code; }  
}
```

# Enum – implements Interface

---

❖ Os tipos enum podem implementar Interfaces

```
public enum Color implements Runnable {  
    WHITE, BLACK, RED, YELLOW, BLUE;  
  
    public void run() {  
        System.out.println("name()=" + name() +  
            ", toString()=" + toString());  
    }  
}
```

– Utilização:

```
for(Color c : Color.values()) { c.run();}
```

– Ou

```
for(Runnable r : Color.values()) { r.run();}
```



# Enum – Métodos Disponíveis

---

- ❖ São Comparable (têm uma ordem).
- ❖ Fornecem alguns métodos úteis:
  - **toString()**
  - **valueOf(String val)** : converte a String (elemento do conjunto) para um valor
  - **ordinal()**: posição (int) do valor na lista de elementos
  - **values()**: devolve a lista de elementos

# Enum - toString

---

- ❖ Por omissão, a representação tipo String é o próprio nome de cada elemento. No entanto, podemos modificar redefinindo o método toString().

```
public enum MyType {  
    ONE {  
        public String toString() {return "this is one";}  
    },  
    TWO {  
        public String toString() {return "this is two";}  
    }  
}
```

– Main:

```
public class EnumTest {  
    public static void main(String[] args){  
        System.out.println(MyType.ONE);  
        System.out.println(MyType.TWO);  
    }  
}
```

this is one  
this is two

# Enum – values()

---

- ❖ O método “values()” retorna um array com todos os elementos.

```
Name[] nameValues = Name.values();
```

- ❖ Exemplo de Utilização:

```
for (Name n : Name.values()){  
    // ... ;  
}
```

# Enum – Switch Statement

---

- ❖ A instrução switch funciona com enumerados

```
Color myColor = Color.BLACK;
```

```
switch(myColor){  
    case WHITE: ...;  
    case BLACK: ...;  
    ...  
    case BLUE: ...;  
    default: ...;  
}
```

# Enum – ordinal()

---

## ❖ Definimos um Enum para os Meses do Ano

```
public enum Mes {  
    JANEIRO, FEVEREIRO, MARCO, ABRIL, MAIO,  
    JUNHO, JULHO, AGOSTO, SETEMBRO, OUTUBRO,  
    NOVENBRO, DEZEMBRO ;  
}
```

## ❖ Se utilizarmos na classe Data

```
// Construtor de Data  
public Data (int iDia, Mes iMes, int iAno){  
    //...  
}  
// Main  
Data d1 = new Data(11, Mes.MAIO, 1900);
```

# Exemplo 1

```
enum Mes {  
    JANEIRO, FEVEREIRO, MARCO, ABRIL,  
    MAIO, JUNHO, JULHO, AGOSTO,  
    SETEMBRO, OUTUBRO, NOVEMBRO, DEZEMBRO ;  
}  
  
public class Enum1 {  
    public static void main(String[] args) {  
        for (Mes t: Mes.values())  
            System.out.println(t+" : "+t.name()+" : "+t.ordinal());  
    }  
}
```

```
JANEIRO, JANEIRO : 0  
FEVEREIRO, FEVEREIRO : 1  
MARCO, MARCO : 2  
ABRIL, ABRIL : 3  
...
```

```

enum Mes {
    JANEIRO(1), FEVEREIRO(2), MARCO(3), ABRIL(4),
    MAIO(5), JUNHO(6), JULHO(7), AGOSTO(8),
    SETEMBRO(9), OUTUBRO(10), NOVEMBRO(11), DEZEMBRO(12);

    private final int mes;
    private Mes(int m) {
        this.mes=m;
    }
    public int numMes() {
        return mes;
    }
    @Override public String toString() {
        return this.name().substring(0, 1)+
            (this.name().substring(1,this.name().length())) .toLowerCase();
    }
}

public class Enum2 {
    public static void main(String[] args) {
        for (Mes t: Mes.values())
            System.out.println(t+", "+t.name()+" : "+
                t.ordinal() + ", "+t.numMes());
    }
}

```

```

Janeiro, JANEIRO : 0, 1
Fevereiro, FEVEREIRO : 1, 2
...

```

# Exemplo 3

```
public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfMusicians;
    Ensemble(int size) {
        numberOfMusicians = size;
    }

    public int numberOfMusicians() {
        return numberOfMusicians;
    }
}
```



```

public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,    7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22,  1.137e6);

    private final double mass;    // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;    this.radius = radius;
    }
    public double mass()    { return mass; }
    public double radius() { return radius; }

    // universal gravitational constant (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity() {return G*mass/(radius*radius); }
    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}

```

# Tipos Genéricos

# Motivações

---

- ❖ Quando os programas aumentam de dimensão é possível começarmos a ter métodos que executam operações similares com diferentes tipos de dados
- ❖ O que há de “errado” com o seguinte bloco de código?

```
String add (String a, String b) { return a + b; }  
int      add (int a, int b)      { return a + b; }  
double add (double a, double b) { return a + b; }
```

```
public static void main(String[] args){  
    System.out.println(add(Gene, ricos));  
    System.out.println(add(2, 5));  
    System.out.println(add(3.2, 5.6));  
}
```

**O código está correto mas definimos estaticamente operações repetidas.**

# Motivações

## ❖ Down-Cast e Runtime Error

```
class Node {  
    Object value;  
    Node next;  
  
    Node(Object o) {  
        value = o;  
        next = null;  
    }  
}
```

```
Car c = new Car( ... );  
Node n = new Node( c );  
...  
Vehicle v = ( Vehicle ) n.value;
```

OK

```
Movie m = new UniversalMovie( "ET"  
);  
Node n = new Node( m );  
...  
Vehicle v = ( Vehicle ) n.value;
```

Run-Time Error

# O que são Genéricos?

---

- ❖ Uma forma de Polimorfismo Paramétrico
- ❖ Estruturas e Algoritmos são implementados uma única vez, mas utilizados com diferentes tipos de dados
- ❖ Dizemos que:
  - Os Tipos de dados também são um Parâmetro
- ❖ Genéricos aplicados a:
  - Métodos
  - Classes
  - Interfaces
- ❖ Introduzidos em JAVA na versão 5
  - Em C++, designam-se por templates

# Classes Genéricas

## ❖ Exemplo: Conjunto Genérico

### ❖ Declaração

```
class Conjunto<T> {  
    T[] c;  
    // ...  
}
```

O tipo parametrizado (T), não pode ser instanciado com um tipo primitivo.  
(ex: Conjunto<int> ..)

### ❖ Utilização

```
Conjunto<Pessoa> c1 = new Conjunto<Pessoa>(..);  
Conjunto<Jogador> c2 = new Conjunto<Jogador>(..);  
Conjunto<Integer> c3 = new Conjunto<Integer>(..);
```

# Classes Genéricas - Exemplo

## Sem Genéricos

```
class Stack {  
    void push(Object o) { ... }  
    Object pop() { ... }  
    ...}  
  
String s = "Hello";  
Stack st = new Stack();  
...  
st.push(s);  
...  
s = (String) st.pop();
```



E se estivermos errados?



Runtime Error

## Utilizando Genéricos

```
class Stack<T> {  
    void push(T a) { ... }  
    T pop() { ... }  
    ...}  
  
String s = "Hello";  
Stack<String> st =  
    new Stack<String>();  
  
st.push(s);  
...  
s = st.pop();
```

OK

# Genéricos

## ❖ Detecção de Erros em Compilação

```
class Node< T > {  
    T value;  
    Node< T > next;  
  
    Node( T t ) {  
        value = t;  
        next = null;  
    }  
}
```



```
Car c = new Car( ... );  
Node<Car> node = new Node<Car>( c );  
    // ...  
Car c2 = node.value;
```

OK



```
Movie m = new ActionMovie( ... );  
    // ...  
Node<Car> node = new Node<Car>( m );
```

Error



# Genéricos - Processamento

---

## ❖ Etapas de processamento de Genéricos em JAVA

- **Check**: Verificação da correcta utilização de tipos
- **Erase**: Remove toda a informação “generic type”
- **Compile**: Geração do byte-code

## ❖ Este processo denomina-se como:

- **Type Erasure**

# Genéricos - Check

---

## ❖ Declaração

```
class Foo <T> {  
    void method(T arg);  
};
```

## ❖ Utilização

```
Foo<Bar> fb = new Foo<Bar>;  
fb.method(aBar);           // OK  
fb.method(not_a_Bar);      // Compile Error
```

## ❖ O Compilador garante que o arg é do mesmo tipo <Bar>

# Genéricos - Erase

---

- ❖ Cada parâmetro definido como genérico é substituído por um `java.lang.Object`
- ❖ Os casts “Object -> Tipo Concreto” são automaticamente introduzidos pelo compilador.

```
class choice <T>  
{ public T best ( T a , T b ) {...} }
```

– É substituído por:

```
class choice  
{ public Object best ( Object a, Object b ) {...} }
```

# Genéricos em Classes

```
public class Stack_Generic<T> {
    private class Node<E> {
        E val;
        Node<E> next;
        Node(E v, Node<E> n) {
            val = v;
            next = n;
        }
    }
    private Node<T> top = null;
    public boolean empty( ) {
        return top == null;
    }

    public T pop( ) {
        T result = top.val;
        top = top.next;
        return result;
    }

    public void push(T v) {
        top = new Node<T>(v, top);
    }
}
```

```
public class TestStack {

    public static Figura randFig() {
        switch ((int)(Math.random()*(3))) {
            case 0:
                return new Circulo (1,3, 1.2);
            case 1:
                return new Quadrado(3,4, 2);
            case 2:
                return new Rectangulo(1,1, 5,6);
        }
    }

    public static void main(String[] args) {
        Stack_Generic<Figura> stk =
            new Stack_Generic<Figura>();

        for (int i=0; i<10; i++)
            stk.push(randFig());

        for (int i=0; i<10; i++)
            System.out.println(stk.pop());
    }
}
```

# Genéricos em Métodos

```
public <T> T doSomething(T a, T b) { ... }

public static <T> void fromArrayToCollection(T[] a, Collection<T> c) {
    for (T o : a)
        c.add(o);
}

public static <T> int countOccurrences(T[] list, T itemToCount) {
    int count = 0;
    if (itemToCount == null) {
        for (T listItem : list)
            if (listItem == null)
                count++;
    } else {
        for (T listItem : list)
            if (itemToCount.equals(listItem))
                count++;
    } return count;
}
```

# Genéricos e Subtipos

- ❖ Como se conjuga Polimorfismo com Tipos Genéricos?

```
public static void main(String[] args) {  
    LinkedList<Figura> list = new LinkedList<Figura>();  
    LinkedList<Quadrado> list2 = new LinkedList<Quadrado>();  
    Quadrado q = new Quadrado(3,4, 2);
```

```
list.add(q);  
list2.add(q);
```

OK


```
LinkedList<Quadrado> list3 = list;  
LinkedList<Figura> list4 = list2;
```

Compile-Time  
Error

Porquê ?

# Genéricos e Subtipos

```
LinkedList<Quadrado> list = new LinkedList<Quadrado>();  
LinkedList<Figura> list2 = list; // Imaginando que é possível  
  
Figura f = new Figura(..); // Supondo que não é abstract  
  
list2.add(f);  
  
Quadrado q = list.get(0); // Runtime ERROR!!!!
```



**Uma Figura não é um Quadrado**

Se X um subtipo de Y e G um tipo genérico,  
não é verdade que  $G<X>$  é um subtipo de  $G<Y>$

# Genéricos e Subtipos

```
public static void main( String[ ] args ) {  
    LinkedList<Quadrado> list = new LinkedList<Quadrado>();  
    Quadrado q = new Quadrado(3,4, 2);  
    list.add(q);  
    list.add(q);  
  
    print(list);  
}
```

**Compile-Time Error**

```
public static void print( LinkedList<Figura> listOfFig ) {  
    Iterator it = listOfFig.iterator();  
    while( it.hasNext())  
        System.out.println( it.next() );  
}
```

**Questão: Como permitir que, tendo um argumento tipo LinkedList de Figura, se possa aceitar uma LinkedList de Figura mas também um dos seus subtipos?**



# Genéricos - Wildcards

---

## ❖ Bounded wildcards

`< ? extends class-T >`

subclass de class-T, incluindo class-T

`< ? extends class-T & interface-E >`

subclass de class-T e int-E, incl. class-T e int-E

`< ? super class-T >`

superclass de class-T, incluindo class-T

## ❖ Unbounded wildcards

`< ? extends Object >`

subclass de Object, i.e. qualquer tipo

`< ? >`

Semelhante a `< ? extends Object >`

# Genéricos e Subtipos

```
public static void main( String[ ] args ) {  
    LinkedList<Quadrado> list = new LinkedList<Quadrado>();  
    Quadrado q = new Quadrado(3,4, 2);  
    list.add(q);  
    list.add(q);  
  
    print(list);  
}  
  
public static void print( LinkedList<? extends Figura> listOfFig )  
{  
    Iterator it = listOfFig.iterator();  
    while( it.hasNext())  
        System.out.println( it.next() );  
}
```

# Genéricos - Wildcards

---

- ❖ É possível ainda especificar um parâmetro tipo genérico que *extend/implement* uma *classe/interface*
- ❖ Para ambos os casos é utilizada a keyword *extends*

```
public class Desk <T extends Serializable & Comparable<T>>  
{..}
```

- Neste caso o tipo genérico *T* deve implementar *Serializable* e *Comparable*

# Genéricos - Vetores

---

- ❖ Não é possível criar um vetor de tipos genéricos

```
T[] array = new T[MAX];
```

**Compile-Time Error**

– Solução:

```
@SuppressWarnings("unchecked")  
T[] newArray = (T[]) new Object[MAX];
```

# Java 7 - diamond <>

---

- ❖ A partir da versão 7 do Java podemos usar a nomenclatura "<>"
  - informalmente designado por diamante (*diamond*).

- ❖ Exemplo

- em vez de:

```
Box<Integer> integerBox = new Box<Integer>();  
Map<String, List<String>> myMap =  
    new HashMap<String, List<String>>();
```

- podemos escrever:

```
Box<Integer> integerBox = new Box<>();  
Map<String, List<String>> myMap = new HashMap<>();
```

# Naming Conventions

---

- ❖ Por convenção os nomes dos tipos paramétricos são letras maiúsculas.
  - Isto é contra a convenção de nomes de variáveis em Java (i.e. `numberOfElements`), mas por uma boa razão
  - Sem esta convenção seria difícil distinguir entre uma variável de tipo paramétrico e uma variável normal.
- ❖ Nomes comuns:
  - E - Element
  - K - Key
  - N - Number
  - T - Type
  - V - Value

# Exemplos

```
public class Pair<K, V> {  
    private K key;  
    private V value;  
  
    public Pair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
  
    public K getKey()    { return key; }  
    public V getValue() { return value; }  
}  
  
...  
Pair<String, Integer> p1 = new Pair<>("Even", 8);  
Pair<String, String>  p2 = new Pair<>("hello", "world");
```

# Exemplos

```
public class Box<T> {  
    private T t;  
    public void add(T t) {  
        this.t = t;  
    }  
    public T get() {  
        return t;  
    }  
    public <U> void inspect(U u){  
        System.out.println("T: " + t.getClass().getName());  
        System.out.println("U: " + u.getClass().getName());  
    }  
    public static void main(String[] args) {  
        Box<Integer> box1 = new Box<>();  
        box1.add(new Integer(10));  
        box1.inspect("some text");  
    }  
}
```

T: java.lang.Integer  
U: java.lang.String



# Genéricos – Sumário

---

## ❖ Conseguimos:

- eliminar necessidade de coerção explícita (cast)
- aumentar robustez: verificação estática de tipo
- aumentar legibilidade

## ❖ Não há múltiplas versões do código

- declaração é compilada para todos os tipos
- parâmetros formais possuem tipo genérico
- na invocação, os tipos dos parâmetros actuais são substituídos pelos tipos dos formais