Programação Orientada a Objetos
Java Generics

Rodrigo Bonifácio

15 de abril de 2013

Implementação de uma Pilha (simples)

```
package br.unb.cic.adt;

public class PilhaComoArray implements PilhaAbstrata {
  private static final int MAX_SIZE = 10;

  private int pilha[];
  private int topo;

  public PilhaComoArray() { ... }
  public void empilha(int valor) { ... }
  public int remove() { ... }
  public boolean pilhaCheia() { ... }
  public boolean pilhaVazia() { ... }
}
```

Implementação de uma Pilha (simples)

```
package br.unb.cic.adt;

public class PilhaComoArray implements PilhaAbstrata {
  private static final int MAX_SIZE = 10;

  private int pilha[];
  private int topo;

  public PilhaComoArray() { ... }
  public void empilha(int valor) { ... }
  public int remove() { ... }
  public boolean pilhaCheia() { ... }
  public boolean pilhaVazia() { ... }
}
```

- Pouco reusável (tamanho e tipo de elementos fixos)
- Sem a definição de contratos para lidar com situações de erros

...estabelecendo contratos de exceções

```
package br.unb.cic.adt;

public interface PilhaAbstrata {
  public void empilha(int valor) throws ExcecaoPilha;
  public int remove() throws ExcecaoPilha;
  public boolean pilhaCheia();
  public boolean pilhaVazia();
}
```

...estabelecendo contratos de exceções

```
package br.unb.cic.adt;

public interface PilhaAbstrata {
  public void empilha(int valor) throws ExcecaoPilha;
  public int remove() throws ExcecaoPilha;
  public boolean pilhaCheia();
  public boolean pilhaVazia();
}
```

- Ainda com a limitação de pouco reuso

Como tornar a pilha mais reusável, em relação ao tipo dos elementos?

Como tornar a pilha mais reusável, em relação ao tipo dos elementos?

Queremos diferentes tipos de pilhas

- Pilhas de figuras geométricas
- Pilhas contendo ambientes de execução de um programa
- Pilhas de qualquer coisa que tenhamos interesse

Nas linguagens imperativas, três soluções bem difundidas

Linguagem C:

Copiar e colar o código fonte da pilha e fazer as alterações manualmente.

Nas linguagens imperativas, três soluções bem difundidas

Linguagem C:

Copiar e colar o código fonte da pilha e fazer as alterações manualmente.
Na verdade, a pilha poderia conter um apontador para `void *`, mas isso requer conversões de tipo espalhadas pelo programa.

Nas linguagens imperativas, três soluções bem difundidas

Linguagem C:

Copiar e colar o código fonte da pilha e fazer as alterações manualmente. Na verdade, a pilha poderia conter um apontador para `void *`, mas isso requer conversões de tipo espalhadas pelo programa.

Linguagens Smalltalk e (Java $<$ 1.5):

Uso de herança, fazendo com que containers como pilhas mantenham referências para uma classe ancestral na hierarquia (possivelmente, `Object`).

Nas linguagens imperativas, três soluções bem difundidas

Linguagem C:

Copiar e colar o código fonte da pilha e fazer as alterações manualmente. Na verdade, a pilha poderia conter um apontador para `void *`, mas isso requer conversões de tipo espalhadas pelo programa.

Linguagens Smalltalk e (Java $<$ 1.5):

Uso de herança, fazendo com que containers como pilhas mantenham referências para uma classe ancestral na hierarquia (possivelmente, `Object`). Novamente, isso requer uma séria de conversões de tipos espalhadas pelo programa.

Nas linguagens imperativas, três soluções bem difundidas

Linguagem C:

Copiar e colar o código fonte da pilha e fazer as alterações manualmente. Na verdade, a pilha poderia conter um apontador para `void *`, mas isso requer conversões de tipo espalhadas pelo programa.

Linguagens Smalltalk e (Java $<$ 1.5):

Uso de herança, fazendo com que containers como pilhas mantenham referências para uma classe ancestral na hierarquia (possivelmente, `Object`). Novamente, isso requer uma séria de conversões de tipos espalhadas pelo programa. Além disso, como C++ suporta herança múltipla, não pode existir a restrição de que todas as classes possuem um ancestral comum.

Java Generics

- Containers (como pilhas) são declarados em termos de tipos parametrizados, que são substituidos pelo compilador quando referenciamos uma instância de um template:

Java Generics

- Containers (como pilhas) são declarados em termos de tipos parametrizados, que são substituidos pelo compilador quando referenciamos uma instância de um template:
  - pilha de inteiros
  - pilha de figuras geométricas
  - pilha de ambientes de execução de um programa, ...

Java Generics

- Containers (como pilhas) são declarados em termos de tipos parametrizados, que são substituidos pelo compilador quando referenciamos uma instância de um template:
  - pilha de inteiros
  - pilha de figuras geométricas
  - pilha de ambientes de execução de um programa, ...
- Em vez de reusar propriedades e métodos de objetos, o mecanismo de *Generics* em Java reusa código fonte
- Mecanismo de *type erasure* implementado pelo compilador.

Java Generics

- Containers (como pilhas) são declarados em termos de tipos parametrizados, que são substituidos pelo compilador quando referenciamos uma instância de um template:
  - pilha de inteiros
  - pilha de figuras geométricas
  - pilha de ambientes de execução de um programa, ...
- Em vez de reusar propriedades e métodos de objetos, o mecanismo de *Generics* em Java reusa código fonte
- Mecanismo de *type erasure* implementado pelo compilador. Não dá para entrar em detalhes sobre como esse mecanismo é implementado na aula de hoje.

Sintaxe para definição de generics

Diferentemente de C++, não se faz necessária o uso de uma palavra reservada para declarar um tipo como sendo parametrizado. Apenas indicamos que o tipo é parametrizado em relação a um ou mais tipos.

Sintaxe para definição de generics

Diferentemente de C++, não se faz necessária o uso de uma palavra reservada para declarar um tipo como sendo parametrizado. Apenas indicamos que o tipo é parametrizado em relação a um ou mais tipos.

```
package br.unb.cic.adt;

public interface PilhaGenerica<T> {
  public void empilha(T valor);
  public T remove();
  public boolean pilhaCheia();
  public boolean pilhaVazia();
}
```

Implementação (1/2)

```
package br.unb.cic.adt;

import java.util.ArrayList;
import java.util.List;

public class PilhaGenericaComoArray<T> implements
  PilhaGenerica<T> {
  private static final int MAX_SIZE = 10;
  private int topo;
  private List<T> pilha;

  public PilhaGenericaComoArray() {
    topo = 0;
    pilha = new ArrayList<T>(MAX_SIZE);
  }
  public void empilha(T valor) {
    pilha.add(topo++, valor);
  }
  ...
```

Implementação (2/2)

```
    public T remove() {
      return pilha.get(--topo);
    }
    public boolean pilhaCheia() {
      return topo >= MAX_SIZE;
    }
    public boolean pilhaVazia() {
      return topo == 0;
    }
}
```

Instanciação de um template

```
public static void main(String args) {
  PilhaGenericaComoArray<int> stack;

  for(int i = 0; i < 5; i++) {
    stack.empilha(fibonacci(i));
  }

  for(int k = 0; k < 5; k++) {
    System.out.println(stack.remove());
  }
}
```

Constantes em templates C++

A parametrização de uma classe não é restrita aos tipos que ela manipula. Valores também podem ser parametrizados em relação a uma definição de classe (isso é válido apenas em C++).

Constantes em templates C++

A parametrização de uma classe não é restrita aos tipos que ela manipula. Valores também podem ser parametrizados em relação a uma definição de classe (isso é válido apenas em C++).

```
template<class T, int msize = 100>
class Stack {
 private:
  int top;
  T stack[msize];
 public:
  Stack();
  T pop() throw (StackException);
  void push(T value) throw (StackException);
  bool isFull() throw ();
  bool isEmpty() throw();
};
```

Neste ponto, precisaríamos revisitar a implementação do método `bool isFull()`

```
template <class T>
bool Stack<T>::isFull() throw() {
  return this->top >= msize;
}
```

Como tornar a pilha ainda mais flexível, sem limite de tamanho?

Como tornar a pilha ainda mais flexível, sem limite de tamanho?

- Simples, usando alocação dinâmica

Como tornar a pilha ainda mais flexível, sem limite de tamanho?

- Simples, usando alocação dinâmica não declaramos um array contendo `msize` ou `MAX_SIZE` elementos.

Como tornar a pilha ainda mais flexível, sem limite de tamanho?

- Simples, usando alocação dinâmica não declaramos um array contendo `msize` ou `MAX_SIZE` elementos.

Tarefa para vocês . . .

Templates C++ podem ser usados para abstrair não apenas os tipos referenciados por classes, mas também os tipos usados para estabelecer as assinaturas das funções.

Pense na função int sqr(int)

```
int sqr(int x) {
  return x * x;
}
```

Pense na função `int sqr(int)`

```
int sqr(int x) {
  return x * x;
}
```

- A princípio, ela poderia ser usada para calcular o quadrado de números inteiros ou ponto flutuante.

Pense na função `int sqr(int)`

```
int sqr(int x) {
  return x * x;
}
```

- A princípio, ela poderia ser usada para calcular o quadrado de números inteiros ou ponto flutuante. Podemos usar templates para reusar essa implementação.

Pense na função int sqr(int)

```
int sqr(int x) {
  return x * x;
}
```

- A princípio, ela poderia ser usada para calcular o quadrado de números inteiros ou ponto flutuante. Podemos usar templates para reusar essa implementação.

```
template <class T>
T sqr(T x) {
  return x * x;
}
```

The Standard Template Library

The C++ STL[15] is a powerful library intended to satisfy the vast bulk of your needs for containers and algorithms, but in a completely portable fashion. This means that not only are your programs easier to port to other platforms, but that your knowledge itself does not depend on the libraries provided by a particular compiler vendor (and the STL is likely to be more tested and scrutinized than a particular vendor's library). Thus, it will benefit you greatly to look first to the STL for containers and algorithms, *before* looking at vendor-specific solutions.

A STL e a API Collections disponibiliza containers e algoritmos

Containers

- vector, deque, list,
- stack, queue, priority_queue
- map, set, bitset, ...

Algoritmos

- recuperar os valores máximo / mínimo
- ordenar os valores de uma lista
- pesquisar e substituir os elementos de um container
- iterators para navegar nos elementos de um container
- map, fold, ...

Antes de desenvolver um novo container ou algoritmo que opere sobre containers, examine a Standard Template Library

Antes de desenvolver um novo container ou algoritmo que opere sobre containers, examine a Standard Template Library ou as classes em `java.util`.

Programação Orientada a Objetos
Java Generics

Rodrigo Bonifácio

15 de abril de 2013

Discussões adicionais . . .

C++ Templates
x
Java Generics

Principais diferenças

Principais diferenças

- Relacionadas a sistemas de tipos (Java tem uma implementação mais robusta, apesar de ser menos interessante que a implementação de C♯)
- Relacionadas as capacidades de programação generativa, Java é bastante limitado

Principais diferenças

- Relacionadas a sistemas de tipos (Java tem uma implementação mais robusta, apesar de ser menos interessante que a implementação de C♯)
- Relacionadas as capacidades de programação generativa, Java é bastante limitado, fazendo com que os hackers de C++ percebem limitações claras na linguagem java.

```
template<class T>
class Stack {
 private:
  int top;
  T stack[MAX_SIZE];
 public:
  Stack();
  ...
  void drawAll();
};
...
template <class T>
void Stack<T>::drawAll() {
  while(!isEmpty()) {
    pop().draw();
  }
}
```

```
int main() {
  Stack<int> intstack;

  for(int i = 0; i < 10; i++) {
    intstack.push(i);
  }
 //intstack.drawAll();
}
```

```
template<class T>
class Stack {
 private:
  int top;
  T stack[MAX_SIZE];
 public:
  Stack();
  ...
  void drawAll();
};
...
template <class T>
void Stack<T>::drawAll() {
  while(!isEmpty()) {
    pop().draw();
  }
}
```

```
int main() {
  Stack<int> intstack;

  for(int i = 0; i < 10; i++) {
    intstack.push(i);
  }
 //intstack.drawAll();
}
```

- Esse código compila em C++

Adding Generics to the Java Programming Language: Participant Draft Specification

Gilad Bracha, Sun Microsystems
Norman Cohen, IBM
Christian Kemper, Inprise
Steve Marx, WebGain
Martin Odersky, EPFL
Sven-Eric Panitz, Software AG
David Stoutamire, Sun Microsystems
Kresten Thorup, Aarhus University
Philip Wadler, Lucent Technologies

April 27, 2001

Generative Programming
Mastering C++ templates

Generative Programming
Mastering C++ templates