Implementação de Generics no .NET CRL

Análise do artigo *Design and Implementation of Generics for the .NET Common Language Runtime*

Discentes: Felipe R. Mizher, Matheus Felipe
Docente: Marco Rodrigo Costa

PUC Minas

Sumário

01

O Problema

Antes da introdução dos Generics no C#, a linguagem apresentava limitações significativas relacionadas à tipagem e ao desempenho no uso de coleções.

- **A "Cegueira" do Compilador:**
  - "No C# 1.0, coleções como ArrayList operavam exclusivamente com System.Object. O compilador perdia a capacidade de verificar a compatibilidade de tipos, permitindo inserções indevidas (ex: misturar strings em listas de números)."
- **Fragilidade em Runtime:**
  - "A ausência de verificação estática transferia a responsabilidade para o tempo de execução. Isso resultava em exceções do tipo InvalidCastException apenas quando o usuário final utilizava o software, dificultando a depuração."
- **Overhead de Código (Casts):**
  - "O código tornava-se verboso e sujo, exigindo Casting (conversão explícita) em toda operação de leitura ((int)lista[0]), violando boas práticas de legibilidade."

```
// C# 1.0 (O jeito antigo)
ArrayList numeros = new ArrayList();
numeros.Add(10); // Ocorre Boxing (int vira Object)
numeros.Add("Erro"); // Compila, mas quebra depois!

int n = (int)numeros[0]; // Necessário Cast explícito
```

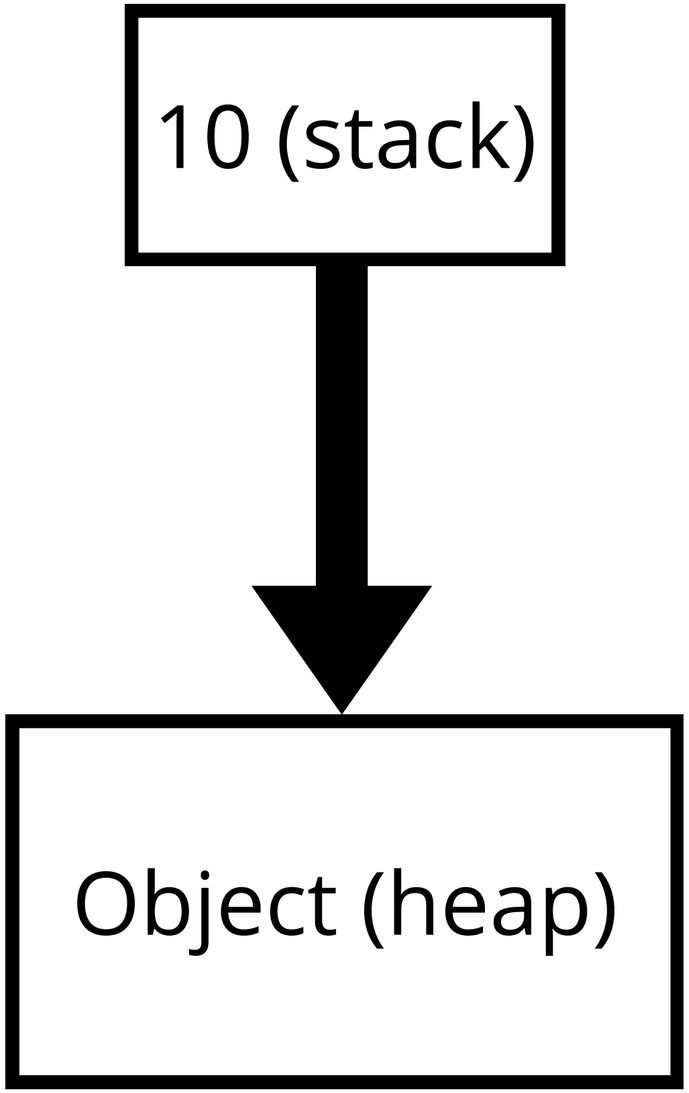
02 O Gargalo de Performance

O Custo da Alocação na Heap:

- "Tipos de valor (int, structs) são otimizados para viver na Stack. Ao inseri-los em um ArrayList, o CLR é forçado a alocar memória na Managed Heap para criar um objeto container."

Pressão no Garbage Collector (GC):

- "O Boxing cria milhares de objetos temporários na memória. Isso aumenta a frequência e a duração dos ciclos de coleta do Garbage Collector, causando pausas perceptíveis na execução de aplicações de alta performance."



03 Solução

Segurança Estática de Tipos (Static Type Safety):
"A introdução do parâmetro <T> transfere a verificação de erros do Runtime para o Compile-time. O compilador agora valida estaticamente se os dados inseridos correspondem ao contrato da lista, impedindo falhas em produção."

Polimorfismo Paramétrico:
"O C# adotou o conceito de polimorfismo paramétrico, permitindo escrever algoritmos (como ordenação ou busca) de forma genérica uma única vez, sendo aplicáveis a qualquer tipo de dado sem reescrita."

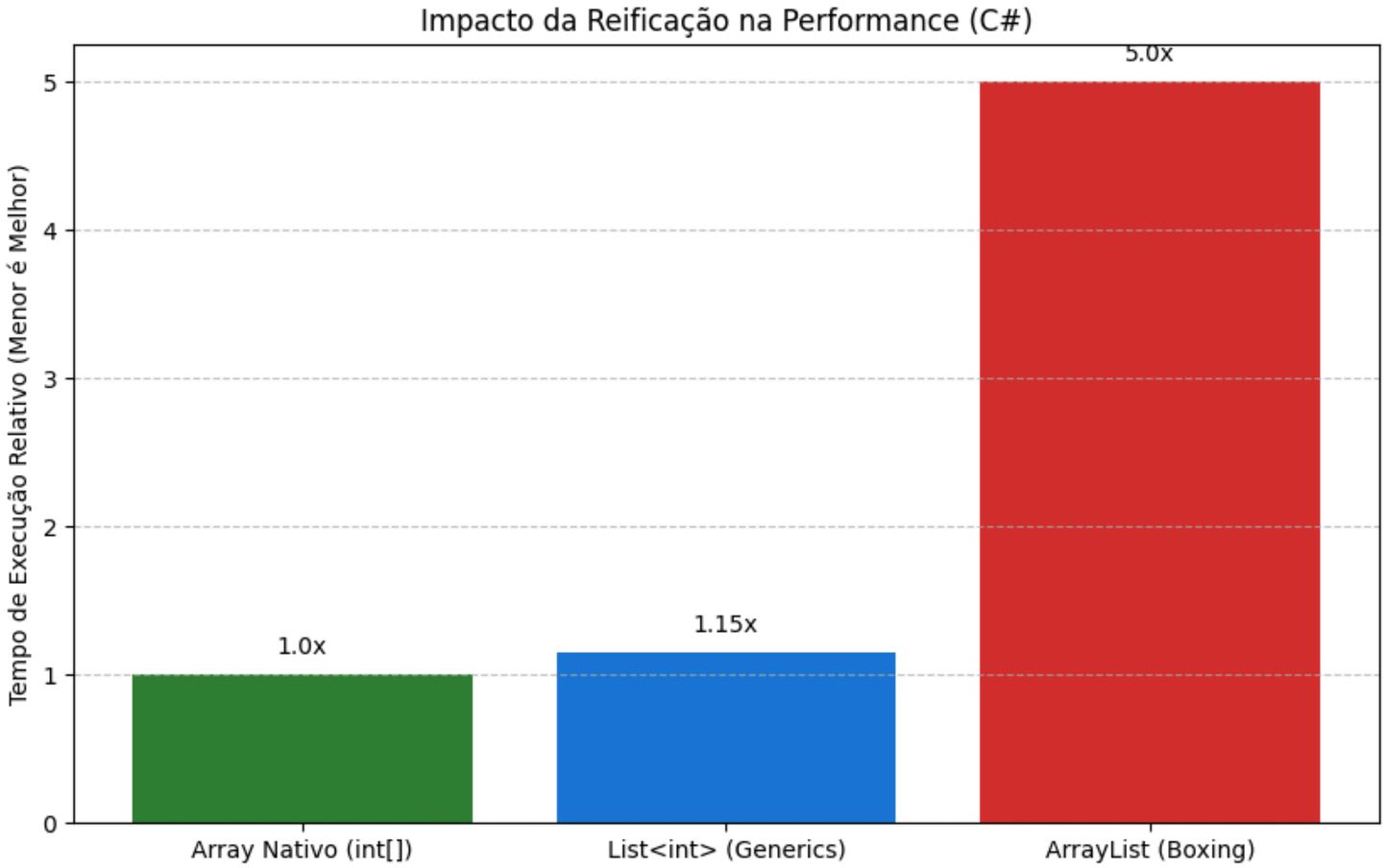
Expressividade e Legibilidade:
"Eliminação da 'poluição visual' causada pelos Casts explícitos (int). O código passa a expressar a intenção do programador de forma clara e direta: uma List<int> é inequivocamente uma lista de inteiros."

```
// Declaração explícita do tipo <int>
List<int> lista = new List<int>();

// 1. Segurança: O compilador bloqueia tipos errados antes de rodar
// lista.Add("Texto"); // Error CS1503: Argument cannot convert string to int

lista.Add(10); // 2. Eficiência: Armazena o int puro, sem conversão

// 3. Legibilidade: Acesso direto, sem necessidade de Cast
int valor = lista[0];
```

04 Implementação no CLR: Reificação vs Type Erasure

- **Diferença Arquitetural (Java vs C#):**
  - "O artigo destaca a divergência de design: O Java optou por Type Erasure (apagamento de tipos) para manter compatibilidade reversa, tratando genéricos como Object por baixo dos panos."
- **Especialização via JIT (Just-In-Time):**
  - "O .NET adotou a Reificação. Quando o JIT encontra List<int>, ele gera dinamicamente uma classe especializada com instruções de máquina nativas para inteiros."
- **Polimorfismo sem Ponteiros:**
  - "Isso permite que estruturas de dados genéricas manipulem tipos de valor diretamente, sem a necessidade de indirection (ponteiros) ou boxing, alcançando performance similar a C++."

05 Impacto na Performance



Conclusão do Gráfico:

- List<int> é drasticamente mais rápido que ArrayList.
- A performance é quase idêntica a de um Array puro.

06 Destaque - Eficiência de memoria

Matheus

- **O Problema do Boxing:**
  - Armazenar tipos simples (como números) em coleções antigas exige criar "caixas" (Objetos).
  - Isso gera um overhead (peso extra) desnecessário na memória RAM.
- **Comparativo de Consumo:**
  - Com Generics (List<int>): Ocupa apenas 4 bytes por item.
  - Sem Generics (ArrayList): Ocupa aprox. 16 bytes por item (Dado + Cabeçalhos do Objeto).
- **Impacto na Escala:**
  - O uso de Generics reduz o consumo de memória em até 4 vezes para tipos de valor.
  - Menos memória ocupada = Menos trabalho para o Garbage Collector limpar.

06 Destaque - Segurança de Tipos

Felipe

- **O Problema da Falta de Segurança de Tipos:**
  - Coleções antigas armazenavam dados como *Object.*
  - O compilador não verificava os tipos em tempo de compilação.
  - Erros de tipo só eram identificados em tempo de execução.
- **Riscos Associados:**
  - Necessidade de casts explícitos.
  - Possibilidade de falhas como *InvalidCastException.*
  - Código menos confiável e mais difícil de manter.
- **Impacto na Confiabilidade:**
  - Redução significativa de erros em runtime.
  - Código mais seguro, legível e fácil de manter.
  - Melhor para aplicações de grande escala.

- **Generics tornam erros de tipo impossíveis antes da execução do programa.**

07 Conclusão

- **Síntese do Artigo:**
  - "O artigo de Kennedy e Syme demonstra que é possível implementar Generics em linguagens gerenciadas sem sacrificar a eficiência."
- **Resultados Obtidos:**
  - "A Reificação provou ser superior ao Type Erasure para operações numéricas e científicas, eliminando o overhead de memória."
- **Impacto no Mercado:**
  - "Essa implementação permitiu que o C# competisse em cenários de alta performance (como Game Development com Unity e Sistemas Financeiros), onde o custo do Boxing seria inaceitável."

08 Referência Bibliográfica

- KENNEDY, Andrew; SYME, Don. Design and Implementation of Generics for the .NET Common Language Runtime. Microsoft Research, Cambridge, UK, 2001.

OBRIGADO