# Iterator

## Objetivo

- Prover uma forma de seqüencialmente acessar os elementos de uma coleção sem expor sua representação interna

## Também conhecido como

- Cursor

## Motivação

- Queremos isolar o uso de uma estrutura de dados de sua representação interna de forma a poder mudar a estrutura sem afetar quem a usa
- Para determinadas estruturas, pode haver formas diferentes de caminhamento ("traversal") e queremos encapsular a forma exata de caminhamento
  - Exemplo: árvore pode ser varrida "em ordem", em "pós-ordem", em "pré-ordem"
  - Exemplo: podemos ter um "iterador com filtro" que só retorna certos elementos da coleção
    - Exemplo: num editor de documentos, poderíamos ter os elementos do documento organizados em árvores (documento consiste de páginas que consistem de parágrafos, ...) e ter um iterador especial para elementos não gráficos (que podem ter sua grafia verificada, por exemplo)
- A idéia do iterador é de retirar da coleção a responsabilidade de acessar e caminhar na estrutura e colocar a responsabilidade num novo objeto separado chamado um iterador
- A interface Iterador:
  - Define uma interface para acessar os elementos da coleção
- A classe Iterador

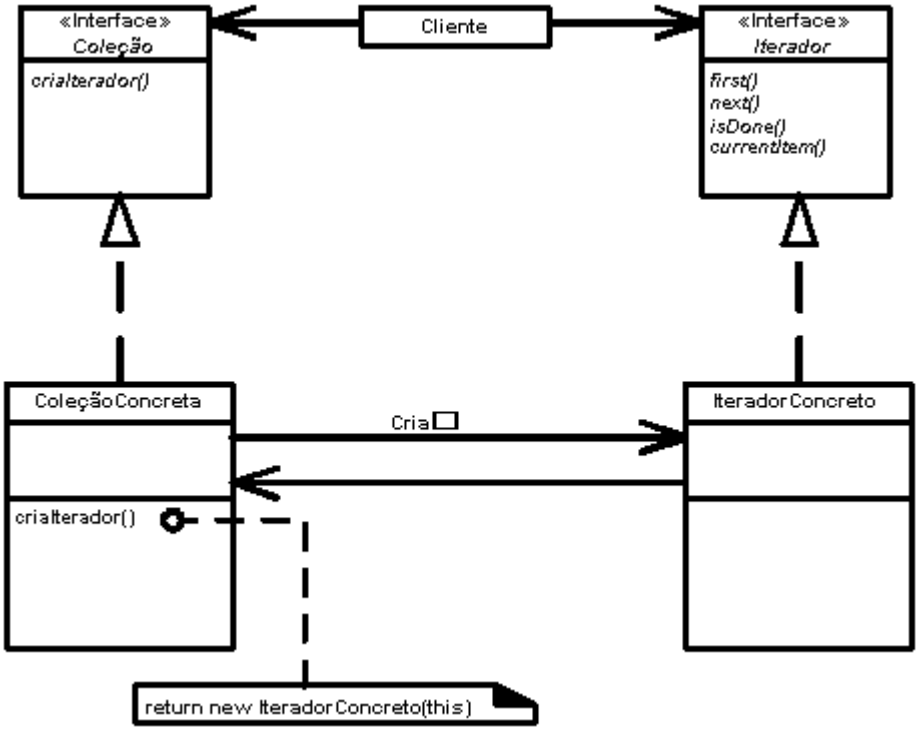
- Implementa a interface Iterador
- Mantém qualquer informação de estado necessária para saber até onde a iteração (caminhamento) já foi (mantém o cursor)
  - Isso não é mantido na própria classe para que threads diferentes possam caminhar em paralelo na coleção

- Como criar um iterador?
  - Não podemos usar new de uma classe concreta diretamente pois o iterador a ser criado depende da coleção a ser varrida
  - Solução: a coleção tem um factory method para criar um iterador
    - Exemplo em java: Vector.iterator()

## Aplicabilidade

- Use o padrão Iterator:
  - Para acessar o conteúdo de uma coleção sem expor suas representação interna
  - Para suportar múltiplas formas de caminhamento
  - Para prover uma interface única para varrer coleções diferentes
    - Isto é, para suportar a iteração polimórfica

## Estrutura




## Exemplo de uso: um impressor genérico em Java

```
class Printer {
    static void printAll(Iterator it) {
        while(it.hasNext()) {
            System.out.println(it.next().toString());
        }
    }
}

class LabirintoDeRatos {
    public static void main(String[] args) {
        Collection ratos = new Vector();
        for(int i = 0; i < 3; i++) {
            ratos.add(new Rato(i));
        }
        // iterador criado aqui
        Printer.printAll(ratos.iterator());
    }
}
```

## Participantes

- **Iterador** (Enumeration no Java original)

- Define a interface para acessar e caminhar nos elementos

- **IteradorConcreto** (o objeto retornado pelo método iterator() em Java)
  - Implementa a interface Iterator
  - Mantém estado para saber o elemento corrente na coleção
- **Coleção** (Collection em Java)
  - Define uma interface para criar um objeto iterador
- **ColeçãoConcreta** (ArrayList, LinkedList, Vector, ... em Java)
  - Implementa a interface de criação do iterador e retorna uma instância do IteradorConcreto

## Conseqüências

- A mera substituição de um iterador permite caminhar numa coleção de várias formas
- Juntar a interface de caminhamento num iterador permite retirar esta interface da coleção, simplificando assim a interface desta coleção
- Várias iterações podem estar ocorrendo ao mesmo tempo, já que o estado de uma iteração é mantido no iterador e não na coleção

## Detalhes de implementação

- Iteradores internos versus iteradores externos
  - Com iterador interno, o cliente passa uma operação a ser desempenhada pelo iterador e este o aplica a cada elemento
  - Com iterador externo (mais flexíveis), o cliente usa a interface do iterador para caminhar mas ele mesmo (o cliente) processa os elementos da coleção
  - Um iterador interno só é usado quando um iterador externo seria difícil de implementar
    - Exemplo: para coleções complexas, manter o

estado da iteração pode ser difícil (teria que armazenar o caminho inteiro dentro de uma coleção recursiva multi-nível). Neste caso, usar um iterador interno recursivo e armazenar o estado na própria pilha de execução pode ser mais simples

- Tratamento de concorrência
  - O que ocorre se houver mudanças à coleção (novos objetos ou objetos removidos) durante uma iteração (devido a um thread)?
  - Um "iterador fail-fast" indica imediatamente que está havendo acesso concorrente (via exceção)
  - Alguns iteradores podem clonar a coleção para caminhar, mas isto é caro em geral
  - Um "iterador robusto" permite fazer iterações e mudar a coleção sem se perder
    - Porém, devido às regiões críticas, fica mais lento
- Operadores do iterador
  - Pode permitir ou não andar para trás, pular posições, etc.
- Iteradores nulos são interessantes para prover condições limites
  - Um iterador nulo sempre diz que a iteração acabou
  - Exemplo: se todo objeto de um Composite (ver à frente) tiver um iterador, uma folha poderia ter um iterador nulo

## Exemplo de código

- O iterador do Vector em Java seria semelhante ao que segue abaixo

```
public Iterator iterator() {
    return new Iterator() { // classe interna anônima
        int cursor = 0;

        public boolean hasNext() {
            return cursor < size();
        }
```

```
        public Object next() {
            try {
                Object next = get(cursor);
                cursor++;
                return next;
            } catch(IndexOutOfBoundsException e) {
                throw new NoSuchElementException();
            }
        }
    };
}
```

## Pergunta final para discussão

- Considere uma coleção (objeto composto) que contenha objetos representando empréstimos. A interface do objeto Empréstimo contém um método chamado ValorDoEmpréstimo() que retorna o valor corrente do empréstimo. Dado um requisito para extrair todos os empréstimos da coleção com valor menor que um limite (ou maior que um limite ou entre dois limites), você usaria ou escreveria um iterador para resolver o problema?

programa