Aula Prática 10

Resumo:
- Pilhas (*stacks*) e filas (*queues*).

Exercício 10.1

Faça um programa que detecte se uma sequência de letras e dígitos é um palíndromo (ou seja, se a sequência lida do início para o fim é igual à sequência lida do fim para o início). O programa deve ignorar todos os caracteres que não sejam letras ou dígitos, assim como ignorar a diferença entre maiúsculas e minúsculas. Por exemplo as frases: `"somos"` e `"O galo nada no lago"` são palíndromos. Utilize uma *pilha* e uma *fila* para resolver este problema.

Exercício 10.2

O programa `SolveHanoi` permite visualizar a resolução do problema das Torres de Hanói, passo a passo. Crie a classe `HanoiTowers` que simula o problema. A classe deve incluir três campos do tipo pilha, um para cada torre. O construtor deve iniciar a primeira torre com n discos e manter as outras duas vazias. O método `solve` deve aplicar um algoritmo idêntico ao discutido na aula de recursividade para resolver o problema.

A classe pode (e deve) definir métodos auxiliares, em particular:

- Um método `moveDisk(a, b)` que simule a operação fundamental de mover um disco de uma torre para outra, modificando as pilhas correspondentes.
- Um método, a ser chamado após cada movimento, que mostre o estado atual do problema, incluindo os conteúdos das três torres. O aspeto pretendido poderá ser como nos exemplos abaixo:

```
After 0 moves:          After 1 moves:     ...      After 31 moves:
A: [5, 4, 3, 2, 1]      A: [5, 4, 3, 2]             A: []
B: []                   B: []                       B: []
C: []                   C: [1]                      C: [5, 4, 3, 2, 1]
```

- Deve acrescentar à classe `Stack` um método `reverseToString` que devolva, sem modificar, o conteúdo da pilha numa string, indo da base para o topo.

Exercício 10.3

As técnicas de processamento digital de sinais usam frequentemente uma estrutura de armazenamento chamada *linha de atraso* (*digital delay line*), que é essencialmente uma fila de tamanho fixo que permite acesso direto (de leitura) a qualquer dos elementos. Quando a linha de atraso é criada, todos os elementos são iniciados com um valor pré-definido (geralmente zero). Quando se introduz um novo elemento, é colocado no fim da fila e, simultaneamente, o elemento mais antigo é descartado automaticamente.

O programa `TestDelayLine.java` usa uma linha de atraso para mostrar a temperatura média das últimas 24 horas, hora-a-hora ao longo de vários dias.

a. Complete a classe `DelayLine` com os métodos necessários para que o programa funcione corretamente.

b. Uma vez que a linha de atraso requer acesso aleatório e o tamanho é fixo, é vantajoso implementá-la com a técnica de array circular. Crie uma classe `DelayArray` com a mesma interface, mas usando essa implementação. Teste a nova classe num programa `TestDelayArray`.

Exercício 10.4

Construa uma calculadora com as quatro operações aritméticas básicas que funcione com a notação pós-fixa (*Reverse Polish Notation*). Nesta notação os operandos são colocados antes do operador. Assim `2 + 3` passa a ser expresso por `2 3 +`. Esta notação dispensa a utilização de parênteses e tem um implementação muito simples assente na utilização de uma pilha de números reais. Sempre que aparece um operando (número) ele é carregado para a pilha. Sempre que aparece um operador, são retirados os dois últimos números da pilha e o resultado da operação é colocado na pilha. Se a pilha não tiver o número de operandos necessário, então há um erro sintático na expressão.

O programa deve ler os operandos e os operadores do *standard input*, separados por espaços. Por exemplo, para calcular $(1 + 2) * 3$ poderá usar o comando:

```
$ echo "1 2 3 * +" | java -ea RPNCalculator
Stack: [1.0]
Stack: [1.0, 2.0]
Stack: [1.0, 2.0, 3.0]
Stack: [1.0, 6.0]
Stack: [7.0]
```

Exercício 10.5

Um dos problemas que os vectores em Java podem colocar reside no facto de o seu número de elementos ser imutável (uma vez criado o vector). Essa limitação faz com que, na presença de problemas onde a dimensão máxima do vector possa ter de variar em tempo de execução, sejamos obrigados a tirar cópias do vector (para um novo vector de dimensão adequada), ou a utilizar outros tipos de dados mais flexíveis.

Neste exercício pretende-se desenvolver uma classe `BlockArrayInt` com uma interface tipo vector (acesso aleatório aos elementos por intermédio de um índice inteiro), mas em que a representação interna desse vector assenta numa lista ligada de blocos, sendo cada bloco um vector de dimensão fixa (definida no construtor da classe). Se o número de elementos por bloco for `blockSize`, então os índices de `[0, blockSize-1]` apontarão para dentro do primeiro bloco (primeiro elemento da lista ligada), os índices `[blockSize, 2*blockSize-1]` apontarão para o segundo, etc.

Para testar a classe, o programa `TestBlockArray.java` armazena (num objecto do tipo `BlockArrayInt`) e escreve todos os números primos encontrados até um valor dado pelo utilizador como argumento do programa. Nesse ficheiro pode também encontrar a interface desejada para a classe, mas onde a implementação está incompleta (apenas define um bloco).

Comece por experimentar o programa com valores menores do que 233 para perceber o funcionamento do programa. Caso utilize um valor acima de 232 verificará que ocorre um erro em tempo de execução. Implemente a classe usando a representação interna acima descrita, por forma a que o programa funcione para qualquer valor numérico. Quando estiver a funcionar experimente utilizar o valor 1000000, por exemplo.