Recursão e iteração

- Considere por exemplo que queremos definir a operação de multiplicação, em termos da operação mais simples de adição (apenas como exemplo ilustrativo de definições usando recursão e iteração, pois certamente essa operação está disponível na linguagem).

- A multiplicação de um número inteiro por outro inteiro maior ou igual a zero pode ser definida por indução como a seguir:

$$
\begin{aligned}
&m \times 0 = 0 \\
&m \times n = m + (m \times (n-1)) \quad \text{se } n > 0
\end{aligned}
$$

Iteração

- Mais informalmente, multiplicar m por n (n não-negativo) é somar m, n vezes:

$$m \times n = \underbrace{m + \ldots + m}_{n \text{ vezes}}$$

- Solução de problema que realiza operações repetidamente pode ser implementada (em linguagens imperativas) usando comando de repetição (também chamado de comando iterativo ou comando de iteração).

Comandos de repetição

Comando `while`

$$\boxed{\texttt{while}\ (e)\ c}$$

Comandos de repetição

Comando `while`

$$\texttt{while}\ (e)\ c$$

Avalia e; se `false`,

Comandos de repetição

Comando `while`

$$\boxed{\texttt{while } (e)\ c}$$

Avalia e; se `false`, termina;
se `true`,

Comandos de repetição

Comando `while`

$$\texttt{while (}e\texttt{)}\ c$$

Avalia e; se `false`, termina;
se `true`, executa c e repete o processo.

Comandos de repetição

Comando `for`

$$\texttt{for (}c_0\texttt{; } e\texttt{; } c_1\texttt{) } c$$

Comandos de repetição

Comando `for`

$$\texttt{for (}c_0\texttt{; }e\texttt{; }c_1\texttt{) }c$$

Executa c_0.

Em seguida, faça o seguinte:

Comandos de repetição

Comando `for`

$$\texttt{for } (c_0\texttt{; } e\texttt{; } c_1\texttt{) } c$$

Executa c_0.

Em seguida, faça o seguinte:

avalia e;

se `false`,

Comandos de repetição

Comando `for`

$$\texttt{for (}c_0\texttt{; }e\texttt{; }c_1\texttt{) }c$$

Executa c_0.

Em seguida, faça o seguinte:

avalia e;

se `false`, termina;

se `true`,

Comandos de repetição

Comando `for`

$$\texttt{for (}c_0\texttt{; }e\texttt{; }c_1\texttt{) }c$$

Executa c_0.

Em seguida, faça o seguinte:

- avalia e;
- se `false`, termina;
- se `true`, execute c, depois c_1 e repita o processo.

Comandos de repetição

Comando `do while`

$$\texttt{do}\ c\ \texttt{while}\ (e)$$

Comandos de repetição

Comando `do while`

$$\texttt{do } c \texttt{ while } (e)$$

Executa c

Avalia e; se `false`,

Comandos de repetição

Comando `do while`

$$\texttt{do } c \texttt{ while } (e)$$

Executa c

Avalia e; se `false`, termina;
se `true`,

Comandos de repetição

Comando `do while`

$$\texttt{do}\ c\ \texttt{while}\ (e)$$

Executa c

Avalia e; se `false`, termina;

se `true`, repete o processo.

Exemplo de iteração com comando `for`

```
static int mult (int m, int n)
{ int r=0;
  for (int i=1; i<=n; i++) r += m;
  return r;
}
```

Iteração

- Exemplo a seguir segue passo a passo a execução de $mult$`(3,2)`
- São mostrados:
  - ⋆ Comando a ser executado ou expressão a ser avaliada
  - ⋆ Resultado (no caso de expressão)
  - ⋆ Estado (após execução do comando ou avaliação da expressão)

Detalhamento da execução de $mult$(3,2)

Comando/ Expressão	**Resultado** (expressão)	**Estado** (após execução/avaliação)
$mult(3,2)$	...	$m \mapsto 3,\ n \mapsto 2$
`int` r `= 0`		$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 0$
`int` i `= 1`		$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 0,\ i \mapsto 1$
i `<=` n	`true`	$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 0,\ i \mapsto 1$
r `+=` m	3	$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 3,\ i \mapsto 1$
i `++`	2	$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 3,\ i \mapsto 2$
i `<=` n	`true`	$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 3,\ i \mapsto 2$
r `+=` m	6	$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 6,\ i \mapsto 2$
i `++`	3	$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 6,\ i \mapsto 3$
i `<=` n	`false`	$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 6,\ i \mapsto 3$
`for ...`		$m \mapsto 3,\ n \mapsto 2,\ r \mapsto 6$
`return` r		
$mult$(3,2)	6	

Chamada de método

- Expressões — chamadas de parâmetros reais — são avaliadas, fornecendo valores dos argumentos.
- Argumentos são copiados para os parâmetros — também chamados parâmetros formais — do método.
- Corpo do método é executado.

Chamada de método

- Chamada cria novas variáveis locais.
- Parâmetros formais são variáveis locais do método.
- Outras variáveis locais podem ser declaradas (ex: r em $mult$).
- Quando execução de uma chamada termina, execução retorna ao ponto da chamada.

Comando `for`: terminologia

$$\text{for } (c_0;\ e;\ c_1)\ c$$

- c_0: *comando de "inicialização"*

  No caso em que é uma declaração, variável criada é comumente chamada de contador de iterações.

- e: *teste de terminação*

- c_1: *comando de atualização*

- c_1: *corpo*

Recursão

Definição indutiva dá origem a implementação recursiva:

```
static int multr (int m, int n)
{ if (n==0) return 0;
  else return (m + multr(m, n-1)); }
```

Recursão

- Cada chamada recursiva cria novas variáveis locais.
- Em chamadas recursivas, existem em geral várias variáveis locais de mesmo nome, mas somente as variáveis do último método chamado podem ser usadas (são acessíveis) diretamente.
- Quando execução de uma chamada recursiva termina, execução retorna ao método que fez a chamada.
- Assim, chamadas recursivas são executadas em estrutura de pilha.

Recursão

- Exemplo a seguir ilustra a execução de $multr$`(3,2)`
- São mostrados, passo a passo:
  - ⋆ Comando a ser executado ou expressão a ser avaliada
  - ⋆ Resultado (no caso de expressão)
  - ⋆ Estado (após execução do comando ou avaliação da expressão)

$multr(3,2)$	...	$m \mapsto 3$ $n \mapsto 2$		
n == 0	false	$m \mapsto 3$ $n \mapsto 2$		
return m+$multr(m, n-1)$	...	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	
n == 0	false	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	
return m+$multr(m, n-1)$	...	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	$m \mapsto 3$ $n \mapsto 0$
n == 0	true	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	$m \mapsto 3$ $n \mapsto 0$
return 0		$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	
return m + 0		$m \mapsto 3$ $n \mapsto 2$		
return m + 3				
$multr$ (3,2)	6			

Recursão

- Estrutura de pilha: último conjunto de variáveis (da pilha) são variáveis locais do último método chamado,
- penúltimo conjunto de variáveis são do penúltimo método chamado, e assim por diante.
- Espaço em memória de variáveis alocadas na pilha para um método é chamado de registro de ativação desse método.

Valor inicial de variáveis locais

- Registro de ativação é alocado no início e desalocado no fim da execução de um método.
- Variáveis locais a um método **não** são inicializadas automaticamente com valor *default*,

Valor inicial de variáveis locais

- Registro de ativação é alocado no início e desalocado no fim da execução de um método.
- Variáveis locais a um método **não** são inicializadas automaticamente com valor *default*,
- ao contrário de variáveis de objetos e de classes.

Variável local tem que ser inicializada “em todos os caminhos até seu uso”

Por exemplo, programa a seguir contém um erro:

```
import javax.swing.*;

class V
{ public static void main (String[] a)
  int x;
    boolean b = Boolean.valueOf(
      JOptionPane.showInputDialog("Digite \"true\" ou \"false\""")).booleanValue();
    if (b) x = Integer.parseInt(
      JOptionPane.showInputDialog("Digite um valor inteiro"));
    System.out.println(x); }

  }
```

Recursão simulando processo iterativo

```
static int multIter (int m, int n, int r)
{ if (n == 0) return r;
  else return multIter(m,n-1,r+m);
}
```

Recursão simulando processo iterativo

- Como na versão iterativa, a cada recursão valor de r ("acumulador") é incrementado de m.
- Diferença:

versão recursiva	acumulador é nova variável a cada recursão
versão iterativa	acumulador é a mesma variável em cada iteração

Exponenciação: implementações análogas

```
static int exp (int m, int n)
{ int r=1;
  for (int i=1; i<=n; i++) r*=m;
  return r; }

static int expr (int m, int n)
{ if (n==0) return 1;
  else return (m * expr(m, n-1)); }
```

$Math.pow$ e $Math.exp$

- `public static double` pow `(double` a`, double` b`)` fornece como resultado valor de tipo `double` mais próximo de a^b.

- `public static double` exp `(double` a`)` fornece como resultado valor de tipo `double` mais próximo de e^a (sendo e a base dos logaritmos naturais).

Eficiência

Definição indutiva da exponenciação:

$$
\begin{aligned}
m^0 &= 1 \\
m^n &= m \times m^{n-1} \quad \text{se } n > 0
\end{aligned}
$$

Definição alternativa (também indutiva):

$$
\begin{aligned}
m^0 &= 1 \\
m^n &= (m^{n/2})^2 && \text{se } n \text{ é par} \\
m^n &= m \times m^{n-1} && \text{se } n \text{ é ímpar}
\end{aligned}
$$

Eficiência

Definição alternativa dá origem a implementação mais eficiente:

```
static int exp2 (int m, int n)
{ if (n == 0) return 1;
  else if (n % 2 == 0) // n é par
          { int x = exp2(m, n/2);
            return x*x; }
       else return m * exp2(m,n-1);
}
```

Eficiência

Diferença em eficiência é significativa:

- Chamadas recursivas na avaliação de $exp2(m, n)$ dividem o valor de n por 2 a cada chamada
- na avaliação de $exp(m, n)$ valor de n é decrementado de 1 a cada iteração
- assim como na avaliação de $expr(m, n)$, valor de n é decrementado de 1 a cada chamada recursiva

Eficiência

- Exemplo: chamadas recursivas durante avaliação de $exp2$`(2,20)`:

$exp2$`(2,20)`	$exp2$`(2,10)`	$exp2$`(2,5)`	
$exp2$`(2,4)`	$exp2$`(2,2)`	$exp2$`(2,1)`	$exp2$`(2,0)`

- Quanto maior n, maior a diferença em eficiência.

- São realizadas da ordem de $log_2(n)$ chamadas recursivas durante avaliação de $exp2$`(`m`,`n`)` — uma vez que n é em média dividido por 2 em chamadas recursivas

- ao passo que avaliação de exp`(`m`,`n`)` requer n iterações.

Fatorial recursivo

$$n! = 1 \qquad \text{se } n = 0$$
$$n! = n \times (n-1)! \qquad \text{em caso contrário}$$

```
static int fatr (int n)
{ if (n == 0) return 1;
  else return n * fatr(n-1); }
```

Fatorial iterativo

$$n! = n \times (n-1) \times (n-2) \times \ldots 3 \times 2 \times 1$$

```
static int fat (int n)
{ int f=1;
  for (int i=1; i<=n; i++) f *= i;
  return f; }
```

Fatorial recursivo que se espelha no algoritmo iterativo

```
static int fatIter (int n, int i, int f)
  // fatIter(n,1,1) = n!  i funciona como contador de recursões e
  //                      f como acumulador (de resultados parciais)
{ if (i > n) return f;
  else return fatIter(n,i+1,f*i); }

static int fatr1 (int n)
{ return fatIter (n,1,1); } }
```

Progressão aritmética de passo 1
Implementação baseada em iteração

```
static int pa1 (int n)
{ int s = 0;
  for (int i=1; i<=n; i++) s += i;
  return s; }
```

Progressão aritmética de passo 1
Implementação recursiva

```
static int pa1r (int n)
{ if (n==0) return 0;
  else return n + pa1r(n-1); }
```

Progressão aritmética de passo 1
Recursão espelhando algoritmo iterativo

```
static int pa1Iter (int n, int i, int s)
{ if (i > n) return s;
  else return pa1Iter(n, i+1, s+i); }

static int pa1rIter (int n)
{ return pa1Iter(n,1,0); }
```

Progressão aritmética

- Exemplos apenas ilustrativos: seriam implementações mal feitas na prática, pois ineficientes. . .

- Uma vez que . . .

Progressão aritmética

- Exemplos apenas ilustrativos: seriam implementações mal feitas na prática, pois ineficientes. . .

- Uma vez que . . .

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Progressão geométrica: $\sum_{i=0}^{n} x^i$

- Implementação iterativa:

Progressão geométrica: $\sum_{i=0}^{n} x^i$

- Implementação iterativa:

```
static int pg (int n, int x)
{ int s = 1, parc = x;
  for (int i=1; i<=n; i++)
      { s += parc; parc *= x; }
  return s; }
```

- *parc* usada para evitar cálculo de x^i a cada iteração.

Progressão geomética

Mostre analogia pa-pg:

- Implemente pgr e $pgrIter$.
- Mostre que pg, pgr e $pgIter$ são ineficientes. . .

Progressão geomética

Mostre analogia pa-pg:

- Implemente pgr e $pgrIter$.
- Mostre que pg, pgr e $pgIter$ são ineficientes. . . deduzindo fórmula para cálculo direto de pgs.
- Dica?

Progressão geométrica

Mostre analogia pa-pg:

- Implemente pgr e $pgrIter$.
- Mostre que pg, pgr e $pgIter$ são ineficientes... deduzindo fórmula para cálculo direto de pgs.
- Dica? multiplique $s = \sum_{i=0}^{n} x^i$ por

Progressão geométrica

Mostre analogia pa-pg:

- Implemente pgr e $pgrIter$.
- Mostre que pg, pgr e $pgIter$ são ineficientes. . . deduzindo fórmula para cálculo direto de pgs.
- Dica? multiplique $s = \sum_{i=0}^{n} x^i$ por $-x$ e

Progressão geométrica

Mostre analogia pa-pg:

- Implemente pgr e $pgrIter$.
- Mostre que pg, pgr e $pgIter$ são ineficientes. . . deduzindo fórmula para cálculo direto de pgs.
- Dica? multiplique $s = \sum_{i=0}^{n} x^i$ por $-x$ e some a s.

Implementação de somatórios

- Usar variável para armazenar soma.
- Decidir se parcela a ser somada vai ser obtida da parcela anterior ou do contador de iterações.
- No 1º caso, usar variável para armazenar valor calculado na parcela anterior (como $parc$ em pg).
- Exemplo do 2º caso: $\sum_{i=1}^{n} \frac{1}{i}$

Implementação de somatórios

- Em vários casos, cálculo não usa a própria parcela anterior, mas valores usados no cálculo dessa parcela.

- Exemplo: cálculo aproximado do valor de π, usando:

$$\pi = 4 * (1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots)$$

- Precisamos guardar não valor mas sinal e denominador da parcela anterior.

Implementação de somatórios

```
static float piAprox (int n)
{ float s = 0.0f, denom = 1.0f; int sinal = 1;
  for (int i=1; i<=n; i++)
        { s += sinal/denom;
          sinal = -sinal; denom += 2; }
  return 4 * s; }
```

Implementação de somatórios

$$e^x = 1 + (x^1/1!) + (x^2/2!) + \ldots$$

```
static float eExp (float x, int n)
{ float s = 1.0f; int i=1;
  float numer = x; int denom = 1;
  while (i<=n)
      { s += numer/denom;
        i++;
        numer *= x; denom *= i; }
  return s; }
```

Não-terminação

Podem ocorrer programas cuja execução, em princípio, não-termina:

```
static int infinito()
{ return infinito() + 1; }
```

```
static void cicloEterno()
{ while (true) ; }
```

Não-terminação

Existem programas cuja execução não termina apenas em alguns casos (para alguns valores de entrada). Exemplo:

```
static int fat (int n)
{ int f=1;
  for (int i=1; i!=n; i++) f *= i;
  return f; }
```

Seleção múltipla

(seleção de um dentre vários casos)

```
switch (e)
{ case e1:  c1;
  case e2:  c2;
  ...
  case en:  cn;
}
```

Expressão e avaliada.

Executado então primeiro comando c_i $(1 \leq i \leq n)$, caso exista, para o qual $e = e_i$.

Se não for executado comando de "saída anormal" (como `break`), são também executados comandos $c_{i+1}, \ldots, c_n$, se existirem, nessa ordem.

Comando `break` e caso `default`

- Execução de qualquer c_i pode ser finalizada (e geralmente deve ser) por meio do comando `break`.
- Se $e \neq e_i$, para todo $1 \leq i \leq n$, caso *default* pode ser usado.

Veja exemplo a seguir:

Comando `break` com caso `default`

```
static double op (char c, double a, double b)
{ switch (c)
  { case '+':  { return a + b; }
    case '*':  { return a * b; }
    case '-':  { return a - b; }
    case '/':  { return a / b; }
    default:   { System.out.println
       ("Caractere diferente de +,*,-,/");
                 return 0.0; }convenção
  }
}
```

Caso `default`

- `default` pode ser usado no lugar de `case` e_i, para qualquer $i = 1, \ldots, n$.
- Em geral usado depois do último caso.
- Se `default` não for especificado, execução de `switch` pode terminar sem que nenhum dos c_i seja executado (isso ocorre se resultado da avaliação de $e \neq e_i$, para $i = 1, \ldots, n$).

Comando `switch`: crítica e restrições

- Necessidade de uso de `break` sempre que se deseja executar apenas uma alternativa em comando `switch` considerada ponto fraco de Java (herança de C).
- Expressão e deve ter tipo `int`, `short`, `byte` ou `char`, e deve ser compatível com tipo de $e_1, \ldots, e_n$.
- Expressões $e_1, \ldots, e_n$ têm que ser valores constantes e distintos.

`break` seguido de nome de rótulo

- Comandos `switch` e comandos de repetição podem ser precedidos de rótulo: nome seguido do caractere ":".
- `break` pode ser seguido de nome de rótulo.
- Ao ser executado, tal comando causa terminação da execução do comando precedido pelo rótulo.

Exemplo: `break` seguido de nome de rótulo

```
static String ident (String s)
/* Procura marca em s; se encontrar, retorna
 * marca encontrada; caso contrário, null.
 * Supõe: marca = cadeia de caracteres que:
 *    começa com caractere '<'
 *    segue cadeia de caracteres não contendo '<','>'
 *    termina com caractere '>'
 */
```

```
{ int i = 0; String marca;
  while (i < s.length())
  { pesq:  while (true)
     { while (s.charAt(i) != '<') { inci }
       marca = "<"; inci
       while (s.charAt(i)!='<' && s.charAt(i)!='>')
       { marca+=Character.toString(s.charAt(i));inci}
       if (s.charAt(i) == '>')
       { marca += ">"; return marca; }
       else break pesq;
     } return null; } }
```

Abreviação: `inci` = `i++; if (i>=s.length()) return null;`