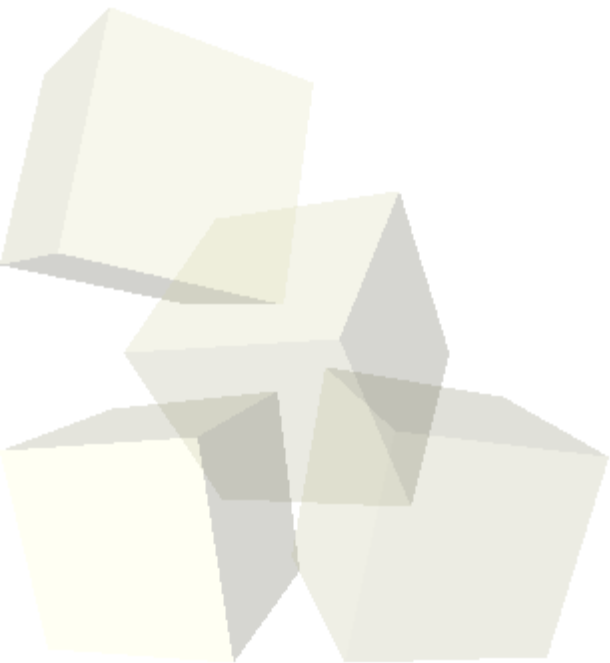


# **SCE-213**

## **Programação Orientada a Objetos**

**Rodrigo Fernandes de Mello**  
**<http://www.icmc.usp.br/~mello>**  
**[mello@icmc.usp.br](mailto:mello@icmc.usp.br)**





- Fundamentos das Linguagens de Programação utilizadas para exemplificar os conceitos
- Estudo de conceitos de orientação a objetos:
  - ♦ Classes
  - ♦ Objetos
  - ♦ Métodos
  - ♦ Mensagens
  - ♦ Encapsulamento
  - ♦ Herança
  - ♦ Sobrecarga
  - ♦ Polimorfismo
  - ♦ Construtores e Destrutores



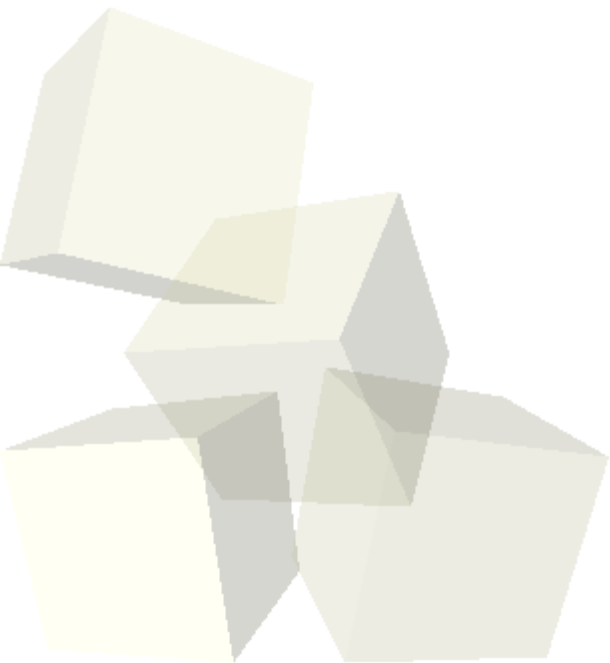
- Utilização das seguintes linguagens para exemplos:
  - ♦ C++
  - ♦ Java
- Condução de, ao menos, um exemplo em C++ e outro em Java para cada conceito
- UML:
  - ♦ Diagrama de Casos de Uso
  - ♦ Diagrama de Classes
  - ♦ Diagrama de Seqüência
- Trabalhos



- Deitel, H. M.; Deitel, P. J., C++ How to Program, Prentice Hall, 2001
- Deitel, H. M.; Deitel, P. J., Java How to Program, Prentice Hall, 2002
- Ghezzi, C.; Jazayeri, M., Programming Language Concepts, John Wiley & Sons, 1998
- Holzner, S., Programando em C++, Editora Campus, 1994
- Dewhurst, S. C.; Stark, K. T., Programando em C++, Editora Campus, 1990

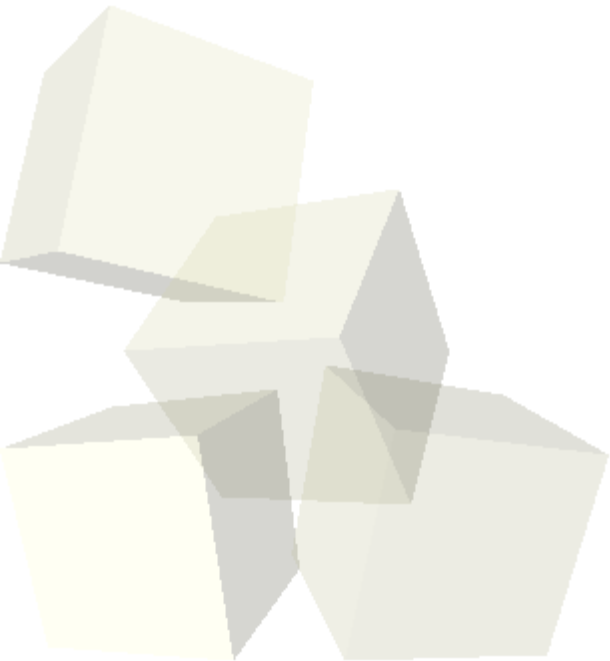


- 2 Provas:  $MP = (P1*2 + P2*3)/5$ 
  - ♦ P1 (peso 2)
  - ♦ P2 (peso 3)
- Não haverá SUB
- Trabalhos – Peso 0.2 (MP peso 0.8)





- P1 – 23/09/2008
- P2 - 04/12/2008



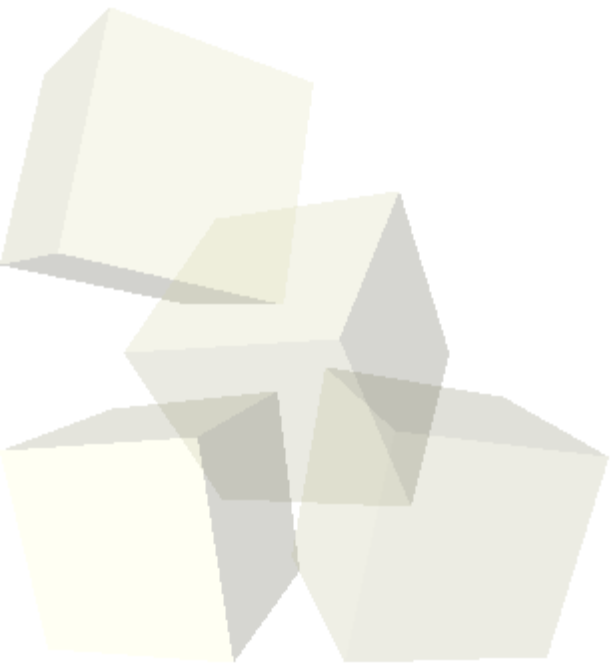


## ■ Sala 3-162

- ♦ Terças das 14hs – 17hs

## ■ Monitoria

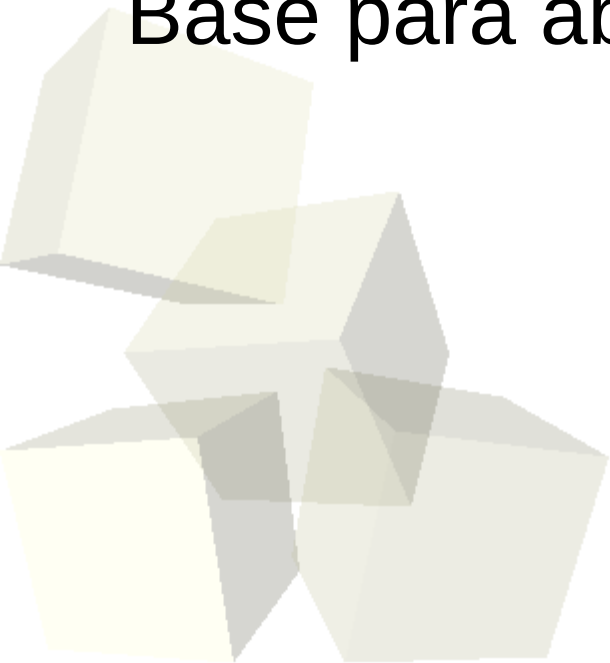
- ♦ Marcelo Albertini ([marcelo.albertini@gmail.com](mailto:marcelo.albertini@gmail.com))
- ♦ Sala 3-162
- ♦ Quartas das 14hs – 17hs





# Fundamentos das linguagens C++ e Java

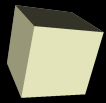
Base para abordar os conceitos de Programação  
Orientada a Objetos





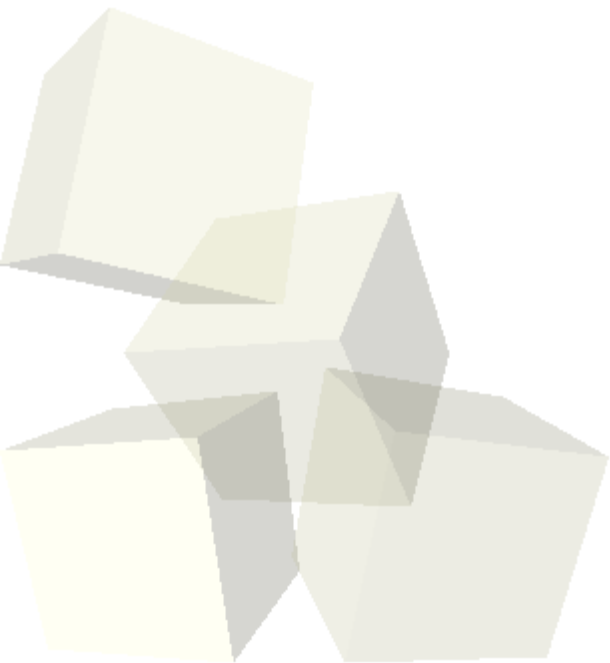


- Pensar em alto nível
- Foco nos dados e funções
- Extensão da linguagem C
  - ♦ Adota:
    - Tipos básicos de dados
    - Operações
    - Sintaxe das instruções
    - Estrutura de programa
  - ♦ Adicionou:
    - Classes e outras melhorias
- Execução veloz (contudo ainda pode ser até 40% mais lenta que C devido, principalmente, ao polimorfismo)
- Autor: Bjarne Stroustrup



# Tipos de dados

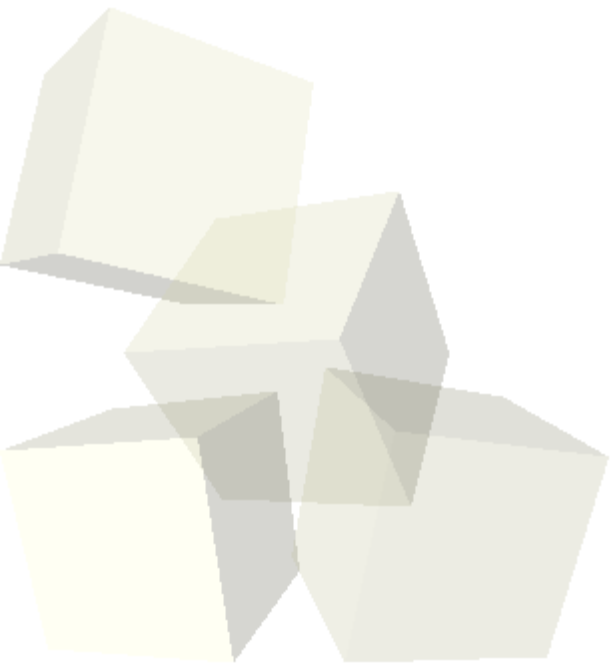
- void
- char
- int
- float
- double

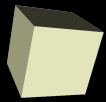




# Operadores aritméticos

- Subtração e menos unário (-)
- Adição (+)
- Multiplicação (\*)
- Divisão (/)
- Resto da divisão (%)
- Decremento (--)
- Incremento (++)





# Operadores Relacionais e Lógicos

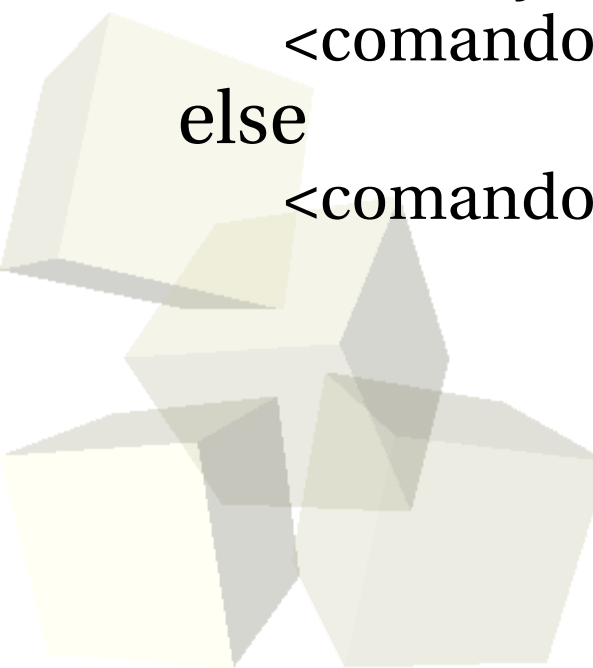
- Maior ( $>$ )
- Maior ou igual ( $\geq$ )
- Menor ( $<$ )
- Menor ou igual ( $\leq$ )
- Igual ( $==$ )
- Diferente ( $\neq$ )
- And ( $\&\&$ )
- Or ( $\|\|$ )
- Not ( $!$ )



- Os operadores >> e << foram sobrecarregados para executar operações de entrada e saída
- Todo programa, ao ser executado, recebe 4 referências para dispositivos externos
  - ♦ cin (entrada padrão)
  - ♦ cout (saída padrão)
  - ♦ cerr (saída de erros)
- Exemplos:
  - ♦ cin >> x;
  - ♦ cout << "Nome: " << n;
  - ♦ cerr << "Erro nr. " << nerro << "\n";
- Exemplo - ex01



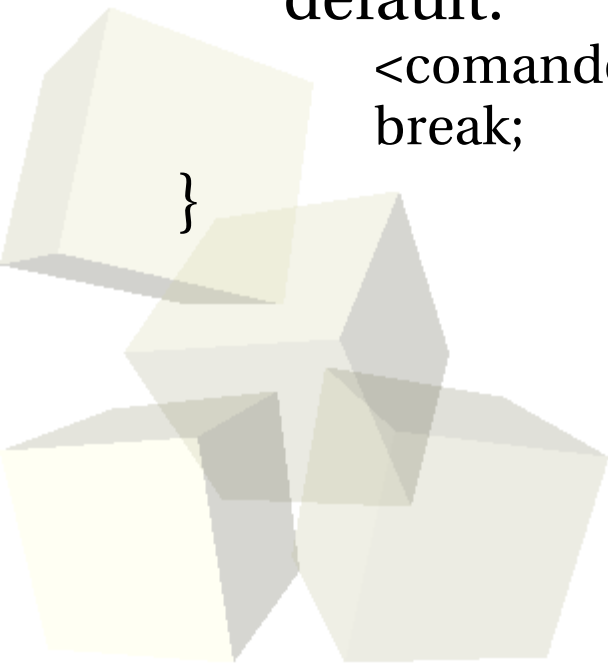
- Comando for
  - for (<inicialização>; <teste>; <incremento>)
  - <Comando a ser executado>
- Comando while
  - ♦ while (<condição>)
  - <Comando a ser executado>
- Comando if
  - if (<condição>)
  - <comando 1>
  - else
  - <comando 2>





■ Comando switch

```
switch (<condição>)  
{  
    case <valor 1> :  
        <comando>;  
        break;  
    case <valor 2> :  
        <comando>;  
        break;  
    default:  
        <comando>;  
        break;  
}
```





- C++ permite funções com parâmetros default
  - Todos os parâmetros que assumem valores default devem aparecer à direita dos que não assumem
  - É uma técnica que vale a pena quando a maior parte das vezes o parâmetro recebe o valor default

```
void teste(char *str, int x=0, int y=1)
```

```
{  
    cout << "string = " << str << endl;  
    cout << "x = " << x << endl;  
    cout << "y = " << y << endl;  
}
```

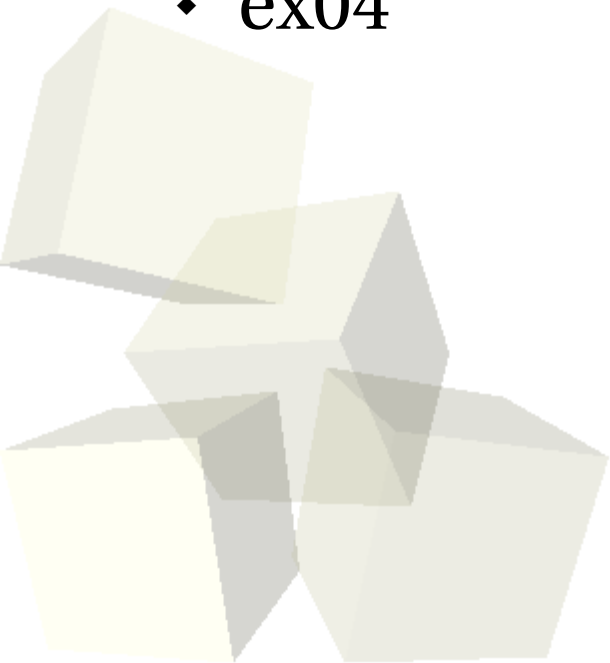
■ Exemplo - ex02





# Número variável de parâmetros

- Pode-se especificar o tipo dos argumentos iniciais e deixar o número e tipo dos demais argumentos sem definição fixa
- Exemplo:
  - ♦ `int printf(const char *, ...)`
- Exemplos:
  - ♦ ex03
  - ♦ ex04





- Uma função inline é expandida através da inserção de uma cópia de seu código no local onde foi chamada, de maneira similar às macros da linguagem C
  - ♦ Contudo, ao contrário das macros em C, uma função inline tem a mesma semântica das funções comuns
  - ♦ Exemplo de macro em C:  

```
#define multiplica( a, b ) a * b
```

```
int r = multiplica( m * 5 + 3, n / 2 ) ;
```
- A macro será expandida em  $m * 5 + 3 * n / 2$ 
  - ♦ Isso é processado durante a compilação
- Exemplo - ex05
- Uma função inline é eficiente (executa mais rápido, pois não há pilha, há a inserção de código), entretanto o tamanho do programa aumenta



## ■ Declaração:

- ♦ Dentro do header file (.h)

```
class G
{
    // ...
    int sqr(int i)
    {
        return i*i;
    }
};
```

- ♦ Dentro do arquivo c++ (.cc ou .cpp)

```
inline int sqr(int i)
{
    return i*i;
}
```

## ■ Exemplo - ex06



## ■ Retorno dentro de uma seleção ou repetição

```
inline int Calcula( int Codigo)
{
    switch( Codigo )
    {
        case 0 :
            // ...
        case 1 :
            // ...
            return 1;
    }
    return 0;
}
```

## ■ Função void com return:

```
inline void Mostra( int Modo )
{
    // ...
    if( Modo == 0)
        return;
}
```



## ■ Função recursiva

```
int Fatorial( int i )  
{  
    if( i == 1 )  
        return 1;  
    else  
        return i * Fatorial( i - 1 );  
}
```

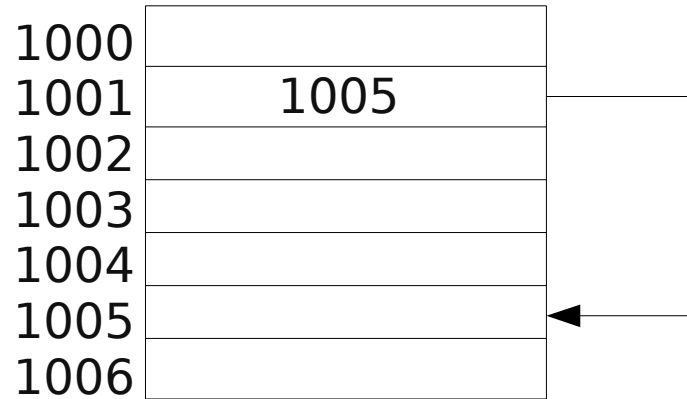
## ■ Função com variável static

```
inline void Incrementa()  
{  
    static int i = 0;  
    i ++;  
}
```

## ■ Exemplo - ex05



- Ponteiros são variáveis que armazenam endereços



- Declaração:
- `<Tipo> *<Nome>;`



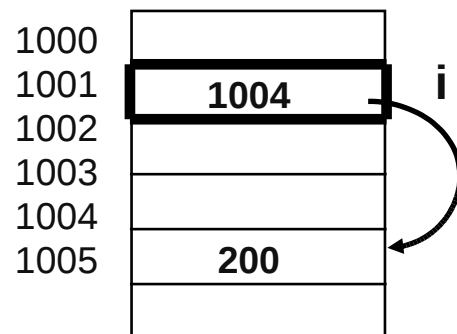
## ■ Operador &

- Obtem o endereço da variável

## ■ Operador \*

- Obtem o valor armazenado no endereço de memória apontado pela variável

## ■ Exemplos



```
int *i;
```

```
&i  => endereço de i ..... 1001
```

```
i    => conteúdo de i ..... 1004
```

```
*i   => valor apontado por i .... 200
```

## • Exemplo - ex08



# Ponteiros - Vetores

## ■ Declaração

<Tipo> <Nome> [ < Tamanho > ] ;

- Na declaração de um vetor a memória é alocada e a variável armazena o endereço do início do vetor

## ■ Exemplo

```
int V[3];
```

```
cout << V [ 2 ];
```

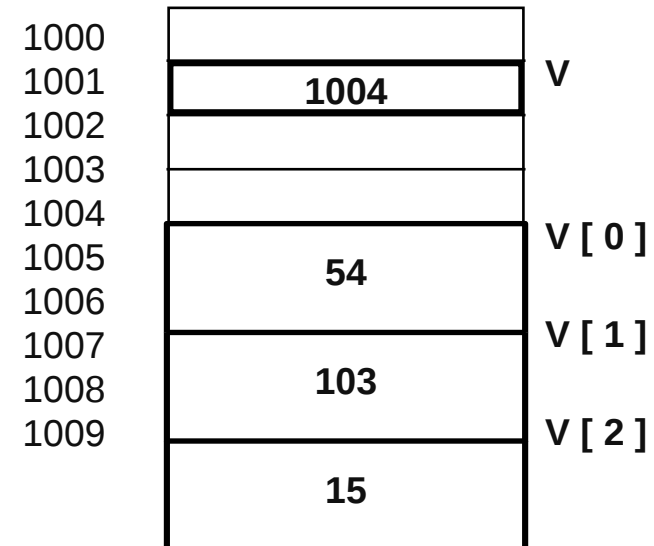
```
// operador [ ] calcula
```

```
// a posição no vetor
```

```
// de acordo com o
```

```
// tipo ( int )
```

```
// ( 1004 + 2 * 2 )
```



- Exemplo - ex09



# Ponteiros - Vetores de caracteres

```
main( )
{
    char Str[ ] = "Dia" ;
    char *P;
    int    i;

    for( i=0 ; i < 3 ; i++)
        cout << Str[ i ] ;

    P = Str ;
    while ( * P != '\0' )
    {
        cout << * P ;
        P++;
    }
}
```

- Exemplo - ex10

1000		
1001	1004	Str
1002		
1003	1004	P
1004		
1005		Str [ 0 ]
1006	D	Str [ 1 ]
1007	i	Str [ 2 ]
1008	a	Str [ 3 ]
1009	\0	Str [ 4 ]

# Ponteiros – Alocação Dinâmica de Memória

```
main( )
```

```
{
```

```
    char *P;
```

```
    // P = malloc ( 4 );
```

```
    P = new char[4];
```

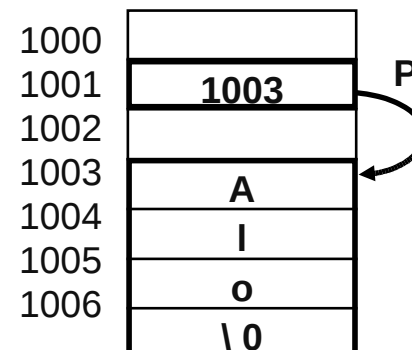
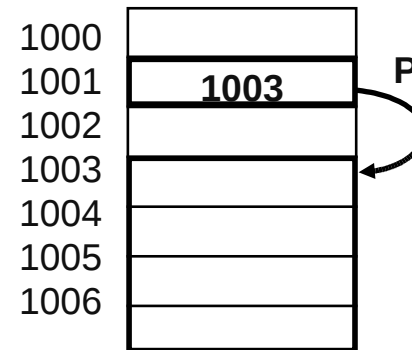
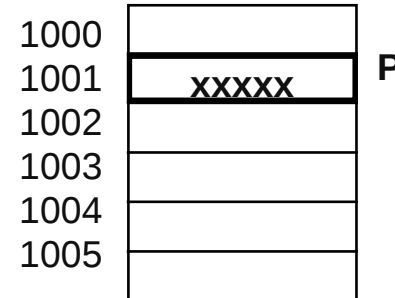
```
    P = "Alo" ; // erro !!
```

```
    strcpy ( P , "Alo" );
```

```
    free( P );
```

```
}
```

- Exemplo - ex11

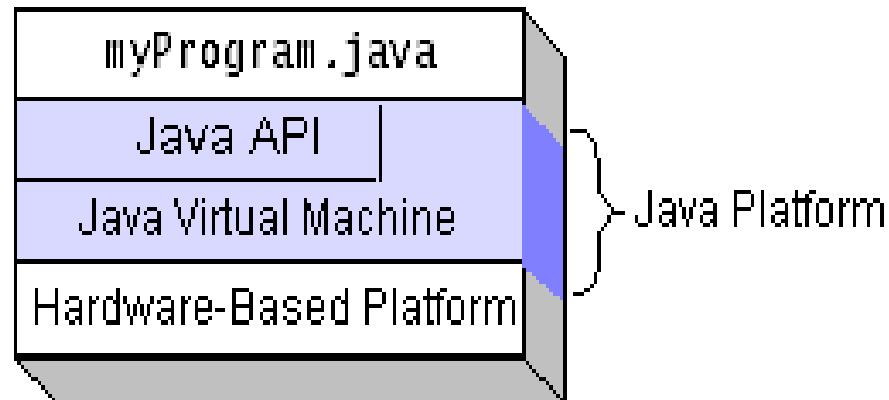




## ■ O que é Java?

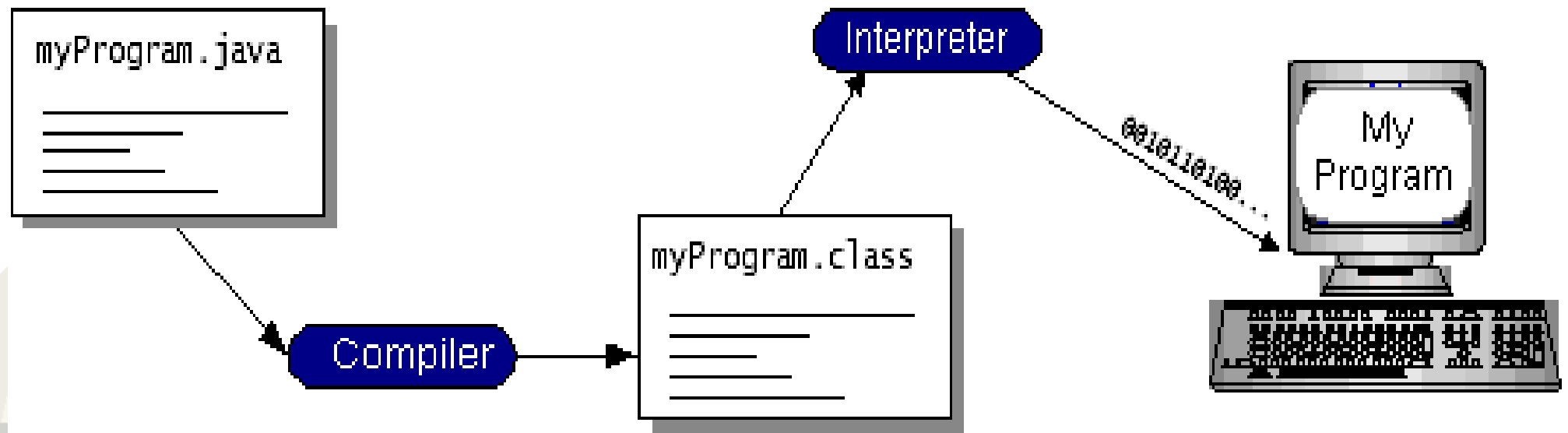
### ♦ Plataforma

- API
- Compilador
- Linguagem
- Máquina Virtual
- Outras Ferramentas
  - JNI (Java Native Interface)
  - Java/Corba (Geração de IDLs)
  - Java/RMI (Geração de Stubs/Skeletons)





# Compilação e Interpretação

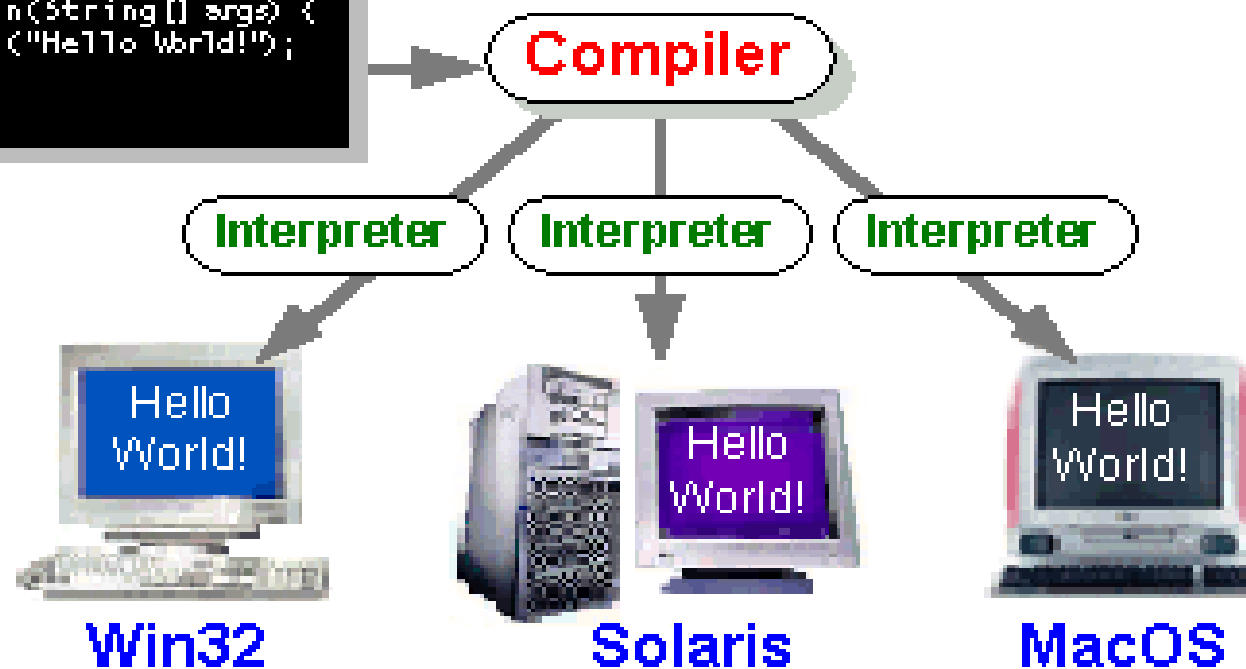




## Java Program

```
class HelloWorldApp {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

HelloWorldApp.java





# Primeiro Exemplo

PrimeiroExemplo.java

```
public class PrimeiroExemplo {  
  
    public static void main(String args[]) {  
        System.out.println("Primeiro Exemplo");  
    }  
}
```

Configurar CLASSPATH:

Linux (Bash) – Bom colocar todos pacotes no CLASSPATH!!!

export CLASSPATH=/usr/local/j2sdk/src.jar::

Windows

set CLASSPATH=c:\j2sdk\src.jar;.

Compilar:

javac PrimeiroExemplo.java

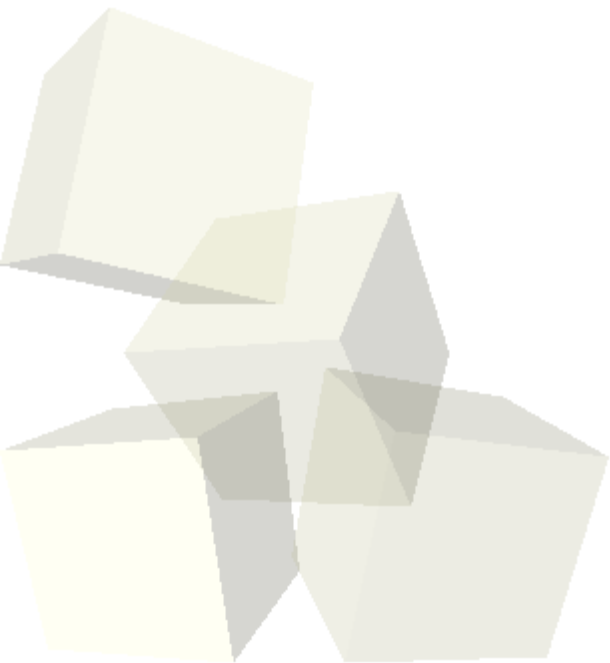
Executar:

java PrimeiroExemplo

•Exemplo - ex12



- Todas as classes tornam-se acessíveis para o Compilar e Máquina Virtual uma vez que tenham sido declaradas no CLASSPATH
- Pode-se adicionar algo ao CLASSPATH:
  - Linux(bash): `export CLASSPATH=$CLASSPATH:/usr/`
  - Windows: `set CLASSPATH=%CLASSPATH%;c:\`
- Uso do ponto (.) no CLASSPATH

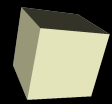




# Alguns de Seus Recursos

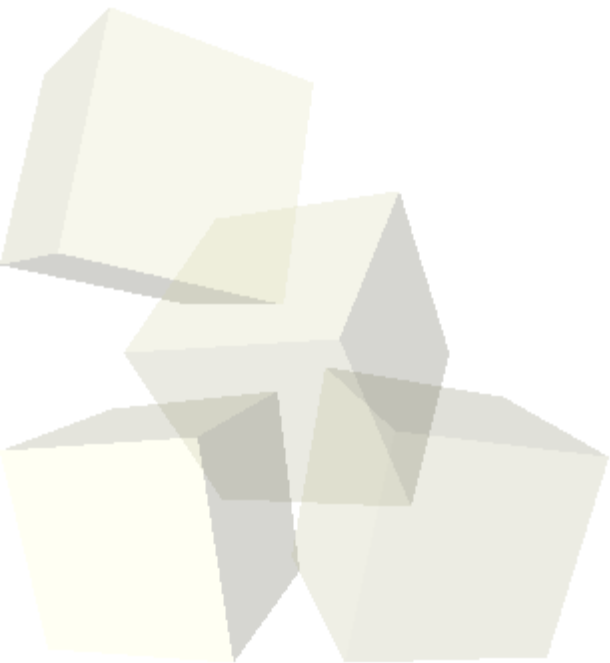
- Util
  - ♦ Vector, Array, Hashtable
- SQL
- Conexões com BD via JDBC
- Suporte a XML
- Corba
- RMI
- Threads
- I18n
- Packages
- Applets, etc

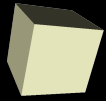




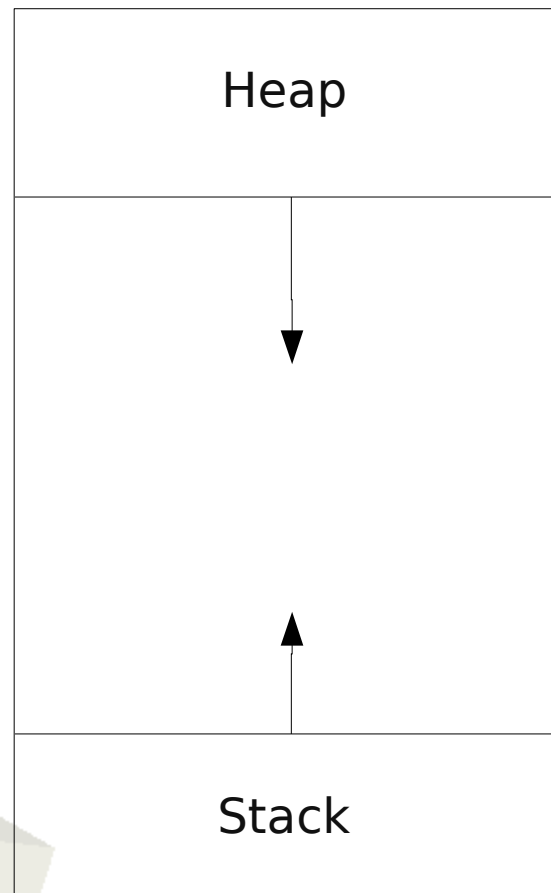
# Principais Diferenças Entre Java e C++

- Não trabalha de forma explícita com Ponteiros
- Interpretada
- Gerenciamento de Memória através do Coletor de Lixo



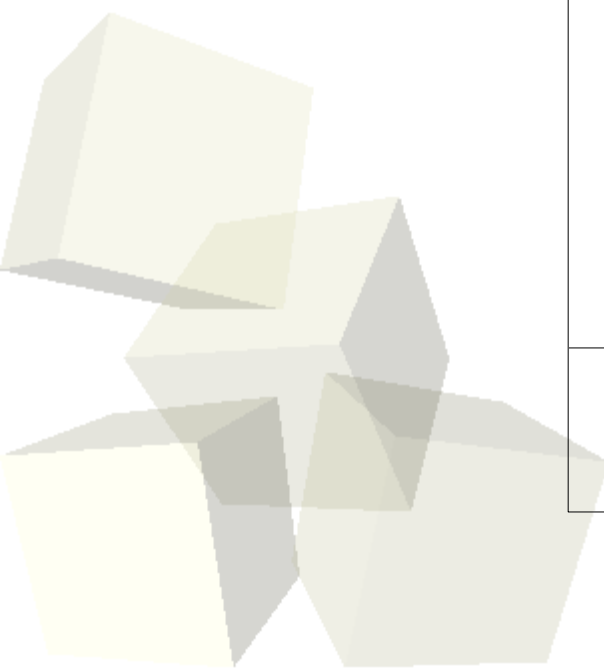


# Gerenciamento de Memória em C++



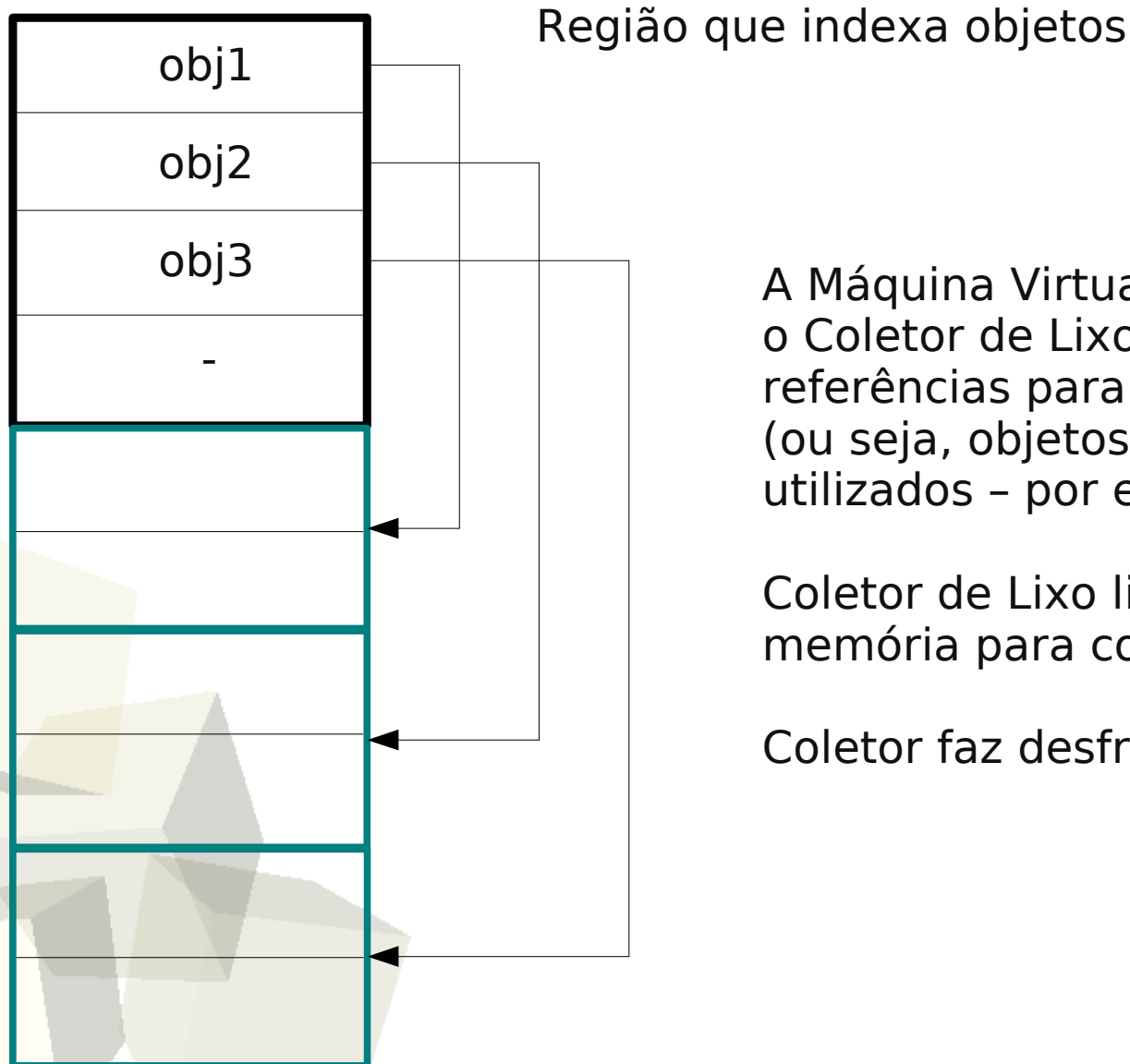
Tudo que é dinâmico!!!  
Deve ser desalocado!!!  
P.E.: Problemas com CGIs

Tudo que é estático!!!





# Coletor de Lixo



A Máquina Virtual ativa, periodicamente, o Coletor de Lixos para observar se as referências para objetos foram quebradas (ou seja, objetos não mais podem ser utilizados – por escopo etc)

Coletor de Lixo limpa as regiões de memória para conter novos objetos.

Coletor faz desfragmentação se necessário



- If, Switch, While, Do-While e For semelhantes aos da linguagem C e C++
  - ♦ Diferença: Não aceita condições tais como  
if (1) {} ou if (0) {}
  - ♦ Aceita somente um resultado booleano na condição
- Assim como C e C++ tem:
  - ♦ Return
  - ♦ Break
  - ♦ Continue
  - ♦ Mesmo formato de comentários  
// comentário  
/\* comentário \*/



## ■ Comentários

*// texto*

Todos os caracteres de *//* para o final da linha são ignorados.

*/\* texto \*/*

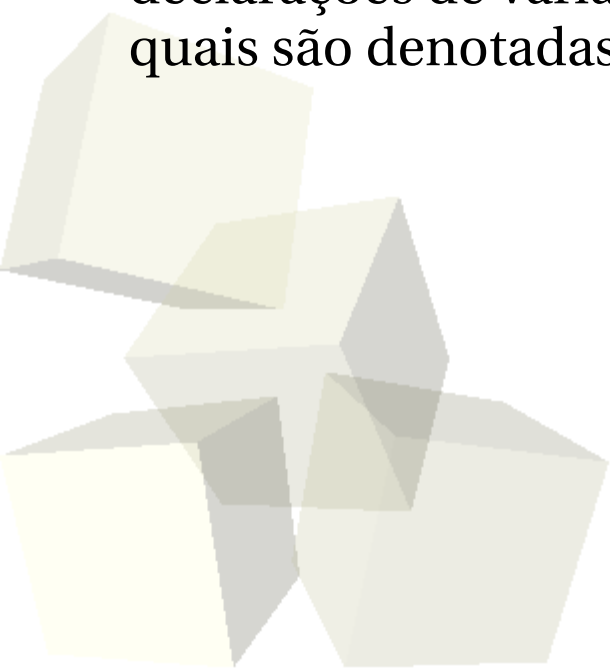
Todos os caracteres de */\** para *\*/* são ignorados.

*/\*\* texto*

*\* @ consulte java.applet.Applet*

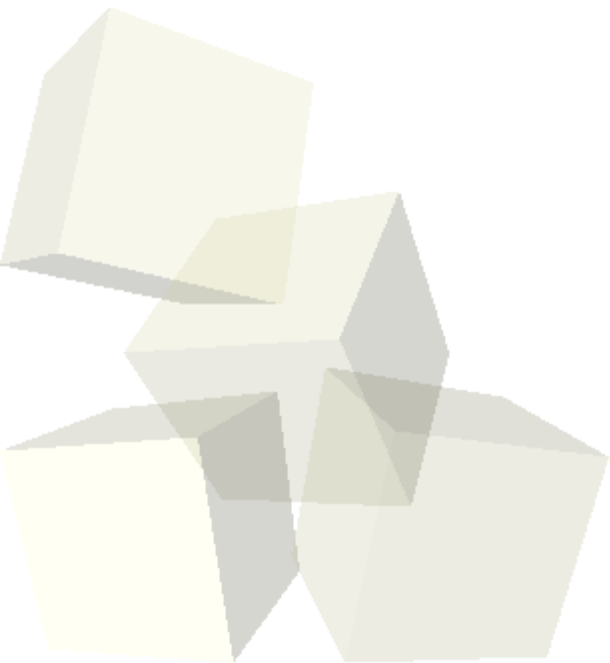
*\*/*

*Comentário de documentação* : é uma forma especial de comentário usado pela ferramenta javadoc. Deve ser usado antes da classe pública, do método e das declarações de variáveis. O javadoc reconhece diversas variáveis especiais, as quais são denotadas por @ (sinal de arroba) dentro desses comentários.





- Devem começar com uma letra, sublinhado (“\_”) ou sinal de cifrão (“\$”).
- Java é sensível a maiúsculas e minúsculas. (Case sensitive)





# Caracteres Especiais

Os caracteres em Java são índices do conjunto de caracteres Unicode que são valores de 16 bits que podem ser convertidos em inteiros e manipulados com operadores inteiros. Um character inteiro é representado dentro de um par de apóstrofos ( ' '). Para os caracteres que não podem ser inseridos diretamente a tabela abaixo mostra algumas seqüências de escape de caracteres.

<code>\ddd</code>	<i>Character octal (ddd)</i>
<code>\uxxxx</code>	<i>Character UNICODE hexadecimal (xxxx)</i>
<code>\'</code>	<i>Apóstrofo</i>
<code>\"</code>	<i>Aspas</i>
<code>\\</code>	<i>Barra Invertida</i>
<code>\r</code>	<i>Retorno de carro</i>
<code>\n</code>	<i>Nova linha (linefeed)</i>
<code>\f</code>	<i>Form feed</i>
<code>\t</code>	<i>Tab</i>
<code>\b</code>	<i>Backspace</i>



- ⇒ Blocos de declaração composta são demarcados por {}.
- As variáveis são válidas a partir do ponto onde são declaradas até o final da declaração composta.
  - Não é permitido declarar duas variáveis distintas com o mesmo nome, como no C++.

```
class Escopo{  
    public static void main(String args[ ]) 0 {  
        int var = 10; { // cria um novo escopo  
            int var = 20; //Erro de tempo de compilação...  
        }  
    }  
}
```

- Exemplo - ex13

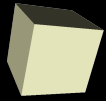




# Tipos Primitivos

<u>Tipo</u>	<u>Largura</u>	<u>Faixa</u>
byte	8	-128 a 127
short	16	-32768 a 32767
int	32	-2.147.483.648 a 2.147.483.647
long	64	-9.223.372.036.854.775.808... 9.223.372.036.854.775.807
float	32	3,4e - 308..1,7e + 308
double	64	1,7e - 038..3,4e + 038
char	16	0 a 65536
boolean		true ou false

⇒ Exemplo - ex14



# Operadores Aritméticos e Relacionais

<u>Operador</u>	<u>Ação</u>
+	Adição
-	Subtração e menos unário
*	Multiplicação
/	Divisão
%	Módulo (resto da divisão)
+	Incremento
+=	Atribuição Aditiva
-=	Atribuição Subtrativa
*=	Atribuição de Multiplicação
/=	Atribuição de Divisão
%=	Atribuição de Módulo (resto de divisão)
--	Decremento

<u>Operador</u>	<u>Ação</u>
==	Igual a
!=	Diferente
>	Maior do que
<	Menor do que
>=	Maior ou
igual a	
<=	Menor ou
igual a	



# Operadores Bit-a-Bit

<u>Operador</u>	<u>Ação</u>
~	NOT bit-a-bit unário
&	AND bit-a-bit
	OR bit-a-bit
^	OR exclusivo bit-a-bit
>>	Deslocamento à direita (shift right)
>>>	Com zeros
<<	Deslocamento à esquerda (shift left)
&=	Atribuição AND bit-a-bit
=	Atribuição OR bit-a-bit
^=	Atribuição OR exclusivo bit-a-bit
>>=	Atribuição de deslocamento à direita
>>>=	Atribuição de deslocamento à direita com preenchimento
<<=	Atribuição de deslocamento à esquerda

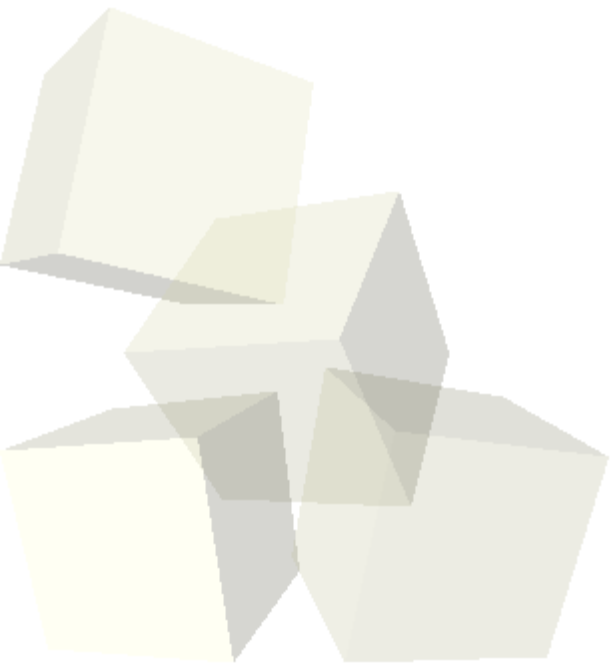


# Operadores Lógicos Booleanos

<b><u>Operador</u></b>	<b><u>Ação</u></b>
&	AND lógico
	Lógico OR
^	Lógico XOR (OR exclusivo)
	OR dinâmico
&&	AND dinâmico
!	NOT unário lógico
&=	Atribuição de AND
=	Atribuição de OR
^=	Atribuição de XOR
==	Igual a
!=	Diferente de
?:	if-then-else ternário



# Conceitos de Programação Orientada a Objetos





# Mudança de Paradigma

## ■ Paradigma Imperativo

- Modularização em arquivos
- Uso de procedimentos e funções
- Função dissociada dos dados
- Reuso manual de código
- Programação focada em procedimentos e funções

## • Paradigma OO

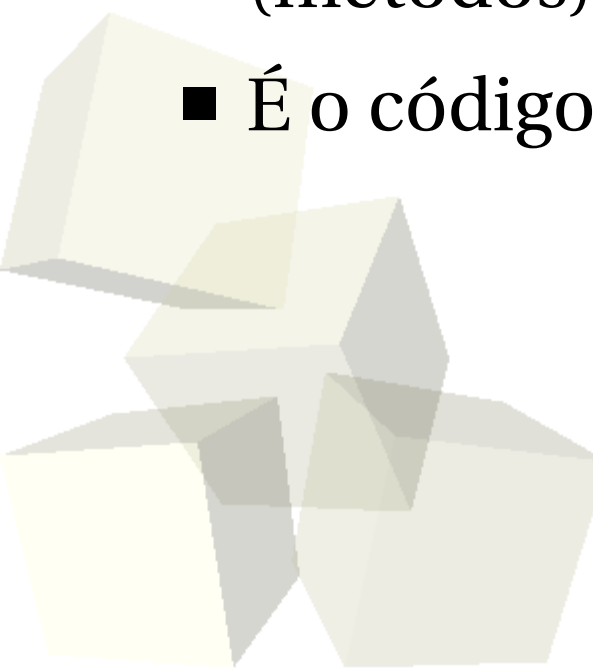
- Modularização em Classes
- Uso de métodos (associação da função e dados)
- Reuso de código através de herança
- Programação voltada a objetos que apresentam características (identificadores) e funções (métodos)



- Classes
- Objetos
- Métodos
- Mensagens
- Encapsulamento
- Herança
- Sobrecarga
- Polimorfismo
- Construtores e Destrutores



# Classes

- Tipo de Dado
  - Contém características (identificadores)
  - Contém ações (métodos)
  - É o código escrito!
- 

# Objetos

- Instância de uma Classe (na memória)
- Especifica as características
- Pode executar ações





# Exemplo

## Classe

```
class Pessoa {  
    String cpf;  
    String nome;  
    String endereco;  
    public mudaEndereco(String novo) {  
        endereco = novo  
    }  
}
```

## Memória Principal

Objetos

João
Pedro
Paulo



- A definição de novos tipos pelo usuário é feita utilizando uma estrutura chamada class
- Uma class permite armazenar dados e funções
- Forma Geral da class:

```
class Nome : [ public / private ] classePai {  
    private:  
        // dados e funções membros private (padrão)  
    protected:  
        // dados e funções membros protected  
    public:  
        // dados e funções membros public  
};
```

- Exemplo - ex15



- Uma struct também pode ser utilizada para armazenar dados e funções
- A única diferença entre uma class e uma struct é que os membros de uma class são private por default, enquanto que na struct são public
- Exemplo - ex16

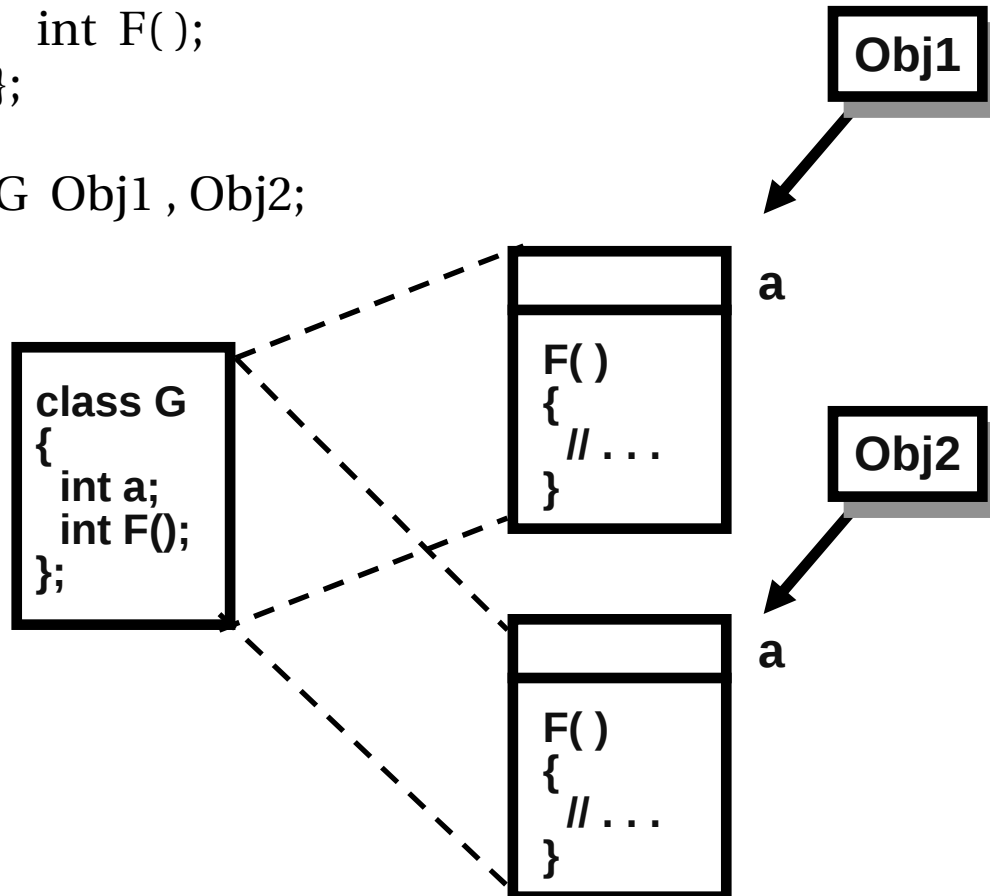


- Objetos são instâncias de uma class

- Exemplo

```
class G
{
    int a;
    int F();
};
```

```
G Obj1 , Obj2;
```





- O acesso aos dados membros de uma class deve ser feita através de suas funções membros
- Utiliza-se o operador . seguido do nome da função

### Observação

Em ponteiros para objetos utiliza-se o operador ->

### ■ Exemplo

```
class G
{
    int a;
public:
    int GetA() { return a; }
};

main()
{
    G i, *j;
    // ...
    cout << i.GetA();
    cout << j->GetA();
}
```

- Exemplo - ex17



# Exemplo em C++

```
#define Pot2( a )  a * a          // macro para potencia de 2

class Ponto
{
    int x;                        // private por "default"
    int y;
public:                           // funções de interface
    Ponto( int X , int Y )       // construtor
    {
        x = X;
        y = Y;
    }
    int GetX() { return x; }
    int GetY() { return y; }
    float Distancia( Ponto P )
    {
        float aux;
        aux=sqrt( Pot2( x * P.GetX() ) + Pot2( y *P.GetY() ) );
        return aux;
    }
};

main ( )
{
    Ponto A( 2 , 5 ) , B ( 10 , 8 );

    cout << "Distancia : " << A.Distancia ( B );
}
```

- Exemplo - ex18



A forma geral de uma definição de classe aparece a seguir:

```
class nome-classe extends nome-superclasse {  
    tipo var-instancia1;  
    tipo var-instancia2;  
    tipo var-instanciaN;  
    tipo nome-método1 (lista-de-parametros) {  
        corpo do método;  
    }  
    tipo nome-método2 (lista-de-parametros) {  
        corpo do método;  
    }  
    tipo nome-métodoN (lista-de-parametros) {  
        corpo do método;  
    }  
}
```

Uma outra maneira de representar uma classe é da seguinte forma:

```
class nome-classe {  
}
```

- Exemplo - ex19



# Java: Operador ponto (.)

- É usado para acessar as variáveis de instância e os métodos dentro de um objeto.
- A forma geral para acessar as variáveis de instância:
  - referência-a-objeto . nome-variável
  - referência-a-objeto . nome-método

```
class Ponto{  
    int x, y;  
    public static void main( String args[ ]){  
        Ponto p = new Ponto( );  
        p.x = 10;  
        p.y = 20;  
        System.out.println("x = " + p.x + "y = " + p.y);  
    }  
}
```

⇒ Exemplo - ex20

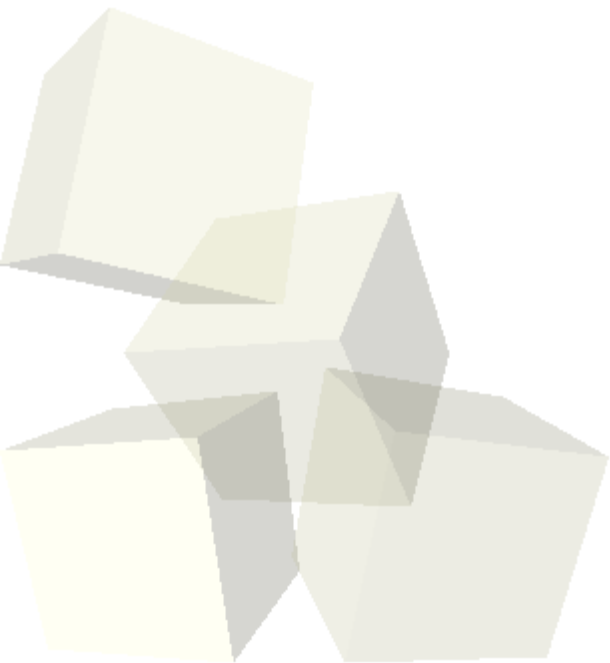




- Como seriam outras classes?
  - ♦ Celular
    - Talk, cancel, etc
  - ♦ Melhorando classe Pessoa
    - Sets e gets
- Entrada e saída:
  - ♦ Java
  - ♦ C++
- Construa programas em Java e C++ (instancie objetos e utilize funções – também conhecidas como métodos em Orientação a Objetos) para gerenciar uma Biblioteca
  - ♦ O que há em uma biblioteca? Como tais entidades se relacionam?

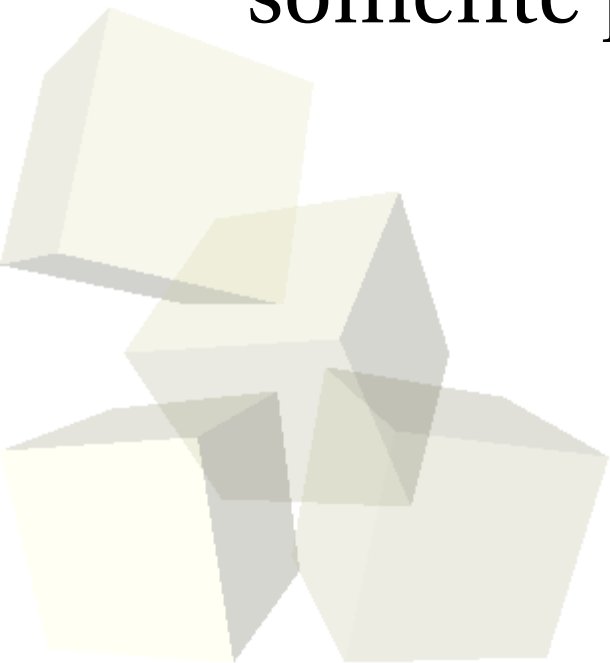


- Implementar uma classe Calculadora em C++ e Java:
  - ♦ Sqrt, log, soma, buffer de acúmulo, etc
  - ♦ Uso de pilha
- Entregar e marcar horário de apresentação com monitor
  - ♦ Questões relativas ao trabalho





- Executam ações (para manipular) sobre os dados internos (identificadores ou variáveis-membro) à classe
- Podem ser acessados por outros objetos ou somente pelo próprio objeto

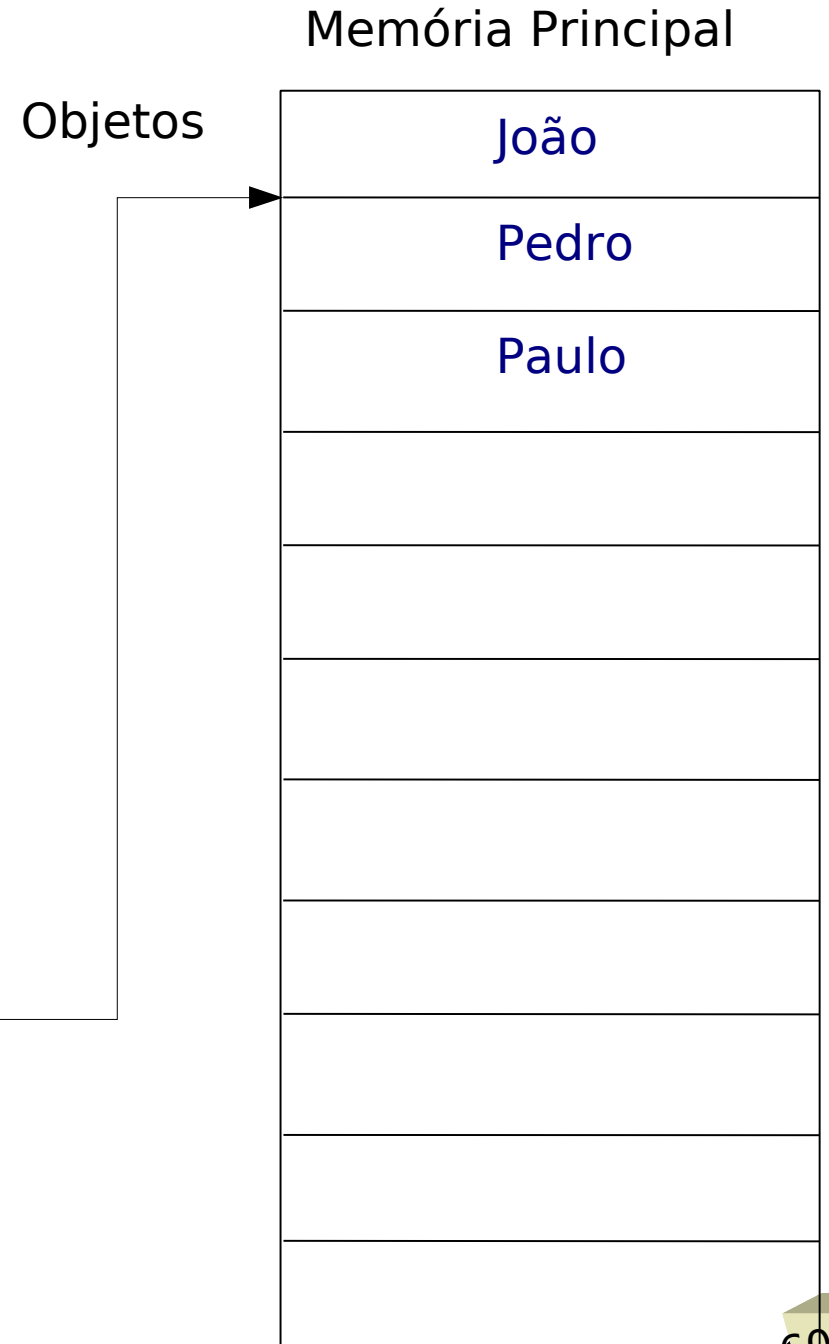




# Exemplo

## Classe

```
class Pessoa {  
    String cpf;  
    String nome;  
    String endereco;  
    public mudaEndereco(String novo) {  
        endereco = novo  
    }  
}
```





# Exemplo

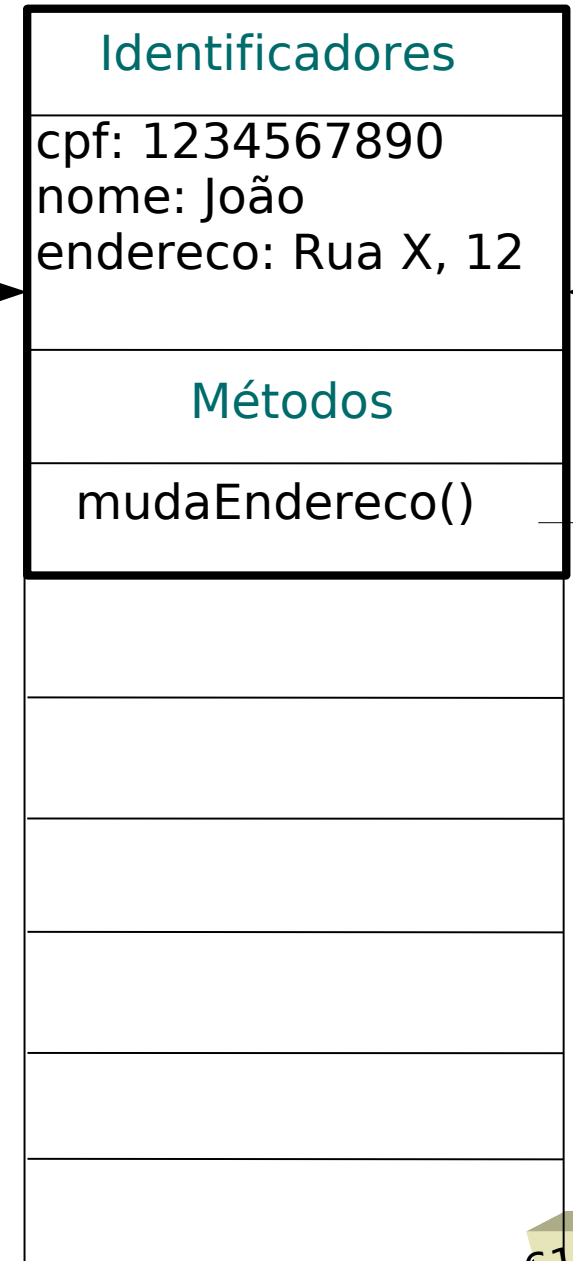
Geralmente, métodos fazem tratamentos e validação dos dados  
Ex: CPF, Cartão de Crédito, etc

## Classe

```
class Pessoa {  
    String cpf;  
    String nome;  
    String endereco;  
    public mudaEndereco(String novo) {  
        endereco = novo  
    }  
}
```

Objeto  
João

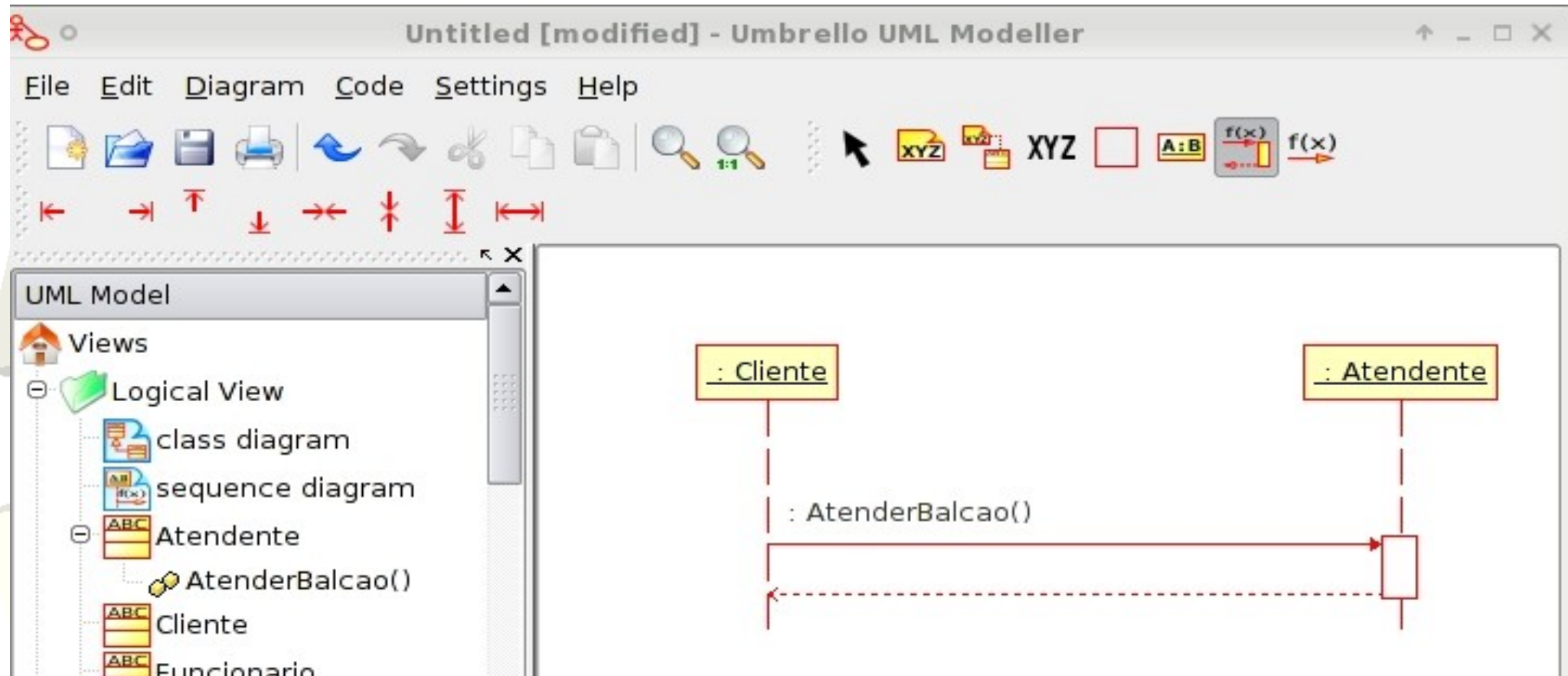
## Memória Principal





# Mensagens

- Mensagem é o ato de chamar um método
- Exemplo:
  - ♦ Um objeto A chama um método (envia uma mensagem) para um objeto B
  - ♦ Objeto B, portanto, executa uma operação e retorna algum resultado para A





## Exemplo 2

Memória Principal

Identificadores
cpf: 1234567890 nome: João endereço: Rua X, 12
Métodos
mudaEndereco()
Identificadores
placa: ABC1234 marca: Ford modelo: Fiesta
Métodos
mudaModelo()

Classe  
Pessoa

Classe  
Carro



# Exemplo 2

Memória Principal

Identificadores
cpf: 1234567890 nome: João endereço: Rua X, 12
Métodos
mudaEndereco()
listarCarros()
Identificadores
placa: ABC1234 marca: Ford modelo: ?
Métodos
obterModelo()

Mensagem



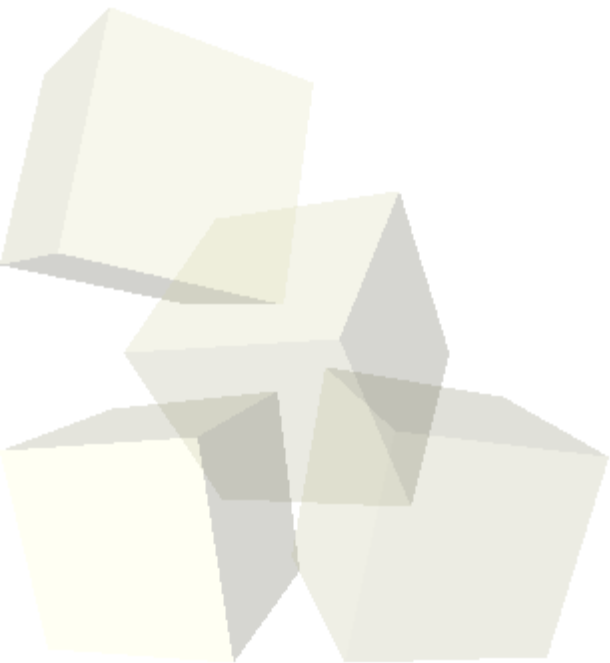
Classe  
Pessoa

Classe  
Carro



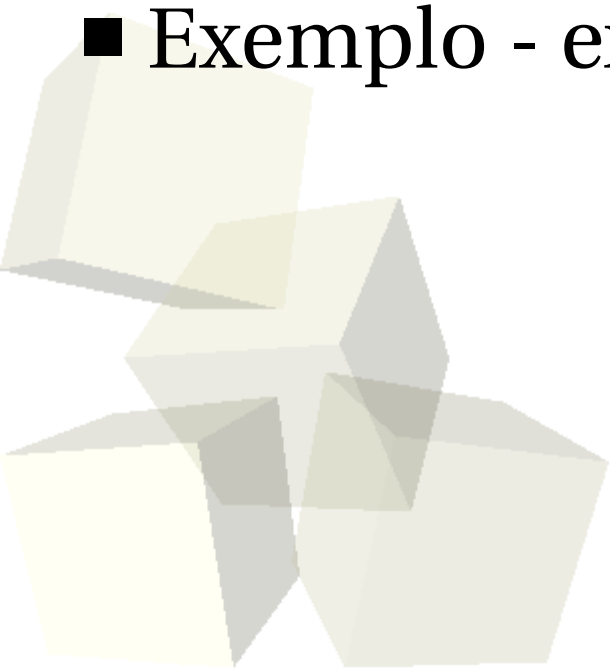


- Continue os programas em Java e C++ da Biblioteca, implementando seus métodos





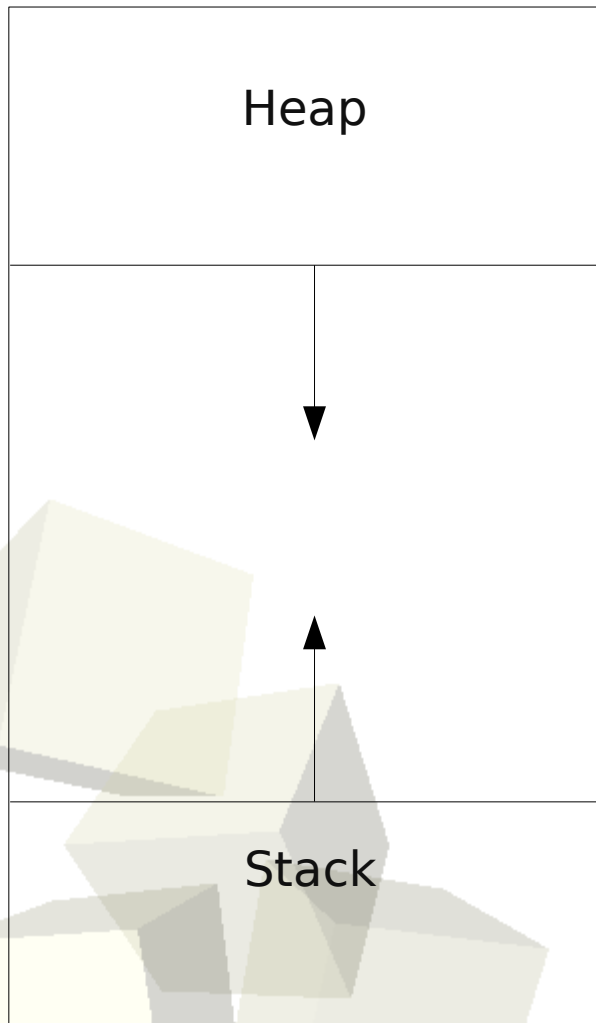
- São utilizados para Instanciar e Inicializar valores de Objetos
- Em Java e C++ têm o mesmo nome da classe e nenhum tipo de retorno explícito
- Retorna uma Instância
- Útil para definir características dos objetos ao serem instanciados
- Exemplo - ex21





- Tudo que é alocado estaticamente fica na memória Stack (Pilha)
- Tudo que é alocado dinamicamente é colocado na memória Heap
- Quando um programa termina ele não libera a Heap
  - ♦ S.O. Pode liberar, mas não é garantido
    - Linux 2.4 para frente libera
  - ♦ Portanto deve-se liberar usando um destrutor



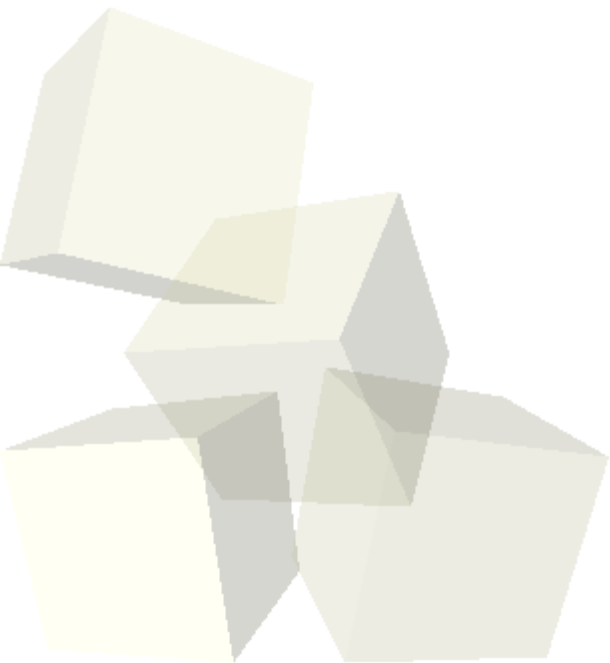


```
class X {  
    private:  
        char *nome;  
        ClasseY *obj;  
    public:  
        X() {  
            nome = (char *) malloc(sizeof(char) * 10);  
            obj = new ClasseY();  
        }  
  
        ~X() {  
            free(nome);  
            delete obj;  
        }  
}
```

## •Exemplo - ex22



- Em C++ é obrigatório o uso de destrutores
  - ♦ Principais problemas da Web -> CGI em C
- Em Java há um coletor de lixo para isso!
  - ♦ Não se usa destrutor
  - ♦ Há uma chamada para o método `finalize()` de `Object`





- Quando um objeto de uma determinada classe é criado, executa-se uma função denominada construtor da classe.
- Da mesma forma, quando um objeto é destruído, é executada a função destrutor da classe.

- Declaração:

```
class Nome
{
    public:
        Nome( parametros ); // construtor
        ~Nome( );           // destrutor
};
```



- Os operadores new e delete, além de alocar e liberar memória, chamam os construtores e destrutores do objeto
- O operador delete pode ser utilizado uma vez e somente sobre em objetos que foram alocados pelo operador new
- Estes operadores podem ser utilizados em tipos embutidos de dados (int, char, ...)
- Exemplo - ex23



- Sempre que alocar memória com `new` deve-se desalocar com `delete`
- Sempre que alocar memória com `new[]` deve-se desalocar com `delete[]` (pois é um vetor)
  - ♦ Compilador/Ambiente de execução trata a estrutura e sabe quantos elementos remover do vetor quando chama-se `delete[]`
- Quando alocamos um vetor com `malloc`, basta chamar `free`
  - ♦ Compilador/Ambiente de execução trata a estrutura e sabe quantos elementos remover do vetor





- Vantagens de new em relação ao malloc
  - ♦ new invoca o construtor implicitamente
  - ♦ new não requer um typecast tal como malloc
    - `[int *p=(int *)malloc(sizeof(int));]`
  - ♦ new não obriga desenvolvedor a definir o tamanho do objeto a ser alocado
- Vantagens de delete sobre malloc
  - ♦ delete invoca o destrutor do objeto que pode ser tratado da forma que quisermos



```
class Vetor {
    int tamanho; // número de elementos
    int *valor;   // ponteiro para inteiros
public:
    Vetor (int);   // construtor não retorna nenhum valor
    ~Vetor();      // destrutor
};

Vetor::Vetor( int s) // aloca quantidade conhecida de memória
{
    if ( s <= 0) {
        cout << "erro ! tamanho menor ou igual a zero" << endl;
    } else {
        tamanho = s;
        // aloca a memória necessária
        valor = new int[s];
    }
}

Vetor::~Vetor() // Libera a memória para o 'heap'
{
    delete valor;
}
```



```
Vetor vet1(100); // faz vet1 um vetor de 100 elementos
Vetor* func1( int n)
{
    Vetor v(n);    // aloca v no “stack”. O construtor é chamado.
                  // v só pode ser usado dentro do bloco de func1

    Vetor* p = new Vetor(n); // O construtor é chamado para
                          // alocar um vetor na memória livre.

    return p; // O ponteiro p permanece alocado e é
             // retornado pela função

    // FIM DA FUNÇÃO
    // o vetor v que está no “stack” é liberado
    // o objeto apontado por p continua na memória, mas a variável p é liberada!
}

void func2()
{
    Vetor *pontvetor = func1(20);
    delete pontvetor; // chama o destrutor e libera a memória
                     // se não chamarmos “delete” o objeto não será eliminado da
                     // memóriaHeap
}
```



## •Exemplo - ex24

```
class Demo {
    char Nome[20];
public:
    Demo ( const char *msg );
    ~Demo ();
}

Demo::Demo ( const char *msg ) {
    strcpy( Nome , msg );
    cout << "Construtor chamado por " << Nome << '\n';
}

Demo::~~Demo () {
    cout << "Destrutor chamado por " << Nome << '\n';
}

void Funcao() {
    Demo ObjetoLocalFuncao( "ObjetoLocalFuncao" );

    static Demo Objetoestatico( "ObjetoEstatico" );
    // Objeto estático: Inicializado quando a função for
    // chamada pela primeira vez e destruído no
    // encerramento do programa.

    cout << "Dentro da funcao";
}
```



```
Demo ObjetoGlobal( "ObjetoGlobal" );  
    // Objeto global : Inicializado no começo da função  
    // main e destruído no final da execução do  
    // programa.
```

```
main()  
{  
    Demo ObjetoLocalMain( "ObjetoLocalMain" );  
        // Objeto local: inicializado no começo do bloco  
        // e destruído no fim do bloco.  
  
    cout << "\nNa main, antes de chamar a funcao\n";  
  
    Funcao();  
  
    cout << "\nNa main, depois de chamar a funcao";  
}
```



# C++: Resultado de Execução

**Construtor chamado por ObjetoGlobal**  
**Construtor chamado por ObjetoLocalMain**

**Na main, antes de chamar a funcao**  
**Construtor chamado por ObjetoLocalFuncao**  
**Construtor chamado por ObjetoEstatico**  
**Dentro da funcao Destrutor chamado por ObjetoLocalFuncao**

**Na main, depois de chamar a funcao**  
**Destrutor chamado por ObjetoLocalMain**  
**Destrutor chamado por ObjetoEstatico**  
**Destrutor chamado por ObjetoGlobal**



## O operador new

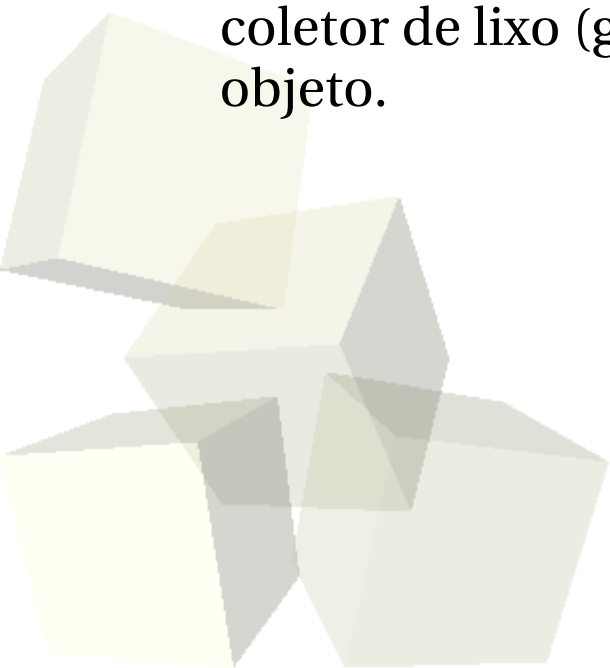
- Cria uma instância única.

*Ponto p = new Ponto ();*

*Ponto p2 = p;*

*p = null;*

Quando um objeto não é mais referenciado por nenhuma variável, o coletor de lixo (garbage collection) libera a memória usada por aquele objeto.





- Um construtor é um método que inicializa um objeto imediatamente após sua criação. Tendo o mesmo nome da classe na qual reside.
- O construtor é chamado imediatamente após a criação do objeto e antes que o operador new termine sua execução.
- Java não tem destrutor explícito

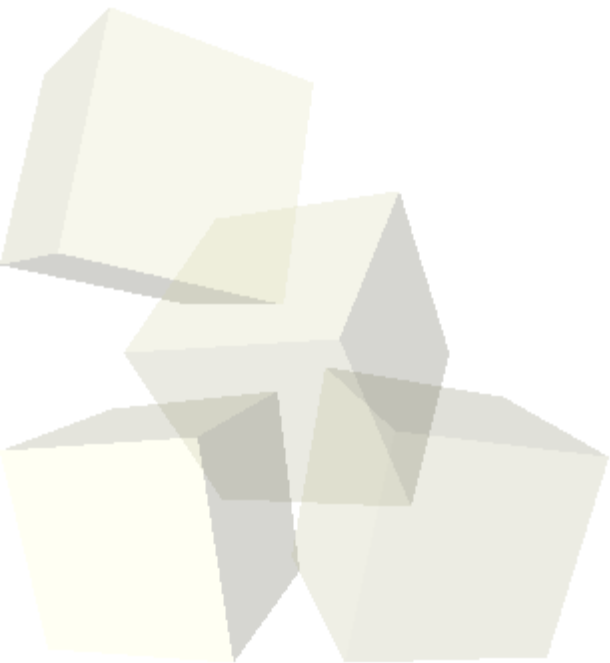
```
class teste {  
    public int x, y;  
    teste(int xa, int ya){  
        x = xa;  
        y = ya;  
    }  
}
```

```
class criateste {  
    public static void main(String args[ ])  
        teste t = new teste(15, 25);  
        System.out.println("x = " + p.x + " y = " +  
        p.y);  
    }  
}
```





- Continue os programas em Java e C++:
  - ♦ Coloque construtores e destrutores para todas as classes
- Recapitular:
  - ♦ Classes e Objetos
  - ♦ Métodos
  - ♦ Mensagens
  - ♦ Construtores e Destrutores





# Outro conceito importante (this)

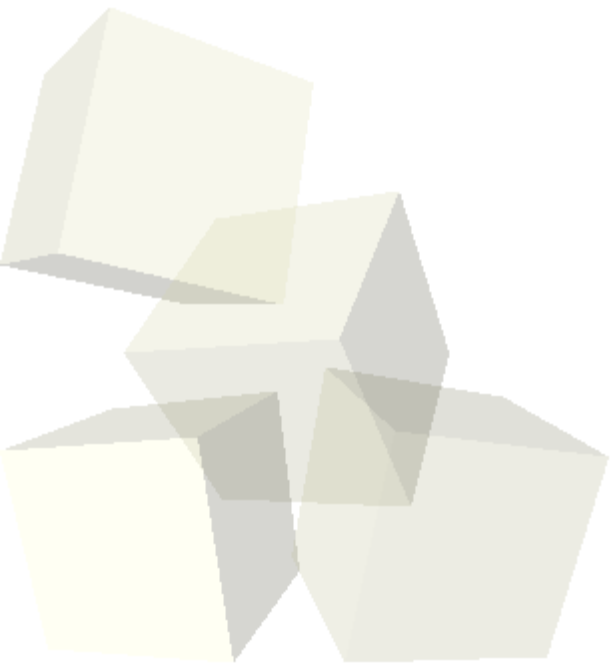
- A palavra reservada ***this*** é uma referência para o objeto da classe atual
- Essa referência pode ser utilizada em C++ ou Java
- Isso permite definir escopo de variáveis

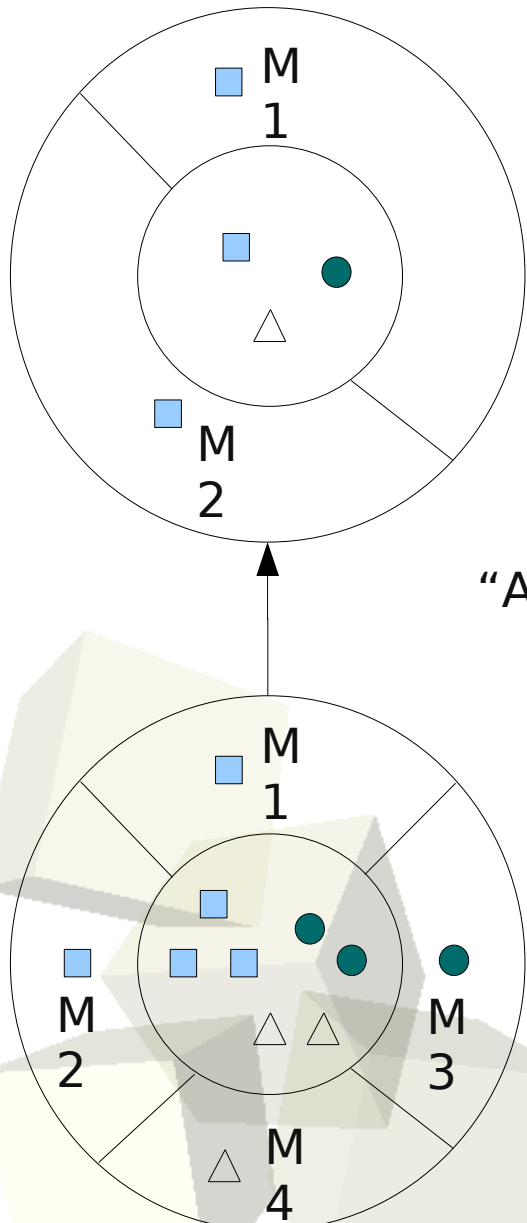
```
class teste {  
    public int x, y;  
    teste(int x, int y){  
        this.x = x;  
        this.y = y;  
    }  
}
```

⇒ Exemplo – ex31



- Pode-se construir uma nova classe baseada em uma anterior
  - ♦ Isso permite reuso de código-fonte (comportamento de outras classes)
  - ♦ Especializar classes
- Herdar as características (identificadores) e comportamento (métodos)



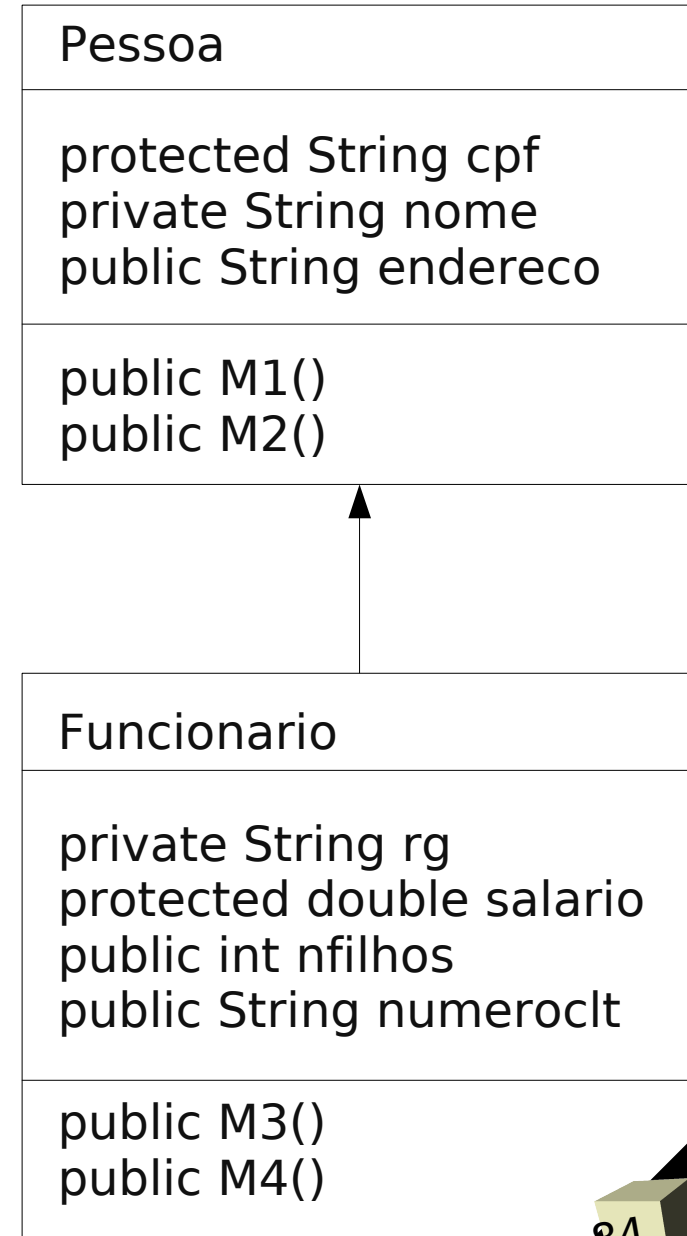


Superclasse  
(ou classe Pai)

Para herança ser válida  
deve-se fazer a pergunta:  
“A subclasse X é uma superclasse?”  
Se sim, herança correta!  
Caso contrário erro!

Subclasse  
(ou classe Filha)

**Exemplo - ex25 e ex26**





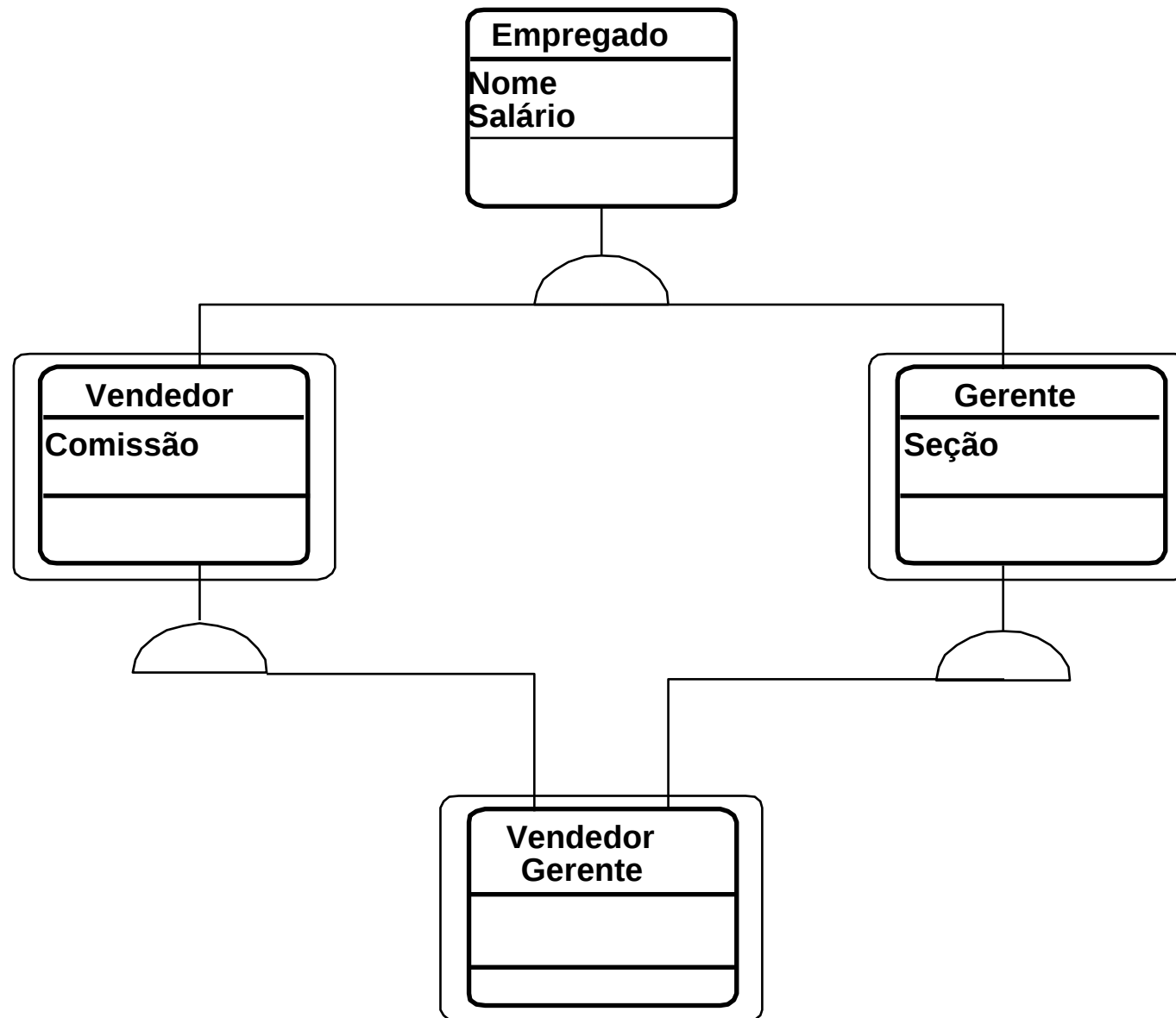
- Uma classe pode ser derivada como:
  - ♦ public, protected ou private
- Visibilidade dos membros da classe derivada:

Membro na classe base	Derivação	Membro na classe derivada
PRIVATE	PUBLIC	Não tem acesso
PROTECTED		PROTECTED
PUBLIC		PUBLIC
PRIVATE	PRIVATE	Não tem acesso
PROTECTED		PRIVATE
PUBLIC		PRIVATE
PRIVATE	PROTECTED	Não tem acesso
PROTECTED		PROTECTED
PUBLIC		PROTECTED

- Exemplo – ex27 e ex28 (chamando construtor classe pai)



# C++: Herança Múltipla





- Declaração

```
class VendedorGerente : public Vendedor, public Gerente
{
    // ...
};
```

- Tipos de classes base:

- ♦ Direta ( Vendedor e Gerente )
- ♦ Indireta ( Empregado )

- Uma mesma classe pode ser classe base indireta mais de uma vez, como no caso de Empregado. Neste caso existirão 2 cópias da classe indireta
- Para que não existam 2 cópias, deve-se declarar Empregado como classe base virtual para Vendedor e Gerente



# C++: Herança Múltipla

```
class Empregado // classe base indireta para
{
    // VendedorGerente
    char Nome[50];
    float SalarioFixo;
public:
    float GetSalario( );
};
class Vendedor : public virtual Empregado
{
    // Empregado é classe base
    // virtual para Vendedor
    float Comissao;
};
class Gerente : public virtual Empregado
{
    // Empregado é classe base
    // virtual para Empregado
    int Secao;
};
class VendedorGerente : public Vendedor, public Gerente {
};
```

- Exemplo - ex29





- Não há herança múltipla em Java
  - ♦ Somente herança simples
- Utiliza-se a palavra reservada ***extends*** para definir a herança
- Caso não definamos herança, Java herda de maneira implícita a classe `java.lang.Object`
- De maneira diferente do C++, quando utilizamos a palavra ***public*** na definição de uma classe Java caracterizamos-a como a principal do arquivo
- Exemplo – ex30 - Extender exemplo adicionando:
  - ♦ Outra classe independente e explorar a palavra reservada ***public***
  - ♦ Terceira classe herdando uma das duas primeiras

# Java: Referência para superclasse (super)

Refere-se diretamente aos construtores da superclasse.

```
class Ponto3D extends Ponto {  
    int z;  
    Ponto3D( int x, int y, int z) {  
        //nada pode ser definido antes!!!  
        super(x, y); //chama o construtor Ponto(x, y)  
        this.z = z;  
    }  
    public static void main (String args[ ]) {  
        Ponto3D p = new Ponto3D(10, 20, 30);  
        System.out.println("x = " + p.x + "y = "  
            + p.y + "z = " + p.z);  
    }  
}
```

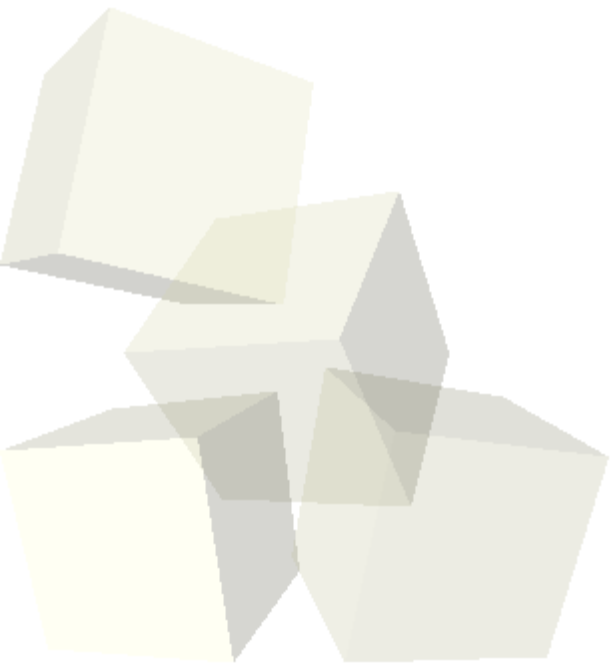
⇒ A referência super também pode ser usada para despachar diretamente métodos nomeados, bem como construtores.

⇒ Você pode tratar super como uma versão de this, que se refere à superclasse.

⇒ Assim sendo, uma declaração como super.init(x, y) significa chamar método init da superclasse na instância this.

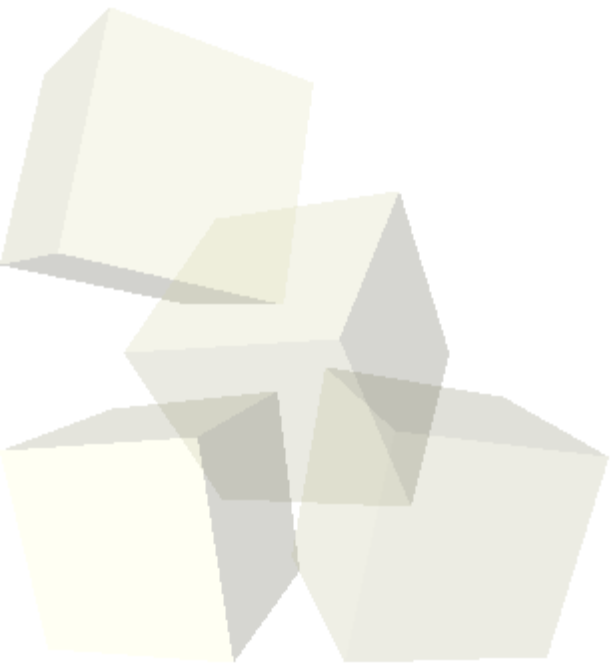


- Continue os programas em Java e C++ :
  - ♦ Pensar em possíveis heranças (e implementar)
  - ♦ Definir identificadores (variáveis-membro) e métodos para cada uma das classes



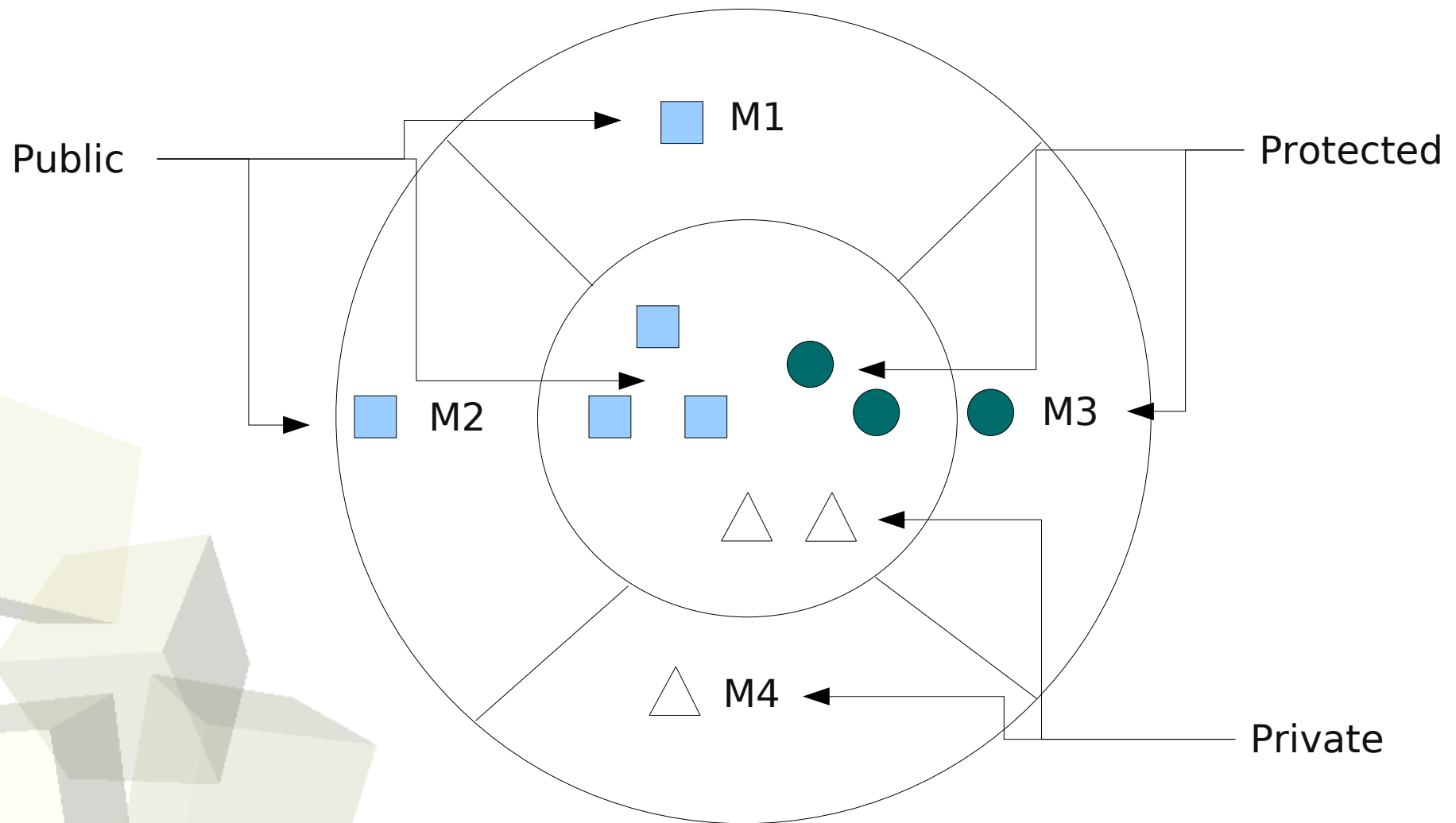


- Define o nível de acesso a identificadores e métodos de uma classe
- Podem ser:
  - ♦ Private – acessível somente a objetos da própria classe
  - ♦ Public – acessível a todos objetos
  - ♦ Protected – acessível somente a objetos oriundos de subclasses





# Encapsulamento





# Exemplo

```
class Pessoa {  
    private String cpf;  
    private String nome;  
    private String endereco;  
    public int nfilhos;  
    protected double salario;  
  
    public mudaEndereco(String novo) {  
        endereco = novo;  
    }  
}
```

Posso à partir de um objeto da classe Pessoa acessar seu próprio CPF, NOME e ENDEREÇO diretamente!!

Não posso de outro objeto QUALQUER acessar o CPF, NOME e ENDEREÇO diretamente!!!



# Exemplo

```
class Pessoa {  
    private String cpf;  
    private String nome;  
    private String endereco;  
    public int nfilhos;  
    protected double salario;  
  
    public mudaEndereco(String novo) {  
        endereco = novo;  
    }  
}
```

Posso à partir de um objeto da classe Pessoa acessar o identificador nfilhos!!!

Posso de qualquer outro objeto acessar nfilhos diretamente, pois é public!!!



```
class Pessoa {  
    private String cpf;  
    private String nome;  
    private String endereco;  
    public int nfilhos;  
    protected double salario;
```

```
    public mudaEndereco(String novo) {  
        endereco = novo;  
    }  
}
```

Posso à partir de um objeto da classe Pessoa acessar o identificador salario!!!

Não posso acessar salario diretamente à partir de qualquer outro objeto , pois é protected!!!

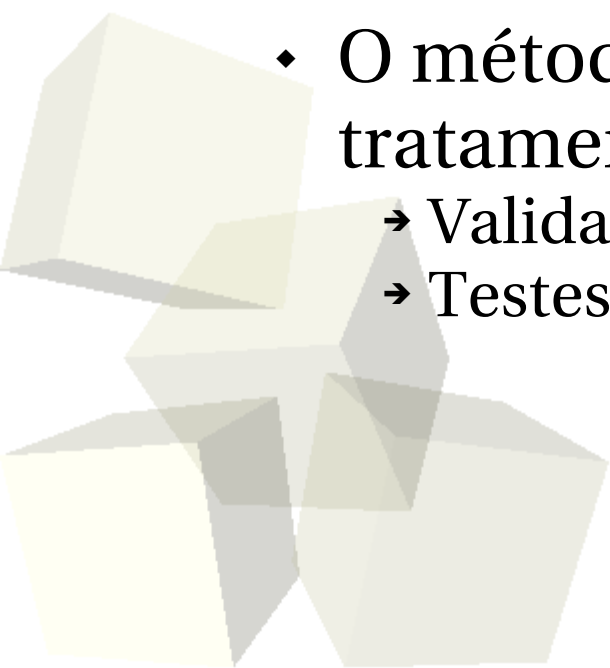
Somente objetos oriundos de classes Filhas de Pessoa podem!!!

•Exemplo - ex32



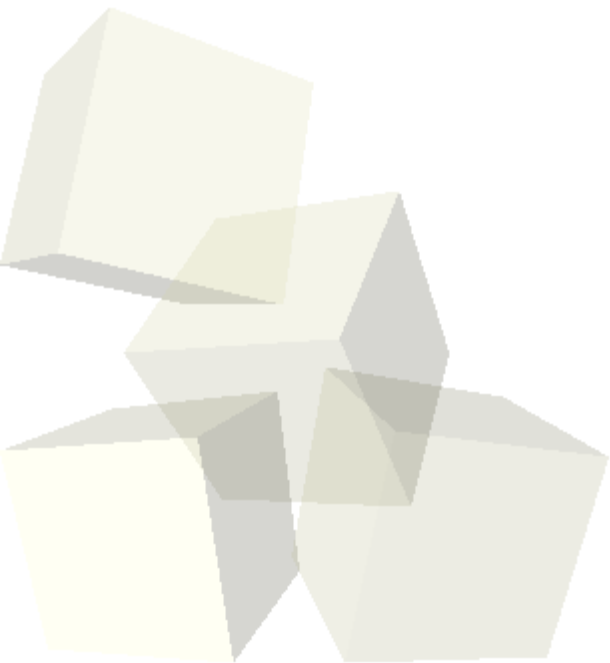


- Não se esqueçam que o mesmo conceito aplicado nos exemplos anteriores valem para métodos
- Por que criar um método **public** para acessar um atributo **private**???
  - ♦ O método controla o acesso, fazendo tratamentos para manipular o identificador
    - Validações
    - Testes, etc





- Private – acessível somente a objetos da própria classe
- Public – acessível a todos objetos
- Protected – acessível somente a objetos oriundos de subclasses
- Exemplos – ex33 e ex34





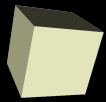
# C++: Visibilidade na classe

## ■ Membros protected

```
class No {  
    protected:          // membros “protected”. Qualquer  
                        // membro de uma classe derivada  
                        // pode acessar. Outros não !  
    No *esquerdo ;  
    No *direito ;  
    public:              // membros públicos. Acesso livre  
    virtual void print();  
};
```

## Classes Derivadas como “private”

```
class PilhaFila {  
    public:  
        void insert( elem* );  
        void append( elem* );  
        elem* remove( );  
        void leave( );  
};
```



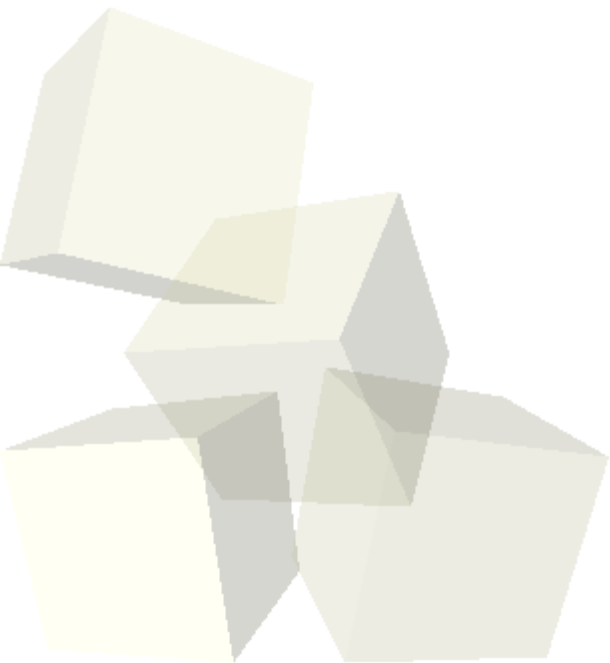
# C++: Visibilidade nas classes derivadas

```
class Pilha : private PilhaFila {  
    //...  
public:  
    PilhaFila::insert;          // torna “insert” um  
                                // membro público  
    PilhaFila::remove;         // torna “remove” um  
                                // membro público  
};  
class Pilha : private PilhaFila {  
    // ...  
public:  
    void push ( elem * ee ) // define nome usual  
        { PilhaFila::insert (ee) ; }  
    elem* pop ( )           // define nome usual  
        { return PilhaFila::remove( ); }  
};
```

**ex34.1**



- C++ adota um conceito que mistura C com C++
  - ♦ Métodos friend
- Nesse caso uma função em C pode acessar variáveis e métodos da classe em C++
- Exemplo - ex35



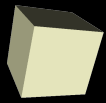


# C++: Visibilidade com Membros Friend

```
class matrix: // referência posterior !
class vect {
    int *p; int size;
    friend vect mpy ( vect &v, matrix &m); // friend permite acessar membros ocultos
public:
};

class matrix {
    int **p; int s1; int s2;
    friend vect mpy (vect &v, matrix &m); // friend permite acessar membros ocultos
public:
};

vect mpy (vect& v, matrix& m) {
    if ( v.size != m.s1) { cerr << "erro !" ; exit(1) ; }
    vect ans(m.s2) ;
    for( int j = 0; j <= m.s2; ++j) {
        ans.p[j] = 0;
        for(int k=0; k <= m.s1; ++k) ans.p[j] += v.p[k]*p[j][k] ;
    }
    return (ans);
};
```



## • Exemplo - ex36

```
int x ;  
class X {  
    int x ;  
    int f() ;  
};
```

(a) `int X :: f ( ) { return x ; } // retorna this -> x`

(b) `int X :: f ( ) { return ::x ; } // retorna x cujo  
// escopo é global.`

(c) `int X :: f ( ) { int x = 3 ;  
return x ; } // retorna x local`

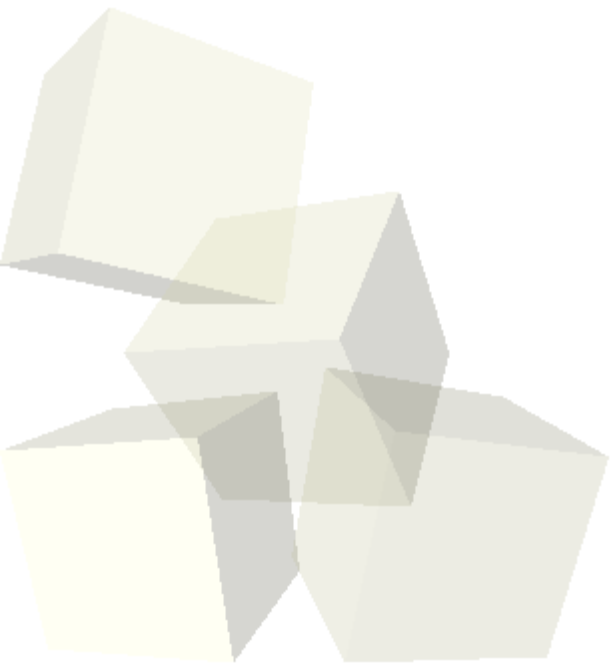
// Usando o operador de escopo pode-se ter ainda:

(d) `int X :: f ( ) { int x = 3 ;  
return X::x ; } // retorna this -> x`

(e) `int X :: f ( ) { int x = 3 ;  
return ::x ; } // retorna x cujo escopo é global.`



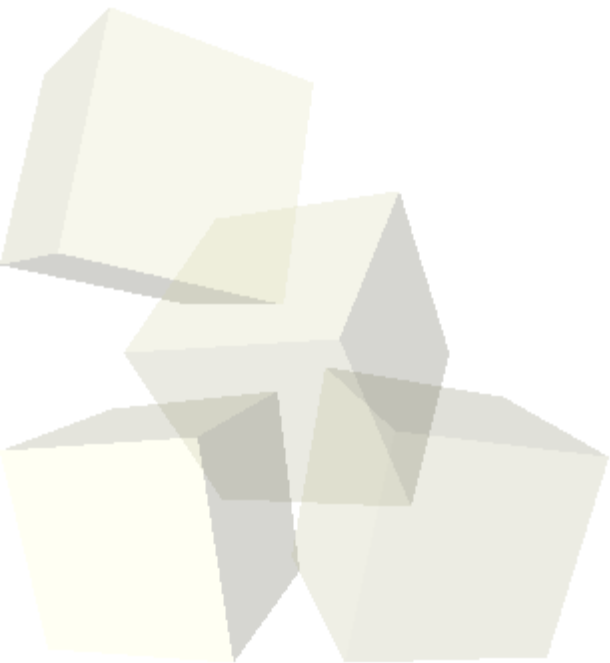
Modificador	classe	subclasse	package	todos
private	X	-	-	-
protected	X	X	X	-
public	X	X	X	X
sem modif	X	-	X	-





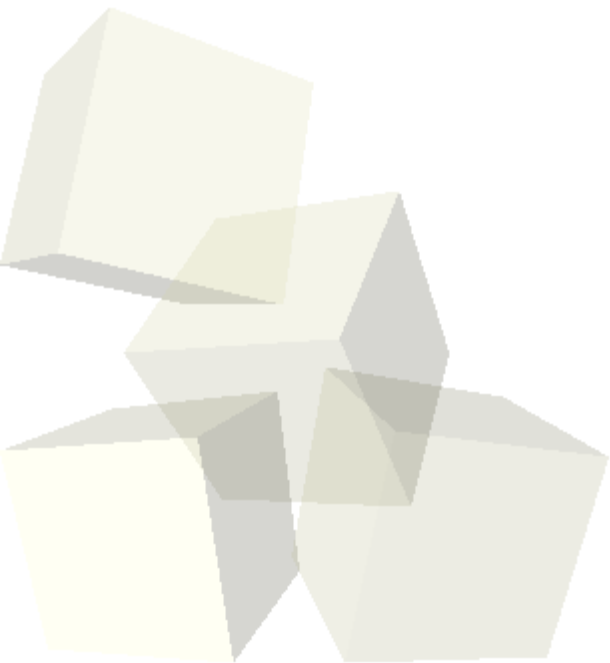


Modificador	classe	subclasse	package	todos
private	X	-	-	-
protected	X	X	X	-
public	X	X	X	X
sem modif	X	-	X	-



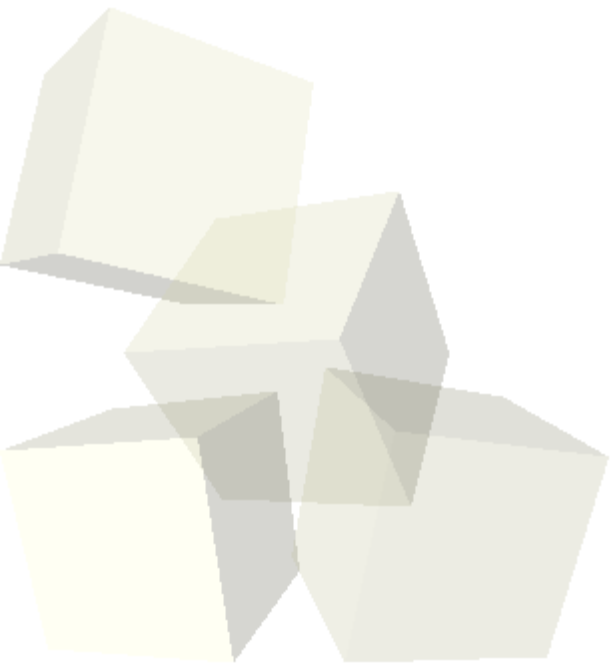


- Continue os programas em Java e C++:
  - ♦ Crie ou altere o encapsulamento (ou visibilidade) de métodos e identificadores





- Sobrecarregar um método é tê-lo duas ou mais vezes definido dentro da mesma classe (ou subclasses)
- O que muda são os parâmetros enviados para o método
  - ♦ Quantidade parâmetros
  - ♦ Ordem em que os tipos foram definidos





```
class Pessoa {  
    private String cpf;  
    private String nome;  
  
    public mudarInfo(String novocpf) {  
        cpf = novocpf;  
    }  
    public mudarInfo(String novocpf, String novonome) {  
        cpf = novocpf;  
        nome = novonome;  
    }  
}
```

•Exemplo - ex37

# Como a sobrecarga é resolvida na linguagem?

## ■ Usando Java como exemplo:

@Pessoa  
@mudarInfo@String  
@mudarInfo@String@String

Para a máquina virtual Java um método é diferente do outro, pois os tipos dos parâmetros e suas respectivas ordens de declaração são considerados parte o “nome” do método!!!



- Permite declarar várias funções com o mesmo nome e que executem diferentes conjuntos de instruções, dependendo dos tipos de dados em que estão sendo aplicados
- Auxilia no tratamento de complexidades  
Permite que conjuntos relacionados de funções sejam acessados pelo mesmo nome
- Pode-se sobrecarregar:
  - ♦ Funções
  - ♦ Operadores (C++)



- A seleção de qual função será aplicada é feita pelo compilador, aplicando-se as seguintes regras:

1) Parâmetros exatamente idênticos

2) Parâmetros iguais após a aplicação das seguintes conversões padrões:

char	=>	int
short	=>	int
int	=>	long ou double
long	=>	double
float	=>	int , double
zero	=>	long, double ou *
ponteiro	=>	void
ponteiro		ponteiro para a
p/ classe	=>	classe pública

3) Parâmetros iguais após a aplicação das conversões especificadas pelo usuário.



```
Class X {  
    public:  
    void Solicitacao(char *Mensagem , int *i)  
    {  
        cout << Mensagem;  
        cin >> *i;  
    }  
    void Solicitacao(char *Mensagem , float *f)  
    {  
        cout << Mensagem;  
        cin >> *f;  
    }  
    void Solicitacao(char *Mensagem , long *l)  
    {  
        cout << Mensagem;  
        cin >> *l;  
    }  
};
```



```
main()
{
    X *x = new X();
    int i;
    float f;
    long l;
    x->Solicitacao("\n Informe um inteiro" , &i);
    x->Solicitacao("\n Informe um float , &f);
    x->Solicitacao("\n Informe um long" , &l);
    cout << '\n' << i << f << l;
}
```

- Resultado da execução

```
Informe um inteiro: 13
Informe um float: 1.23
Informe um long: 1276L
13 1.23 1276L
```



# C++: Sobrecarga de operadores

- Não podem ser sobrecarregados

-> . ? : ,

- Forma geral:

*Tipo* NomeClasse::operator#( *parametros*)

{

// operação definida relativa a classe

}

- Valor de retorno normalmente é do mesmo tipo da classe, para permitir atribuições múltiplas:

a = b = c;

- Funcionamento

a = b + c

A função operator+ do objeto b será chamada, passando c como parâmetro. O valor retornado por esta função será passado como parâmetro para a função operator= de a

- Exemplo – ex39 e ex40

## ■ Diferença:

- ♦ Atribuição: é necessário que o elemento que receberá o valor exista:

```
vet1 = vet2; // copia vet2 para vet1
```

- ♦ Inicialização: o elemento é criado e inicializado com o valor atribuído:

```
Vetor vet2;
```

```
// ...
```

```
Vetor vet1 = vet2;
```

- Para que um objeto de uma classe possa usar a atribuição é necessário que a classe sobrecarregue o operador =
- A inicialização requer que a classe possua um construtor de cópia:

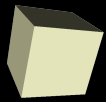
```
class Minha
{
    public:
        Minha( ); // construtor void
        Minha( Minha& ); // construtor de cópia
}
```



# C++: Exemplo de sobrecarga do operador =

```
class Vetor {  
    int * valor;  
    int tamanho;  
public:  
    // ...  
    void operator = (Vetor&); // sobrecarga do  
                                // operador de atribuição  
};  
  
Vetor::operator = ( Vetor& vet ) {  
    if (tamanho != vet.tamanho)  
        error("verifique tamanho");  
    for (int i = 0 ; i < tamanho ; i++)  
        valor[i] = vet.valor[i];  
}  
  
vet1 = vet2 // atribui vet2 a vet1,  
            // isto é copia os elementos
```

- Exemplo - ex41



# C++: Exemplo de construtor de cópia

- Exemplo - ex42

```
class Vetor {
    Vetor(int);      // cria um vetor
    Vetor(Vetor&);   // cria um vetor e copia os elementos de outro
};

Vetor::Vetor(Vetor& vet)
{
    tamanho = vet.tamanho;
    valor = new int[tamanho];
    for( int i=0 ; i<tamanho; i++ )
        valor[i] = vet.valor[i];
}

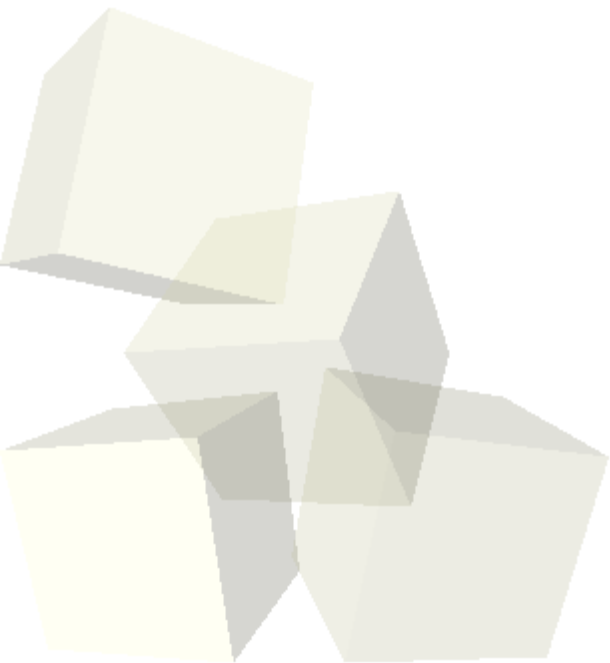
Vetor vet2 = vet1;      // faz vet2 uma cópia de vet1 usando o
                        // construtor de cópia

void fc1( Vetor);
fc1(vet2);      // Usa Vetor(Vetor&) para passar uma cópia
                // de vet2 para a função fc1

Vetor fc2(int tam) {
    Vetor valor(tam);
    return valor;      // Usa Vetor(Vetor&) para retornar uma
                        // cópia de valor
}
```



- Em Java pode-se sobrecarregar métodos somente
  - ♦ Parâmetros devem ser diferentes (considerando a ordem e o tipo)



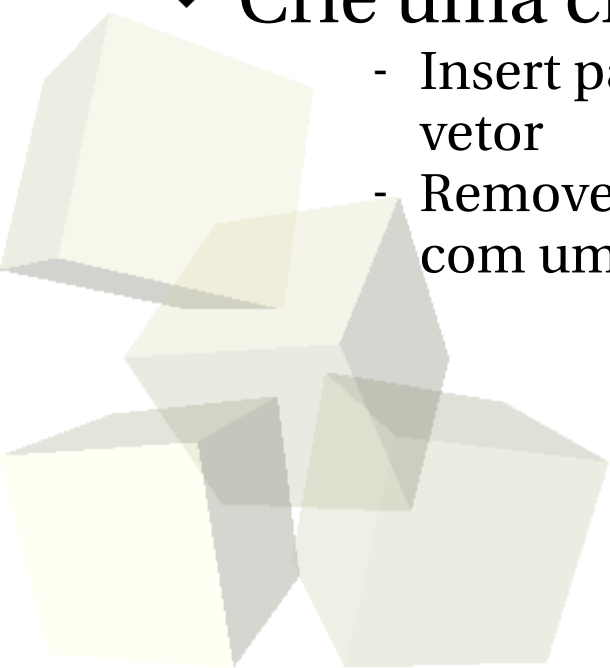


## ■ C++:

- ♦ Crie uma classe Vetor que sobrecarregue:
  - O operador + para concatenar vetores
  - O operador – para remover elementos do vetor
  - Insert para inserir um elemento no final ou em uma posição do vetor
  - Remove para remover um elemento de uma posição do vetor ou com um determinado valor

## ■ Java:

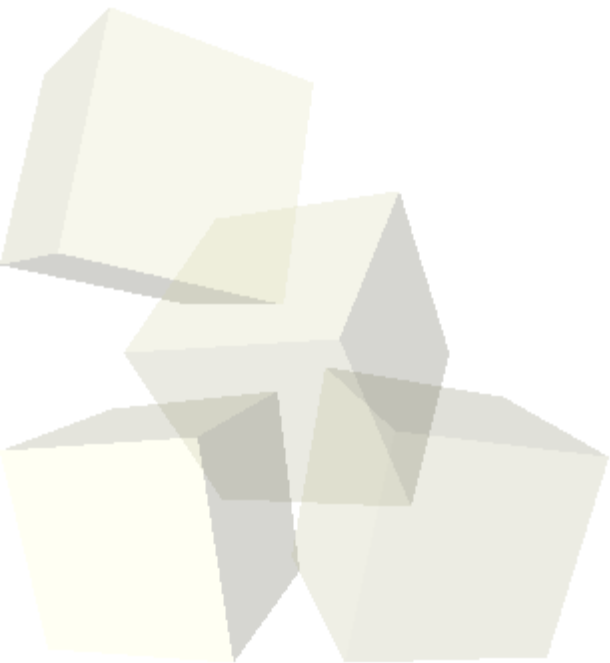
- ♦ Crie uma classe Vetor que sobrecarregue:
  - Insert para inserir um elemento no final ou em uma posição do vetor
  - Remove para remover um elemento de uma posição do vetor ou com um determinado valor







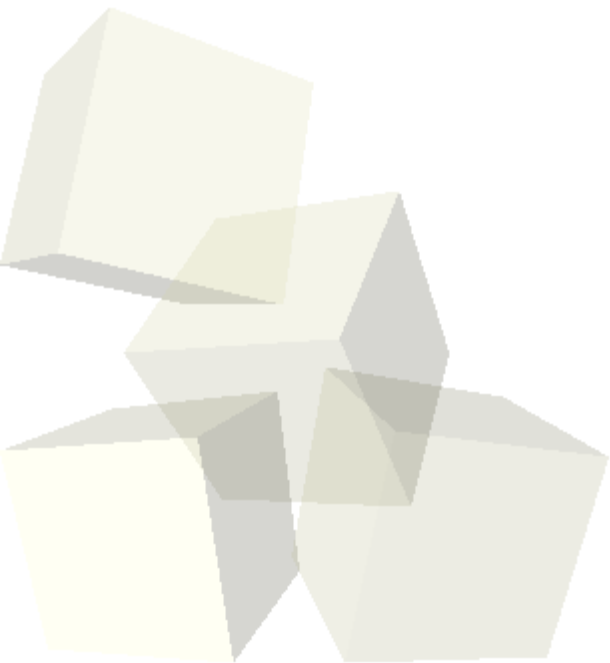
- Pode-se classificar em dois tipos:
  - ♦ Conversão de objetos filhos em pais
  - ♦ Chamadas de métodos (também conhecido como métodos virtuais ou polimorfismo de método)





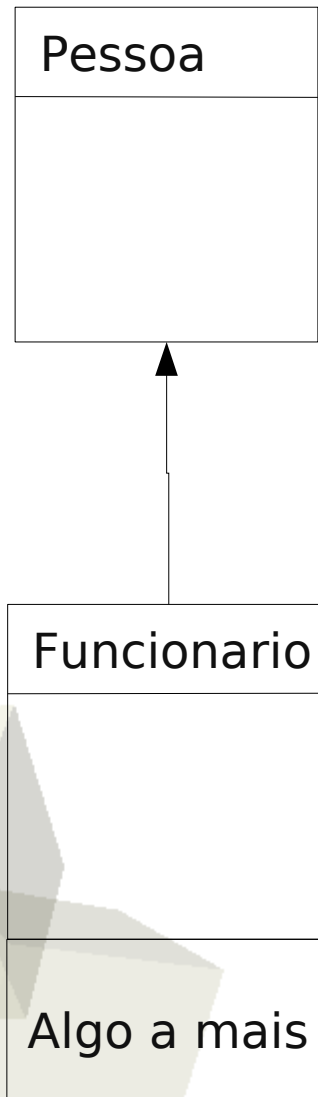
# Conversão de Objetos Filhos em Pais

- Considere uma herança entre a classe Pessoa (superclasse) e Funcionário (subclasse)
- Pode-se criar um Objeto do tipo Funcionário e convertê-lo em uma Pessoa
  - ♦ O contrário não é Possível (mas C++ deixa!)





# Conversão de Filhos em Pais



Funcionário tem o código de Pessoa + algo, portanto podemos desconsiderar a região de memória adicional de Funcionário e tratá-lo com uma pessoa

Essa é a conversão de filho em pai



# Conversão de Filhos em Pais

Tendo um objeto do tipo Funcionário:  
Funcionario f = new Funcionario();

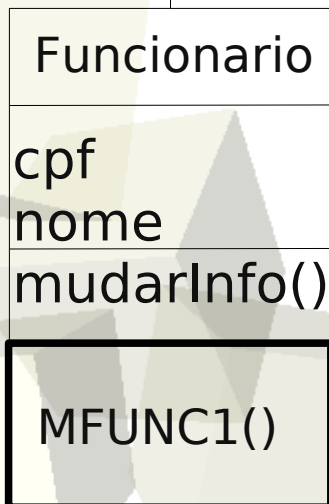
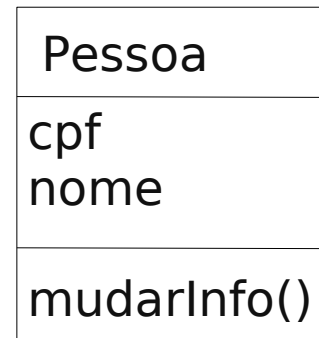
Tendo um objeto do tipo Pessoa:  
Pessoa p = new Pessoa();

Pode-se fazer:  
p = f;

Assim o funcionário será convertido em uma Pessoa!  
À partir desse momento pode-se chamar somente métodos da classe Pessoa!!!

O contrário NÃO é possível, pois se:  
f = p;

Como fazer para solucionar:  
f.MFUNC1(); //dá erro, pois essa região de memória não está alocada pelo objeto Pessoa original

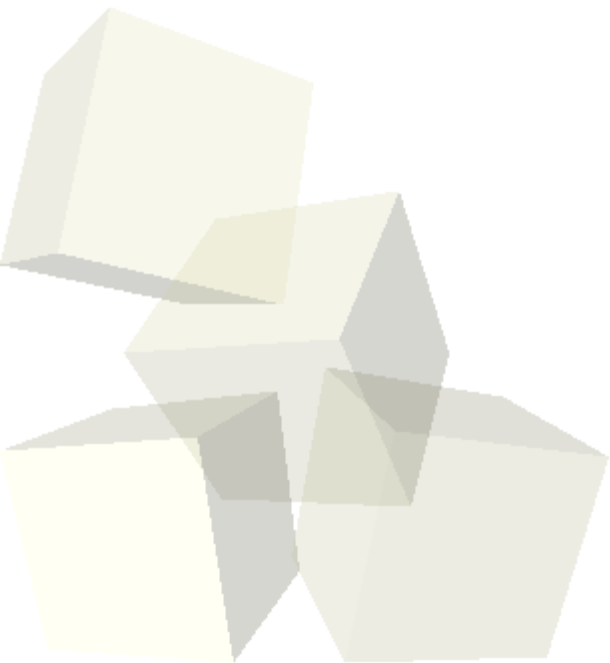


## •Exemplo - ex43



# Construindo um Vetor para a Biblioteca

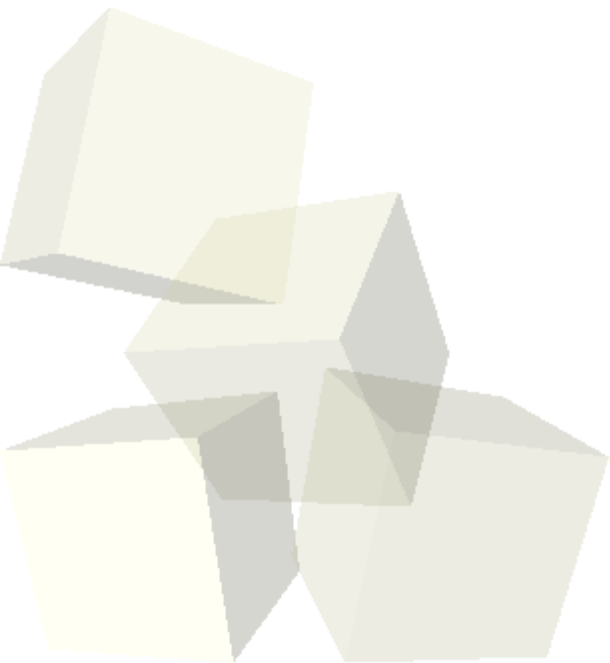
- Considere o programa em C++ que gerencia nossa biblioteca
  - ♦ vamos implementar um Vetor que armazene qualquer tipo de item





# Métodos ou Funções Virtuais

- Um método (ou função) virtual é aquele que pode ser sobrescrito em uma classe filha
- Java permite sobrescrever todos métodos
  - ♦ Todos são implicitamente virtuais
- C++ permite sobrescrever somente métodos definidos com a palavra especial **virtual**
  - ♦ Deve-se especificar de maneira explícita





# Polimorfismo de Método

## Explora o conceito de métodos virtuais

```
class Teste {  
    public m1() {  
        //código  
    }  
    public m2() {  
        m1();  
        //código  
    }  
}
```

```
class TesteFilho extends Teste {  
    public m1() {  
        //código  
    }  
}
```

Tendo:

```
Teste t1 = new Teste();  
TesteFilho tf1 = new TesteFilho();
```

t1.m2(); //chama m2 de Teste

/\*m2 chama m1, como este objeto é do tipo Teste, começa à partir dele à procurar um m1\*/

t2.m2(); //chama m2 de Teste

/\*m2 chama m1, como este objeto é do tipo TesteFilho, começa à partir dele à procurar um m1\*/

## Exemplo - ex44



# C++: Métodos ou Funções Virtuais

- Função definida na classe base e que pode ser redefinida pelas classes derivadas, por outra função que tenha o mesmo nome e parâmetros
- Define-se uma função como virtual da seguinte forma:

```
class ClasseBase
{
    virtual int Funcao( ) // permite sobrescrever o método
    { // ... }
};
```

```
class ClasseFilha
{
    Funcao( )
    {
        // redefinição de Funcao
    }
};
```

- Exemplo – ex45 e ex46





# C++: Visibilidade

- Usa-se o corpo da função virtual na classe derivada sempre que for possível
- Exemplo

```
class B {  
    public:  
        virtual char f()    { return 'B' } ;  
        char g()           { return 'B' } ;  
        char testeF ()     { return f() ; } ;  
        char testeG()      { return g () ; } ;  
};
```

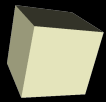
```
class D : public B {  
    public:  
        char f() { return 'D' ] ;  
        char g() { return 'D' } ;  
};
```

B :: f e B :: g retornam 'B' ; e  
D :: f e D :: g retornam 'D'.

Caso se tenha uma instância: D d ; tem-se que :  
d.testeF () ; chama D :: f e retorna 'D' ; e  
d.testeG() ; chama B :: g e retorna 'B'.

## • Exemplo - ex47

- Uma função virtual pura é aquela cujo comportamento só será definido pelas classes derivadas
- Declaração:  
class G  
{  
    virtual void Funcao( ) = 0;  
};
- Uma classe que possua alguma função virtual pura é chamada de abstrata e não se pode declarar objetos desta classe.
- Exemplo - ex48



# C++: Funções virtuais puras e classes abstratas

```
class Pai    // classe abstrata pois possui uma
{
    // função virtual pura

    virtual void Funcao( ) = 0; // função virtual
                                // pura

    // ...
};
class Filho : public Pai
{
    void Funcao( )    // Filho define Funcao
    {
        // Pode-se ter objetos desta
        // ...        // classe
    }
};
class Outra : public Pai    // Outra é uma classe
{
    // abstrata pois não
    // define Funcao

    // ...
}
```

# C++: Funções virtuais puras e classes abstratas

```
main( )  
{  
    Pai *PonteiroPai;    // OK, pode-se ter  
                        // ponteiros para  
                        // classes abstratas  
  
    Outra O;            // Erro, Outra é uma classe  
                        // abstrata e não se pode ter  
                        // objetos desta classe  
  
    Filho F;           // OK, Filho não é abstrata  
  
    PonteiroPai = new Filho; // OK, objeto da  
                        // classe Filho  
    delete PonteiroPai;  
  
    PonteiroPai = new Pai; // Erro, objeto de  
                        // classe abstrata  
}
```

```
class Pai {  
    void metodo1() {  
        //codigo  
    }  
    void metodo2() {  
        metodo1();  
        //codigo  
    }  
}  
  
class Filho extends Pai {  
    void metodo1() {  
        //codigo  
    }  
}  
  
...  
Pai p = new Pai();  
Filho f = new Filho();  
p.metodo2(); // chama metodo1 de Pai  
f.metodo2(); // chama metodo1 de Filho  
//Busca método à partir do tipo do objeto atual
```

Podemos converter um objeto da classe Filho para um da classe Pai:

```
Filho f = new Filho();  
Pai p = new Pai();
```

```
p = f; //OK
```

O contrário não funciona (seguindo corretamente OO), pois restaria região de memória sem uso para objeto do tipo Pai

C++ permite isso!

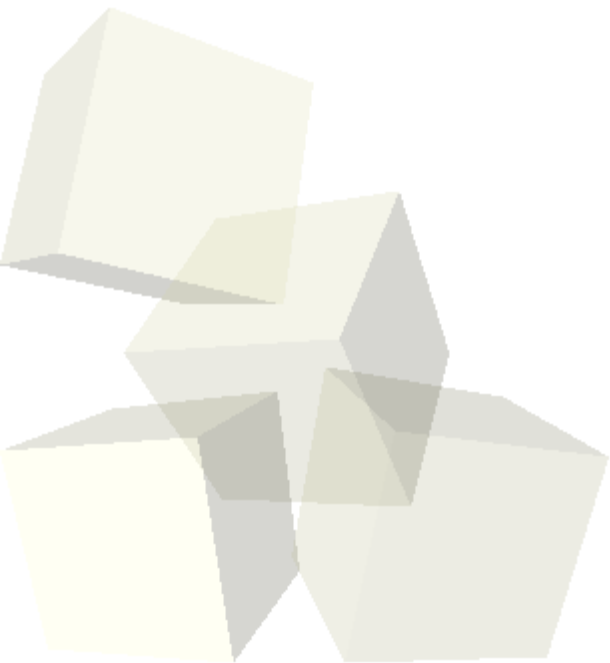


- ⇒ Classe que declara a estrutura de uma abstração, sem fornecer completamente a implementação de cada método.
- ⇒ A classe que conter o método declarado **abstract** deve ser declarada **abstract**. Essas classes não podem ser instanciadas com o operador **new**, pois sua implementação não está definida.

```
abstract class A{  
    abstract void call( );  
    void callA( ){  
        System.out.println("Dentro de A");}  
}  
class B extends A {  
    void call( ){  
        System.out.println("dentro de B"); }  
}  
class Principal {  
    public static void main(String args[ ]){  
        A a = new B( ); a.call( ); a.callA( );  
    }  
}
```



- Fazer um estudo das implementações de Biblioteca em C++ e Java:
  - ♦ Discutir:
    - Polimorfismo (1- conversão de filho em pai e 2- polimorfismo de método) seria útil?
      - Relembrar vetor que utiliza polimorfismo para converter filho em pai!
      - o que pode ser implementado utilizando polimorfismo?
    - Alguma classe poderia em abstrata? Qual(is) e por quê?





# Java: Interfaces

- O objetivo é explorar melhor os conceitos de classes abstratas, mas permitindo algo similar à herança múltipla de C++
  - ♦ Em Java pode-se herdar de somente uma classe
  - ♦ Mas pode-se implementar múltiplas interfaces
- Protocolo de Comportamento
- É uma especificação explícita de um conjunto de métodos, que pode ser implementada por uma classe sem nenhuma implementação real associada à especificação.
- As interfaces de java são criadas para suportar a resolução dinâmicas de métodos em tempo de execução. Para que um método seja chamado de uma classe para outra, ambas as classes precisam estar presentes em tempo de compilação para que o compilador java possa checar se as assinaturas de método são compatíveis.





- Classes podem implementar qualquer número de interfaces.
- Para implementar uma interface, tudo de que uma classe precisa é de uma implementação de um conjunto completo de métodos de interface.
- Esses métodos devem corresponder às assinaturas de método e interface.
- As interfaces vivem em um hierarquia diferente das classes, portanto é possível que várias classes, que não estejam relacionadas em termos de hierarquia de classes, implementem a mesma interface.



⇒ **A declaração interface:** A forma geral de uma interface é:

```
interface nome{  
    tipo-retornado nome-método1 (lista-parametros);  
    tipo nome-var-final = valor;  
}
```

Onde nome é qualquer identificador. Os métodos declarados não tem declaração de corpo. Todos os métodos que são interfaces de implementação precisam ser declarados como *public*. As variáveis podem ser declaradas dentro das declarações de interface e são implicitamente *final*. Por exemplo:

```
interface exemplo{  
    void exemplo(int param);  
}
```



## ⇒ A declaração **implements**

```
class nomeclasse [extends superclasse] [implements interface0 [,interface1...]] {  
    corpo-da-classe  
}
```

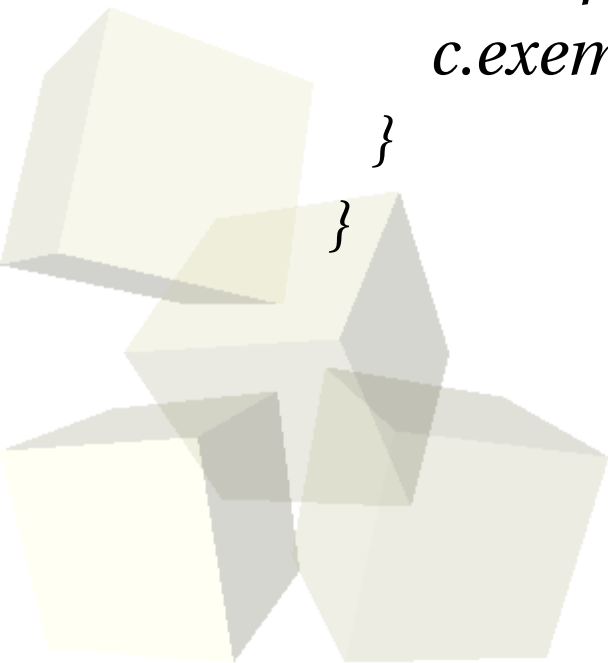
Este exemplo implementa a interface do slide anterior:

```
class client implements exemplo{  
    void exemplo(int p) {  
        System.out.println("exemplo chamada com " + p);  
    }  
}
```



- Você pode declarar variáveis como referências a objetos, que usam uma interface em vez de uma classe como tipo.

```
class TestIface {  
    public static void main(String args[ ]){  
        exemplo c = new client( );  
        c.exemplo(22);  
    }  
}
```



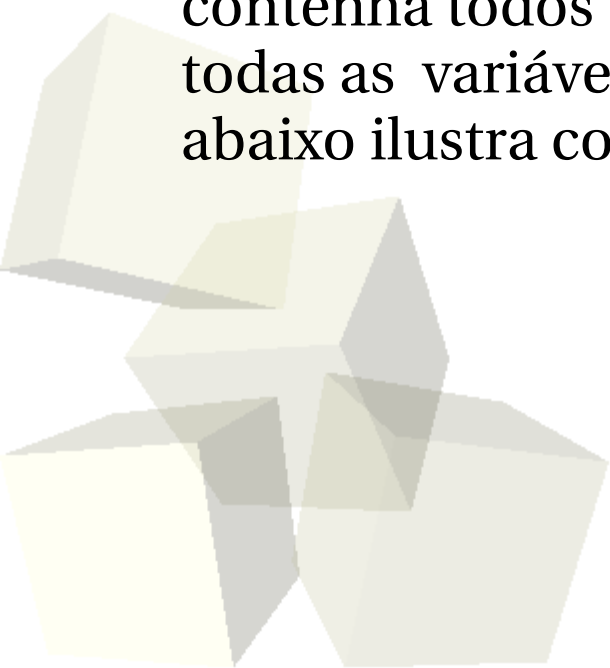


⇒ **Permitem que Classes de diferentes Hierarquias implementem uma mesma interface**

⇒ **Permite que sejam submetidas como parâmetro para uma mesma função!**

⇒ **Variáveis em Interfaces**

Você pode usar as interfaces para importar constantes utilizadas por várias classes, sendo necessário que defina uma interface que contenha todos os nomes e valores. Ao implementar esta interface todas as variáveis estarão no escopo como constantes. O exemplo abaixo ilustra como fica essas interfaces:





# Java: Interfaces

```
import java.util.Random;
interface ConstComp {
    int NAO = 0;
    int SIM = 1;
    int TALVEZ = 2;
}
Class Questao implements ConstComp {
    Random rand = new Random();
    int pergunta () {
        int prob = (int) (100 *
            rand.nextDouble ());
        if (prob < 30)
            return NAO;
        else if (prob < 60)
            return TALVEZ;
        else if (prob < 75)
            return SIM;
        }
    }
```

```
class Pergunte implements ConstComp {
    static void resposta(int result) {
        switch (result) {
            case NÃO:
                System.out.println("Não");
                break;
            case TALVEZ:
                System.out.println("Talvez");
                break;
            case SIM:
                System.out.println("Sim");
                break;}
    }
    public static void main(String arg[ ]) {
        Questao q = new Questao();
        resposta(q.pergunta());
        resposta(q.pergunta());
        resposta(q.pergunta());
        resposta(q.pergunta());}
}
```



- Considere um software de Locadora:
  - ♦ Classe DVD
  - ♦ Classe CD
  - ♦ Classe Fita
- Como poderiam estar relacionadas?
  - ♦ Pode-se criar uma superclasse Item tal como na Biblioteca ou pode-se criar uma interface
  - ♦ Mas e se múltiplas hierarquias já existissem?
    - A interface seria a melhor escolha, pois não faria sentido forçar a hierarquia entre classes distintas onde a questão “a subclasse é uma superclasse?” não fosse positiva!
- Vamos implementar esse sistema em Java utilizando interface e com a classe Vetor aceitando elementos da interface:
  - ♦ Listando todos os elementos



## ■ Em C++:

- ♦ Pode-se criar variáveis únicas para todas as instâncias de uma mesma classe
- ♦ Exemplo – ex49

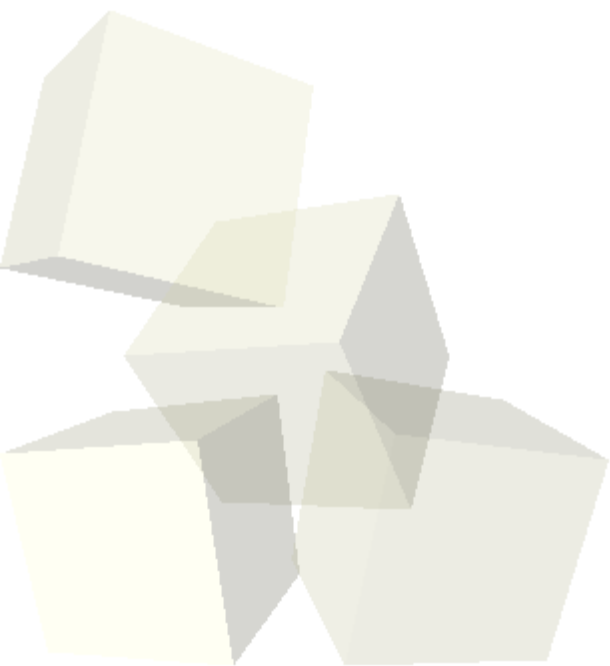
## ■ Em Java:

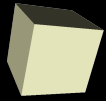
- ♦ Pode-se criar variáveis únicas para todas as instâncias de uma mesma classe
- ♦ Métodos que são usados fora do contexto de qualquer instância, são declarados como métodos **static**
  - Não pertencem à classe
  - Não podem acessar variáveis-membro, nem métodos da classe
- ♦ Estes métodos refere-se a outros métodos **static** e não podem se referir a `this` ou `super`.
- ♦ Exemplo – ex50 e ex 51



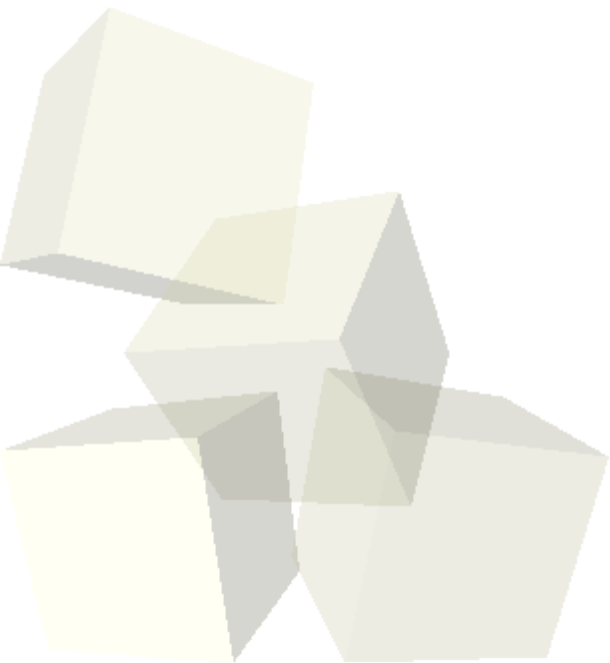


# Tratamento de Exceções





# Acesso a Bancos de Dados





# UML (Unified Modeling Language)



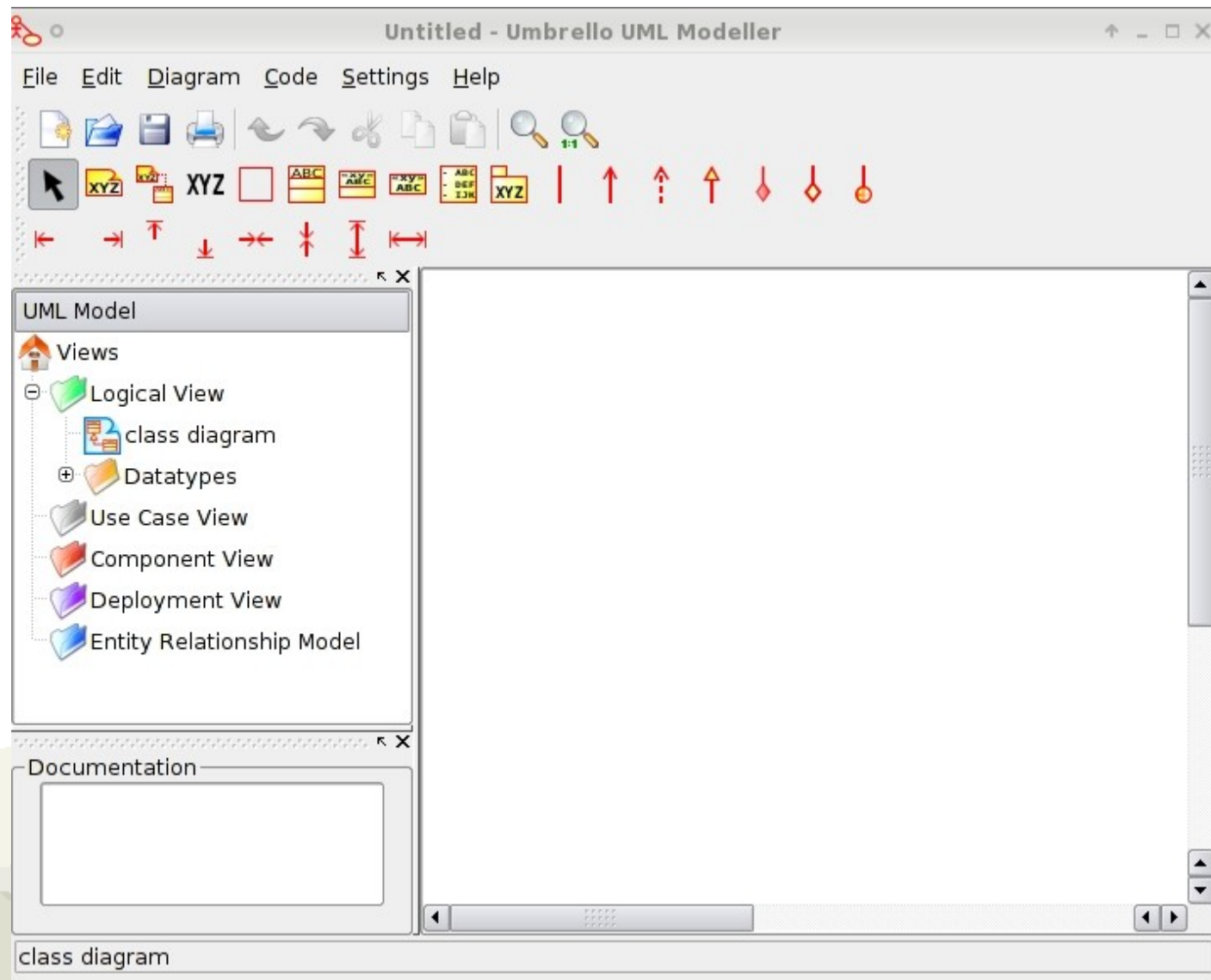


## ■ Como modelar um sistema?

- ♦ O que o sistema deve fazer?
- ♦ Observar “QUEM” faz “O QUE” no sistema (detectar entidades envolvidas)
- ♦ Começar modelagem
  - Caso de Uso
  - Diagrama de Classes
  - Diagrama de Seqüência

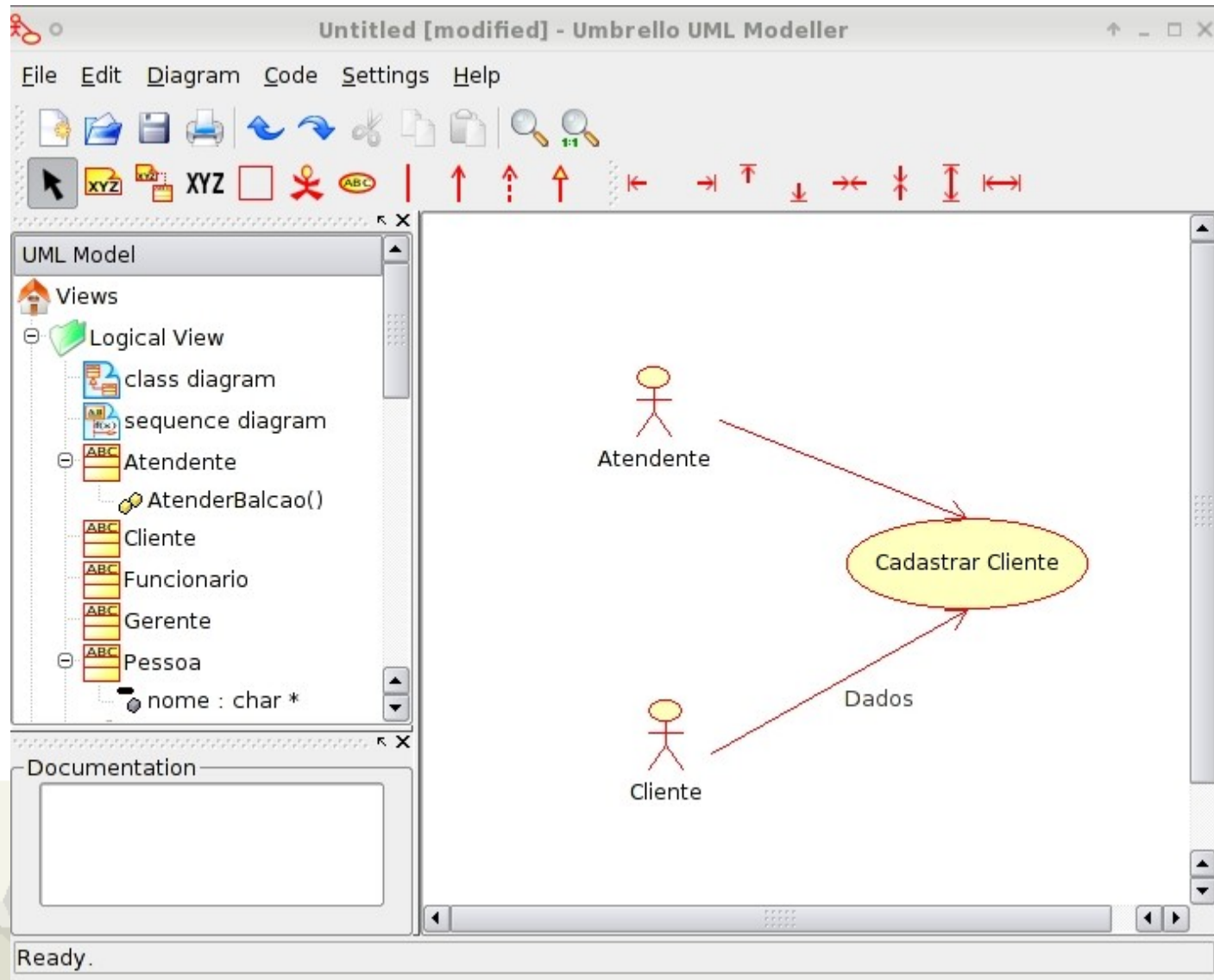
## ■ Ferramentas:

- ♦ ArgoUML (licença BSD)
- ♦ Umbrello (licença GNU/GPL)
- ♦ Rational Rose (proprietário)
- ♦ etc (pode-se obter algumas delas em [www.freshmeat.net](http://www.freshmeat.net))





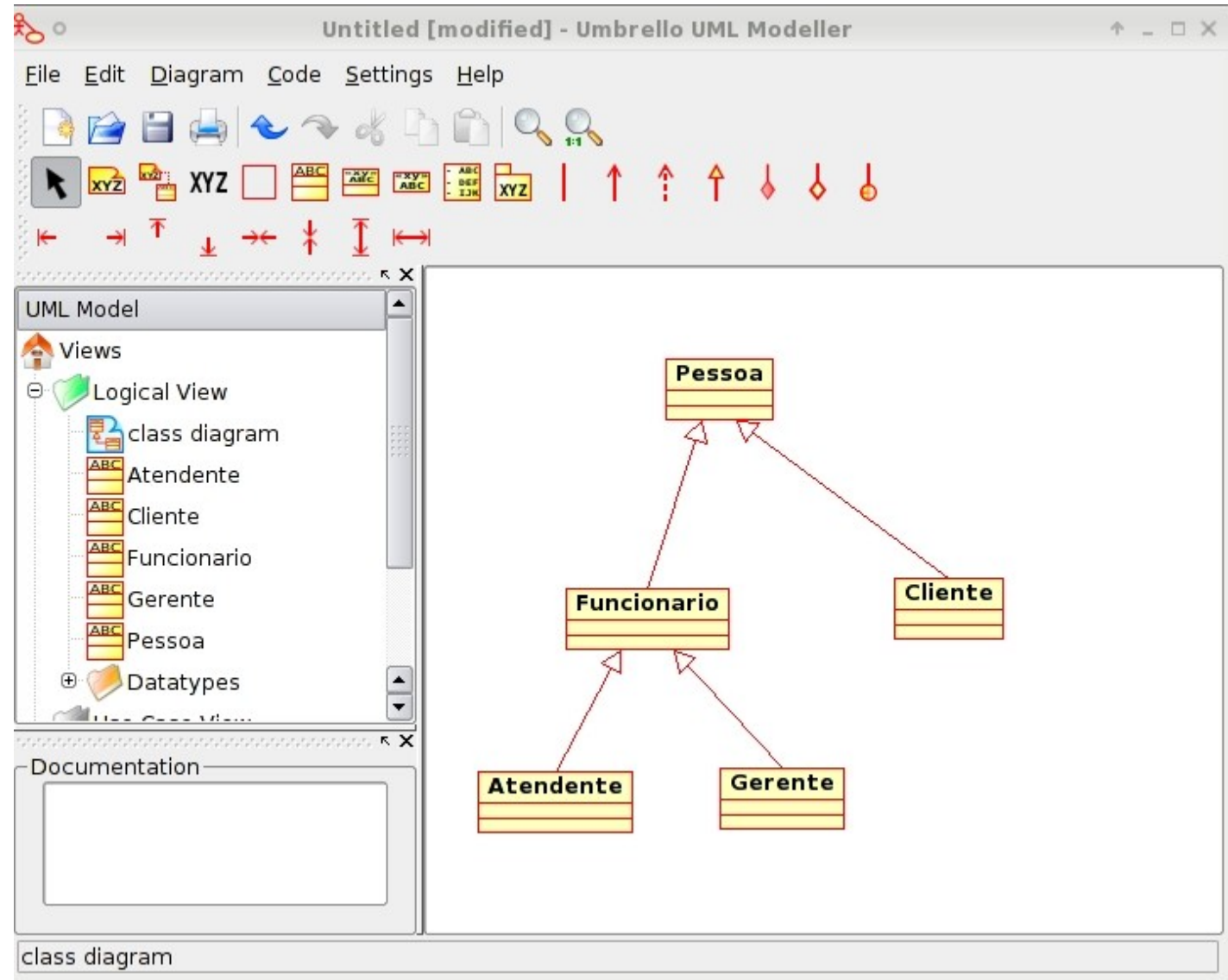
# Diagrama de Caso de Uso





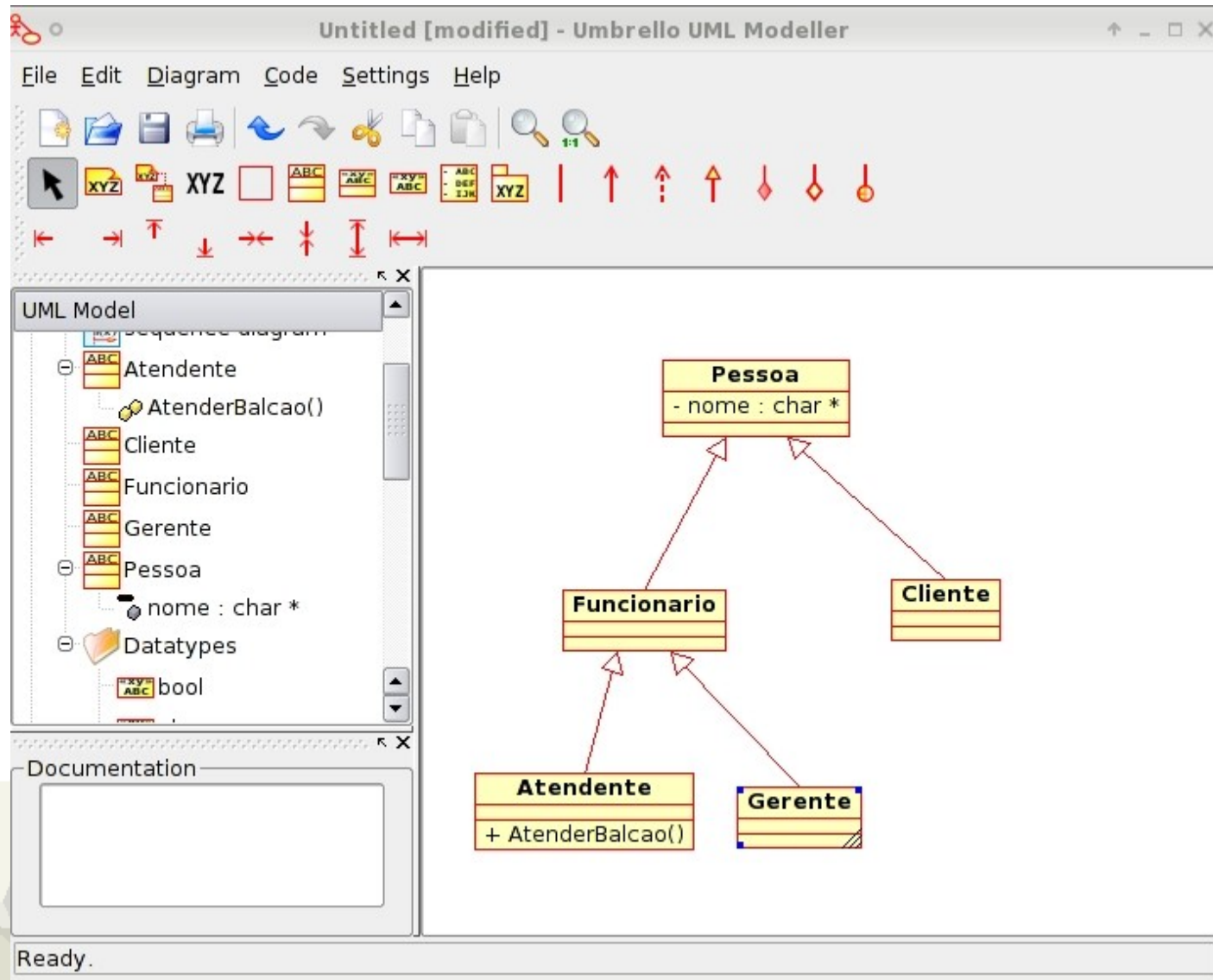
# Diagrama de Classes

- Herança
- Agregação
- Composição





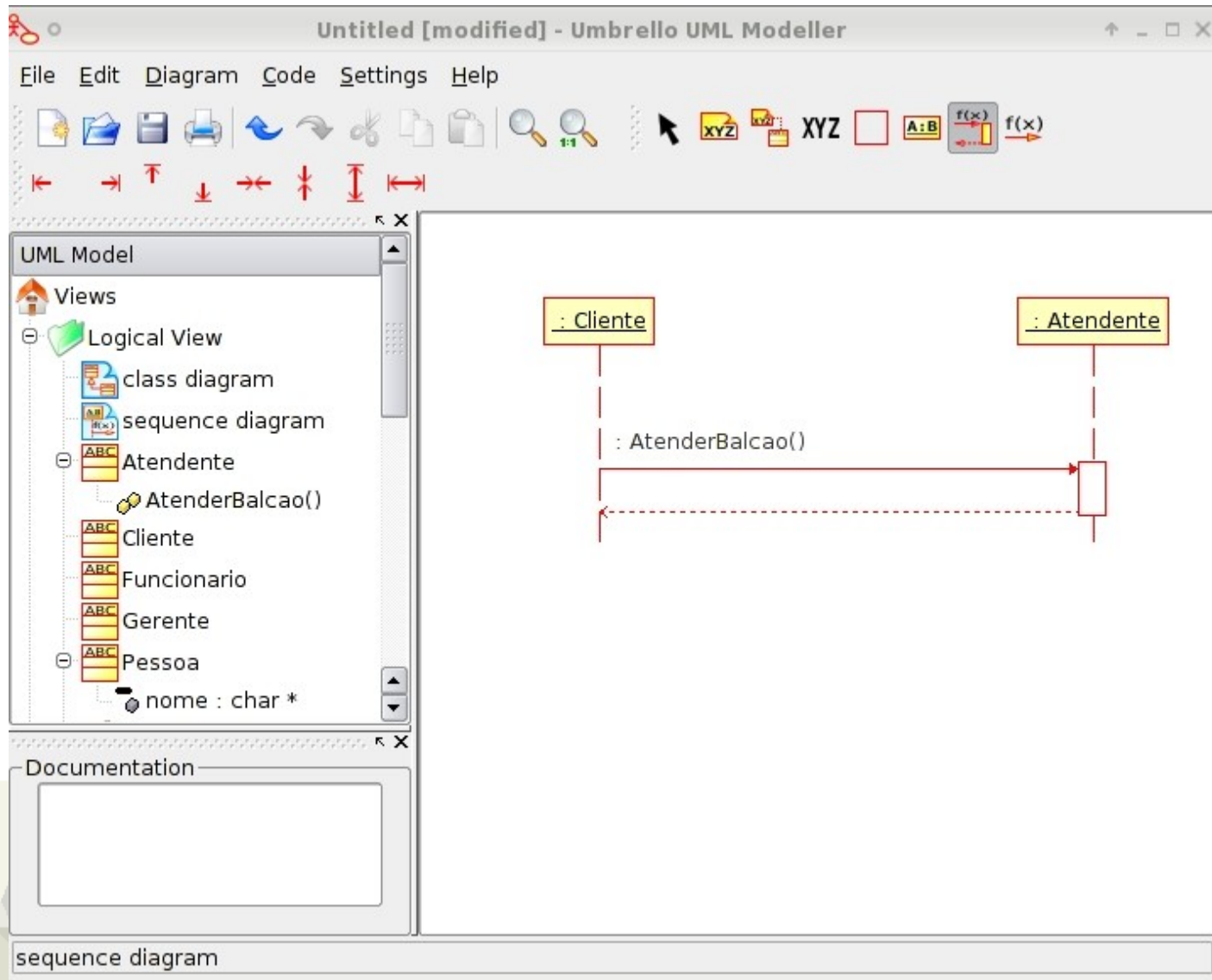
# Diagrama de Classes







# Diagrama de Seqüência





- Uma empresa de locação de veículos deseja construir seu sistema
  - ♦ Ela aluga carros de 3 diferentes categorias:
    - Econômico
    - Pickups
    - Luxo
  - ♦ E também aluga caminhões de 2 tipos:
    - Pequeno
    - Médio porte
- Projete o sistema utilizando os diagramas de (utilize essa ordem):
  - ♦ Caso de Uso
    - Quais os atores e funcionalidades do sistema?
  - ♦ Classes
  - ♦ Seqüência



- Uma empresa deseja criar um sistema de leilão de produtos via WEB
  - ♦ Produtos são colocados no leilão por um período definido pelo usuário
  - ♦ Usuários são cadastrados no sistema
  - ♦ Compradores também pode vender produtos
  - ♦ Leilão sempre manter o maior preço oferecido pelo produto, quando e por quem foi oferecido
  - ♦ A empresa deseja uma listagem de produtos vendidos por dia, por mês e por ano
  - ♦ Cobra-se um percentual por venda de produto
    - A empresa também quer uma listagem por venda, contendo o valor percentual
  - ♦ Cada categoria de produto tem um percentual distinto

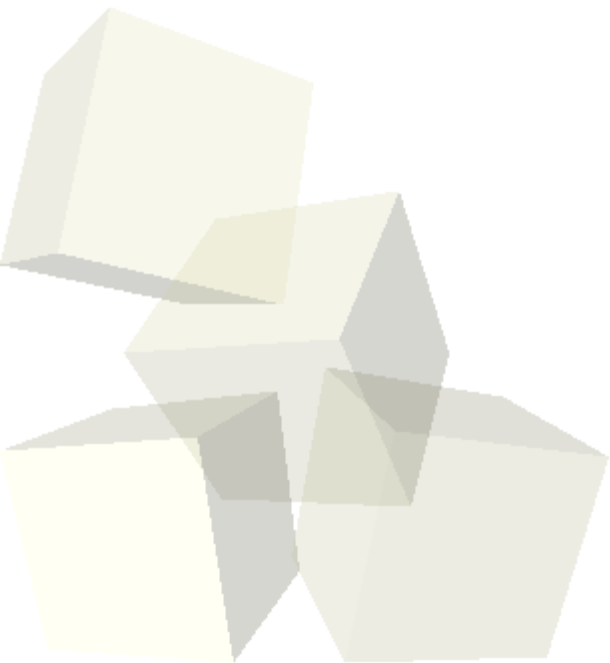


- Um banco deseja montar seu sistema de online banking (WEB)
  - ♦ Clientes poderão:
    - Pagar contas
    - Transferir valores entre contas do mesmo banco
    - Visualizar saldo e extrato da conta corrente e poupança
  - ♦ Banco define:
    - Percentual de rendimento da poupança
    - Visualiza listagem de valores de contas corrente e poupança por cliente em uma data definida



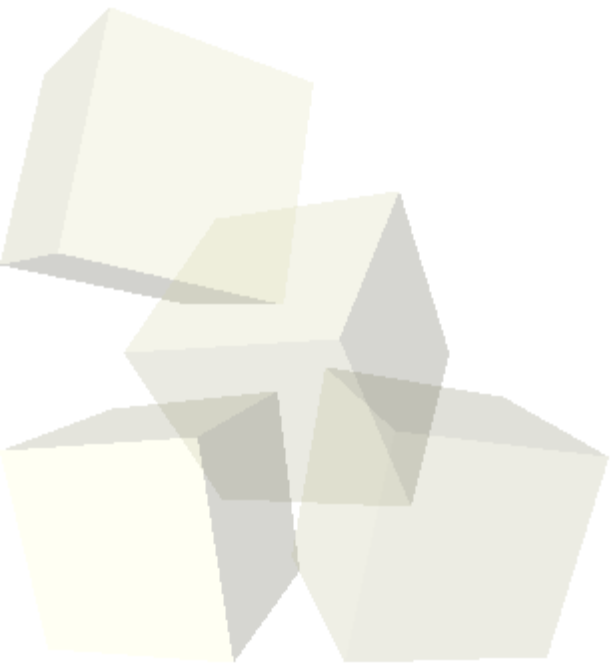


- Uma nova empresa entra no mercado e deseja criar um clone do Orkut
  - ♦ O que precisa ser feito?
  - ♦ Quais as entidades envolvidas?
  - ♦ Quais as operações que cada entidade realiza?



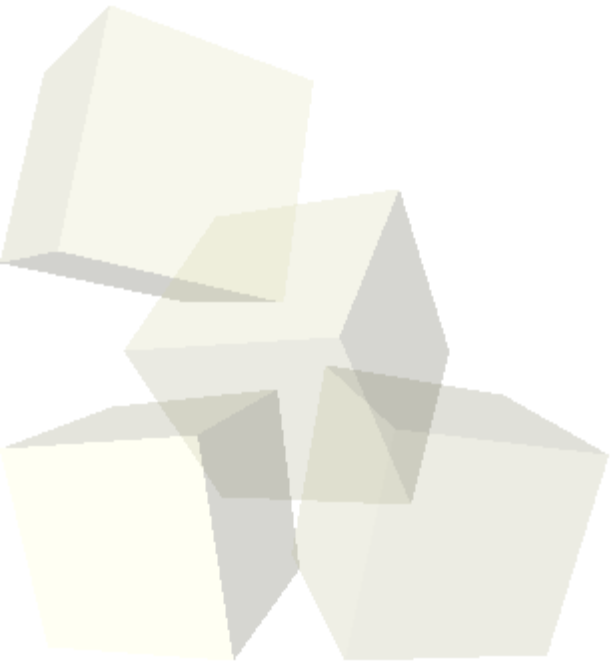


# Alguns Padrões de Projeto





## Outros Conceitos Adotados em C++





- Uma referência é um nome alternativo para uma variável
- Declaração de uma referência:  
*TipoDado &Ref= VarReferenciada;*
- Deve ser obrigatoriamente inicializada com o elemento referenciado, exceto quando:
  - ♦ é parâmetro de uma função
  - ♦ é o valor de retorno de uma função
  - ♦ é membro de uma classe
- Uso de referências
  - ♦ Passagem de parâmetros
  - ♦ Retorno de função
  - ♦ Retorno como referência a um dado privado
  - ♦ Variáveis comuns





```
void Funcao1( int & i )    // i é inicializado quando
{                          // Funcao1 for chamada
    // ...
}

int &Funcao2( );           // Funcao2 retorna uma
                           // referência a um inteiro

main()
{
    // ....
    int &i = Funcao2( );   // i é inicializado quando
                           // Funcao2 retornar
}
```



```
int Valor1 , Valor2;           // variáveis globais
class Classe1{
    int &Numero;                // referência não
                                // inicializada
public:
    Classe1(int i) : Numero(i) // inicialização da
    {                          // referência Numero
        // ...                // no construtor de
    }                          // Classe1
};
int main(){
    // ...
    Classe1 MeuObjeto( Valor1 ); // Inicialização da
    // referência MeuObjeto.Numero com a
    // variável global Valor1

    Classe1 OutroObjeto( Valor2 ); // Inicialização da
    // referência MeuObjeto.Numero com a
    // variável global Valor2
}
```



# Referências: Passagem de Parâmetros

- C++ aceita três maneiras para passar parâmetros para uma função
  - ♦ Passar uma cópia do argumento
  - ♦ Passar um ponteiro para o argumento
  - ♦ Passar uma referência ao argumento

# Referências: Passagem de Parâmetros: Exemplo

```
void Valor( int i ){
    i = i * 2 ;
}
void Ponteiro( int *i ){
    *i = *i * 2 ;
}
void Referencia( int &i ){
    i = i * 2 ;
}
main( ) {
    int x = 5 ;

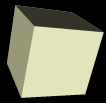
    Valor( x );
    cout << x ;      // mostra  5
    Ponteiro( &x );
    cout << x ;      // mostra 10
    Referencia( x );
    cout << x ;      // mostra 20
}
```

## ■ Exemplo

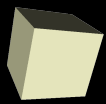
```
int MeuNumero = 0; // Variável global
int &Num( )
{
    return MeuNumero;
}
main()
{
    int i;

    i = Num( );           // i = 0

}
```



- Pode-se retornar uma referência a um dado private de uma classe, possibilitando o acesso ao mesmo
  
- Desvantagens
  - ♦ A sintaxe torna-se confusa
  - ♦ Pouco eficiente
  - ♦ Esta técnica transforma um dado private em public, contrariando o princípio do encapsulamento



# Referências: Retorno de uma referência a um dado private

```
class Data {
    int Dia, Mes, Ano;
public:
    Data(int D, int M, int A);
    ~Data();
    int &ReferenciaAoMes();
}

int &Data::ReferenciaAoMes() {
    Mes = ( Mes < 1 ? 1 : Mes ); // verifica se é
                                // menor que 1 e
                                // acerta
    Mes = ( Mes > 12 ? 12 : Mes ); // idem para 12
    return Mes;
}

main()
{
    Data Inicio(8,12,90);

    cout << Inicio.ReferenciaAoMes();
    Inicio.ReferenciaAoMes() = 15;
    Inicio.ReferenciaAoMes()++;
}
```



# Referências: Referência à uma variável qualquer

- Qualquer variável pode ser uma referência a outra
- Exemplos

```
int val1 = 2;    // val1 é um inteiro  
                // inicializado com "2".
```

```
int& r1 = val1;  // "r1" é uma referência  
                // inicializada com val1.
```

```
int val3 = r1;   // "val3" é inicializado com o valor  
                // de r1 , ou seja, "2".
```

```
r1 = 5;          // o valor de r1, ou seja, o valor de  
                // val1 torna-se "5".
```





- Uma referência pode ser vista como um ponteiro sem que seja preciso ser de-referenciado
  
- Operações não aplicáveis em referências:
  - ♦ Apontar
  - ♦ Saber seu endereço
  - ♦ Comparar
  - ♦ Atribuir um valor
  - ♦ Executar operações aritméticas
  - ♦ Modificar



# Qualificador const

- Torna o elemento ao qual foi aplicado disponível apenas para leitura (*read-only*)
- Pode ser aplicado em

- Variáveis

```
const int i = 10; // i é read-only
```

- Ponteiros

```
int const *ptr1; // ponteiro constante
```

```
const int *ptr2; // ponteiro para um  
                // dado constante
```

```
int *Vetor;
```

```
*ptr1 = 5;           // OK
```

```
ptr1 = Vetor;        // Erro
```

```
*ptr2 = 10;          // Erro
```

```
ptr2 = Vetor;         // OK
```



- ♦ Parâmetros de funções

```
int Calcular( const int *Vetor){  
    for(int i=0;i<10;i++)  
        Vetor[i] ++;    // Erro  
}
```

- ♦ Objetos de uma classe

- Se um objeto for declarado como const, somente podem ser chamadas as funções membros que também forem const
- Deve-se procurar declarar todas as funções que não modifiquem os dados da classe como const.

- ♦ Funções membros

- Se uma função membro for declarada const ela só poderá chamar outra função que também seja const.



# Qualificador const

```
class Data
{
    int Dia, Mes, Ano;
public:
    Data( int D, int M, int A){
        Dia = D;
        Mes = M;
        Ano = A;
    }
    void SetDia( int NovoDia ){
        Dia = NovoDia;
    }
    const int GetDia() // função membro constante
    {
        return Dia;
    }
};

main() {
    const Data Aniversario(30 , 05 , 63);

    cout << Aniversario.GetDia();    // OK

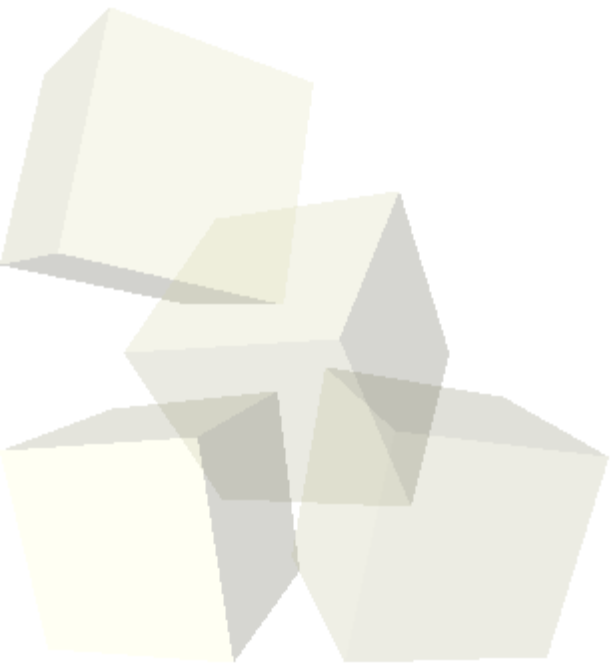
    Aniversario.SetDia(20);           // Erro
}
```



```
class G  
{  
    // ...  
    int MudaValor( );  
    const Mostra( )        // Função membro const  
    {  
        MudaValor( );      // Erro! Esta função não  
                             // é constante  
        // ...  
    }  
};
```



## Outros conceitos adotados em Java





# final

- O modificador do tipo *final* implica que todas as referências futuras a esse elemento vão depender dessa definição. Sendo equivalente ao *const* de C++.
- *final int NOVO = 1;*
- *final int VELHO = 2;*
- **cuidado:** Essa equivalência com C++ não se aplica aos métodos.
- Quando usado em métodos significa que não pode ser sobrescrito (contrário de virtual)

# finalize

- ⇒ É uma função similar ao **destrutor de C++**.
- ⇒ A utilização muito simples: - você adiciona um método a qualquer classe chamado *finalize* e o “runtime” em java chama aquele método sempre que estiver para retornar o espaço daquele objeto.
- ⇒ O coletor de lixo é disparado periodicamente. Antes de uma liberação, java chama o método *finalize* do objeto.



# I/O: Primeira Forma

```
import java.io.*;

public class EntradaSaida {

    public static void main(String args[]) {

        byte[] vetor = new byte[100];
        String teste = null;

        try {
            System.in.read(vetor);
            teste = new String(vetor);
            System.in.skip(100); //descarta buffer teclado
        } catch (Exception e) {}

        System.out.println(teste);
    }
}
```





# I/O: Melhor Forma

```
import java.io.*;

public class EntradaSaida {

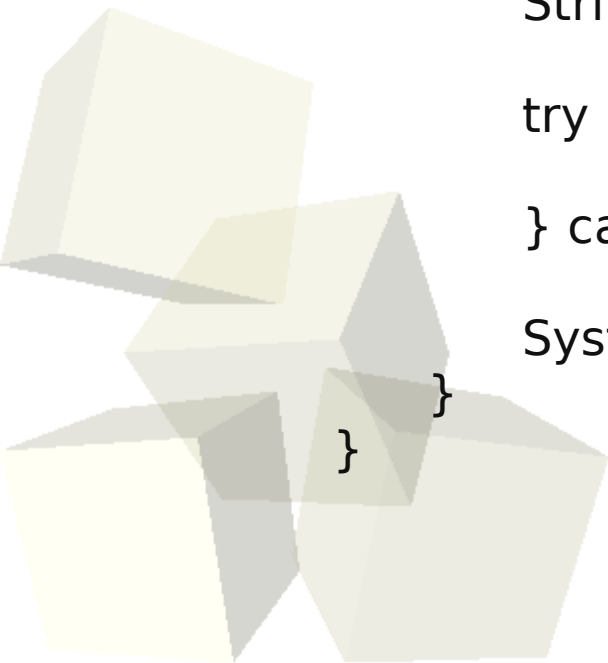
    public static void main(String args[]) {

        InputStreamReader isr = new InputStreamReader(System.in);
        BufferedReader br = new BufferedReader(isr);

        String teste = null;

        try {
            teste = br.readLine();
        } catch (Exception e) {}

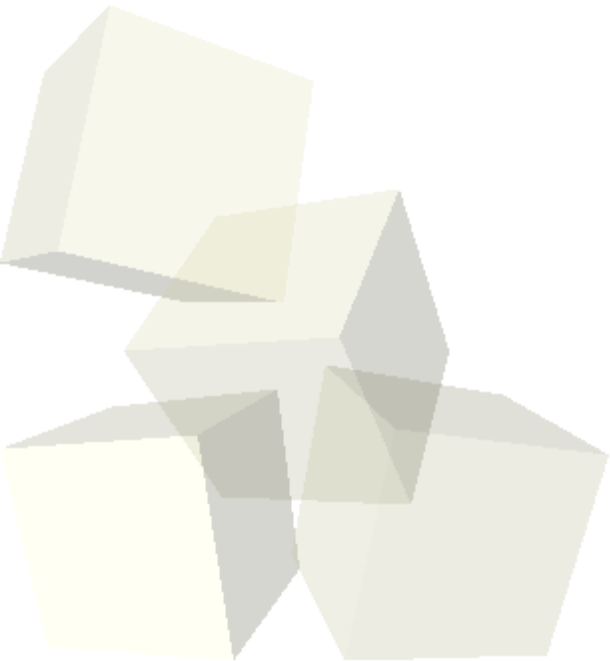
        System.out.println(teste);
    }
}
```





# Algumas das Principais Classes

- String
- Vector
- Hashtable





# Comandos da Classe String

## ■ Construtores:

```
String s = new String();
```

ou

```
String s = "abc";
```

```
System.out.println(s); // imprime (abc) ou
```

```
System.out.println(s.length()); // imprime //3 que é o tamanho da String.
```

## ■ Concatenação de Strings

```
String s = "Ele tem "+ idade + "anos de idade.";
```

é muito mais claro que:

```
String s = new StringBuffer("Ele tem ");
```

```
s.append(idade)
```

```
s.append(" anos de idade.")
```

```
s.toString();
```

que é exatamente o que acontece quando o código é rodado.



# Comandos da Classe String

## ■ Questões de precedência:

*String s = "Ele tem "+ idade + " anos de idade.";*

## ■ Conversões de String:

```
class Ponto {  
    int x, y;  
    Ponto(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public String toString() {  
        return "Ponto[" + x + "," + y + "];"  
    }  
}  
class stringtoString {  
    public static void main(String args[]){  
        Ponto p = new Ponto(10, 20);  
        System.out.println("p = " + p);  
    }  
}
```



# Comandos da Classe String (continuação)

## ■ Extração de Caracteres:

```
class ExtChars{  
    public static void main(String args[ ]){  
        String s = "abc";  
        System.out.println(s.substring(1,2));  
        //caracter [1,2[  
        //imprime b, pois vai do caracter  
        //1 até antes de chegar em 2  
    }  
}
```

## ⇒ Comparação:

```
class equals{  
    public static void main(String args[ ]){  
        String s1 = "Hello";  
        String s2 = "Hello";  
        String s3 = "Good-bye";  
        String s4 = "HELLO";  
        System.out.println( s1 + " equals " +  
            s2 + "-->" + s1.equals(s2));  
        System.out.println( s1 + " equals " +  
            s3 + "-->" + s1.equals(s3));  
        System.out.println( s1 + " equals " +  
            s4 + "-->" + s1.equals(s4));  
        System.out.println( s1 + "  
            equalsIgnoreCase" + s4 + "-->" +  
            s1.equalsIgnoreCase(s4));  
    }  
}
```



# Comandos da Classe String (continuação)

## ■ Igualdade:

```
class EqualNotEqualTo {  
    public static void main(String args[ ]){  
        String s1 = "Hello";  
        String s2 = new String(s1);  
        System.out.println(s1 + "equals" + s2  
            + "--> " + s1.equals(s2));  
        System.out.println(s1 + "==" + s2      +  
            "--> " + (s1==s2));  
    }  
}  
  
// equals testa conteúdo  
// == testa referência
```

## ■ Ordem:

```
class SortString {  
    static String arr[ ] = {  
        "Now", "is", "the", "time", "for", "all",  
        "good", "men", "to", "come", "to", "the",  
        "aid", "of", "their", "country"  
    };  
    public static void main(String args[ ] ) {  
        for (int j = 0; j < arr.length; j++){  
            for (int i = j+1; i < arr.length; i++){  
                if (arr[i].compareTo(arr[j]) < 0) {  
                    String t = arr[j];  
                    arr[j] = arr[i];  
                    arr[i] = t;  
                }  
            }  
        }  
        System.out.println(arr[j]);  
    }  
}
```



# Comandos da Classe String (continuação)

⇒ O resultado deste programa é:

Now is  
aid men  
all of  
come the  
country their  
for time  
good to

⇒ **indexOf** e **lastIndexOf**

*int indexOf(int ch)*

*int lastIndexOf(int ch)*

retorna o índice da primeira(última) ocorrência do caractere *ch*.

*int indexOf(String str)*

*int lastIndexOf(String str)*

retorna o índice do primeiro caractere da primeira(última) ocorrência da substring *str*.

*int indexOf(int ch, int fromIndex)*

*int lastIndexOf(int ch, int fromIndex )*

retorna o índice da primeira ocorrência depois (antes) *fromIndex* caractere *ch*.

*int indexOf(String str, int fromIndex)*

*int lastIndexOf(String str, int fromIndex )*

retorna o índice do primeiro caractere da primeira ocorrência depois (antes) *fromIndex* substring *str*.

```
class IndexOfDemo {
```

```
    public static void main(String args[ ]) {
```

```
        String s="Now is the time for all good men"+           "to  
        come to the aid of their country"+  
        "and pay their due taxes.";
```

```
        System.out.println(s);
```

```
        System.out.println("indexOf(t) = " +  
                            s.indexOf('t'));
```

```
        System.out.println("lastIndexOf(t) = " +  
                            s.lastIndexOf('t'));
```



# Comandos da Classe String (continuação)

```
System.out.println("indexOf(the) = " +  
    s.indexOf("the"));  
System.out.println("lastIndexOf(the) = " +  
    s.lastIndexOf("the"));  
System.out.println("indexOf(t, 10) = " +  
    s.indexOf('t', 10));  
System.out.println("lastIndexOf(t, 60) = " +  
    s.lastIndexOf('t', 60));  
System.out.println("indexOf(the, 10) = " +  
    s.indexOf("the", 10));  
System.out.println("lastIndexOf(the, 60) = " +  
    lastIndexOf("the", 60));
```

⇒ O resultado deste programa é:

Now is the time for all good men to come to the  
aid of their country and pay their due taxes.

```
indexOf(t) = 7  
lastIndexOf(t) = 65  
indexOf(the) = 7  
lastIndexOf(the) = 55  
indexOf(t, 10) = 11  
lastIndexOf(t, 60) = 44  
indexOf(the, 10) = 44  
lastIndexOf(the, 60) = 44
```

## ⇒ Modificações do Texto de String

### □ substring

*"Hello World".substring(6) -> "World"*

*"Hello World".substring(3, 8) -> "lo Wo"*

### □ concat

*"Hello".concat(" World") -> "Hello World"*

### □ replace

*"Hello".replace('l', 'w') -> "Hewwo"*

### □ toLowerCase e toUpperCase

*"Hello".toLowerCase() -> "hello"*

*"Hello".toUpperCase() -> "HELLO"*

### □ trim

*" Hello World ".trim() -> "Hello World"*

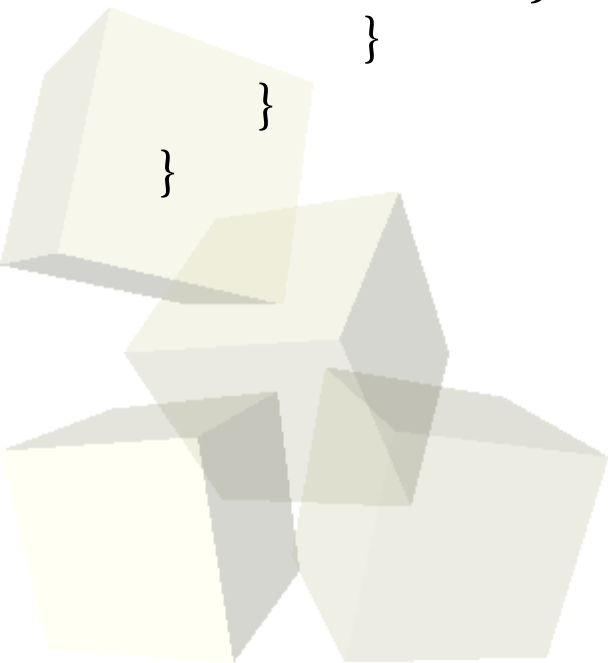




- Utilizado para armazenar objetos

- Exemplo:

```
class Teste {  
    public static void main(String args[]) {  
        Vector t = new Vector();  
        Object x = new Object();  
        t.add(t);  
  
        ...  
        for (int i = 0; i < t.size(); i++) {  
            Object elemento = (Object) t.elementAt(i);  
        }  
    }  
}
```





void	add(int index, Object element)
boolean	add(Object o)
boolean	addAll(Collection c)
boolean	addAll(int index, Collection c)
void	clear()
Object	elementAt(int index)
int	indexOf(Object elem)
Object	remove(int index)
boolean	remove(Object o)
int	size()





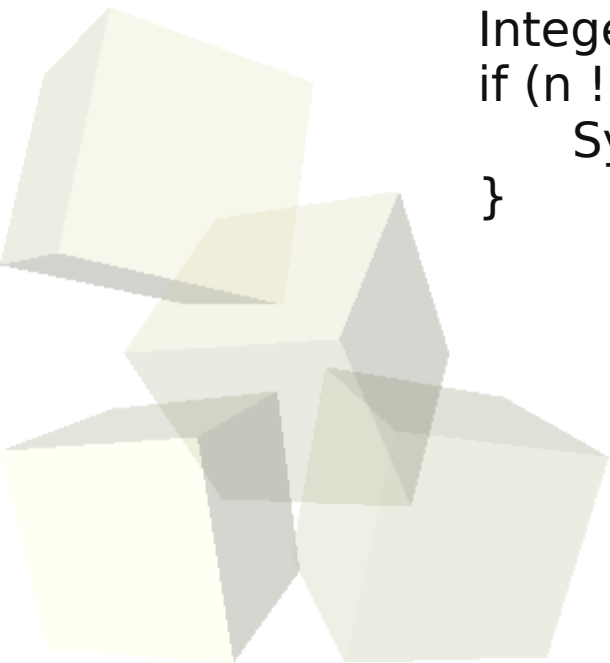
- Parâmetros que afetam performance: initial capacity e load factor.
- Capacity é o número de posições na Hashtable
- A Hashtable é de endereçamento aberto: em caso de colisão, é montada uma lista em cada posição (busca sequencial)
- O load factor é a medida de quão cheia a hashtable deve estar antes de incrementar o número de posições.
- Quando o número de entrada excede a capacidade x load factor então o número de posições é incrementado
- 0,75 é um bom load factor segundo o pessoal da Sun Microsystems



```
Hashtable numbers = new Hashtable();  
numbers.put("one", new Integer(1));  
numbers.put("two", new Integer(2));  
numbers.put("three", new Integer(3));
```

Para recuperar:

```
Integer n = (Integer)numbers.get("two");  
if (n != null) {  
    System.out.println("two = " + n);  
}
```





- São recipientes para classes, sendo ao mesmo tempo, um mecanismo de nomeação e um mecanismo de restrição de visibilidade.
- **A declaração package**
- A primeira coisa permitida em um arquivo java é uma declaração package, que diz ao compilador em qual pacote as classes incluídas devem ser definidas. Se você omitir a declaração package, as classes acabam no pacote padrão, que não tem nome. O compilador java usa os diretórios do sistema de arquivo para armazenar os pacotes. Se você declarar uma classe como estando dentro de um pacote chamado

*meupacote*, então o arquivo-fonte daquela classe deve estar armazenado em um diretório chamado *meupacote*. Lembre-se que as maiúsculas e minúsculas do diretório devem corresponder àquelas do nome do pacote.

*Forma geral da declaração package:*

*package pac1[.pac2[.pac3]];*

⇒ Pode-se criar uma hierarquia de pacotes dentro de pacotes, separando os níveis com pontos. Sendo que esta hierarquia deve refletir no sistema de arquivos de seu sistema de desenvolvimento java. Um pacote declarado como:

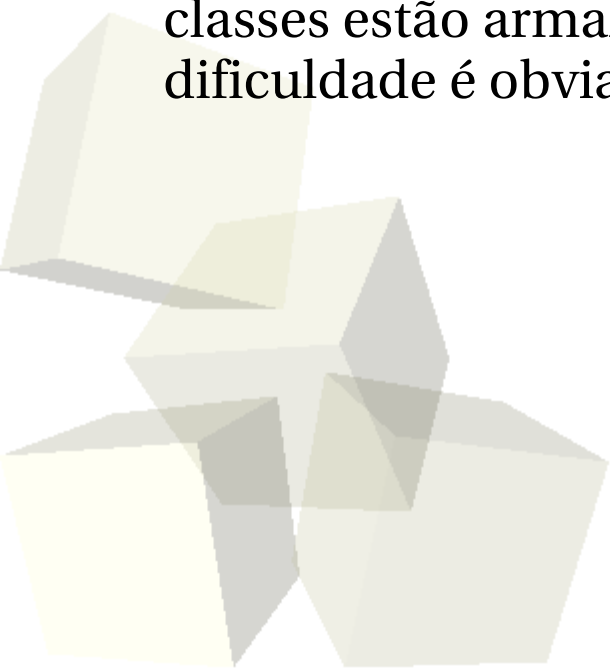
*package java.awt.image*

precisa estar armazenado em *java/awt/image*.



# Pacotes - package (continuação)

- **Compilando classes e pacotes:** os pacotes solucionam muitos problemas, sob a perspectiva de controle de acesso e colisão de espaço de nomes. Entretanto, ela causa algumas dificuldades curiosas em tempo de compilação e execução.
- Não se pode renomear um pacote sem renomear o diretório no qual as classes estão armazenadas. Esta dificuldade é óbvia.
- Se você criar uma classe chamada *TestePac* em um pacote chamado *teste*. Depois, você cria um diretório chamado *teste* e coloca *TestePac.java* dentro desse diretório para compilá-lo.
- Quando for rodar o Interpretador de Java diz “*can't find class Testepac*”. Isto acontece porque essa classe agora está armazenada em um pacote chamado *teste* e você não pode apenas se referir a ela como *TestePac*.





# Pacotes - package (continuação)

Deve-se referir a ela como *teste.TestePac*.

⇒ Portanto, você tenta executar java com *teste.TestePac* e tem a mensagem “can’t find class *teste/TestePac*”.

⇒ Bem, o arquivo *TestePac.class* está bem lá na sua frente no diretório atual, por que java não consegue encontrá-lo? O segredo está oculto em sua variável *CLASSPATH*, que define a parte superior da hierarquia de classes. O problema é que não há diretório *teste* no diretório de trabalho atual, porque você ainda está no próprio diretório *teste*.

⇒ Neste ponto, você tem duas opções:

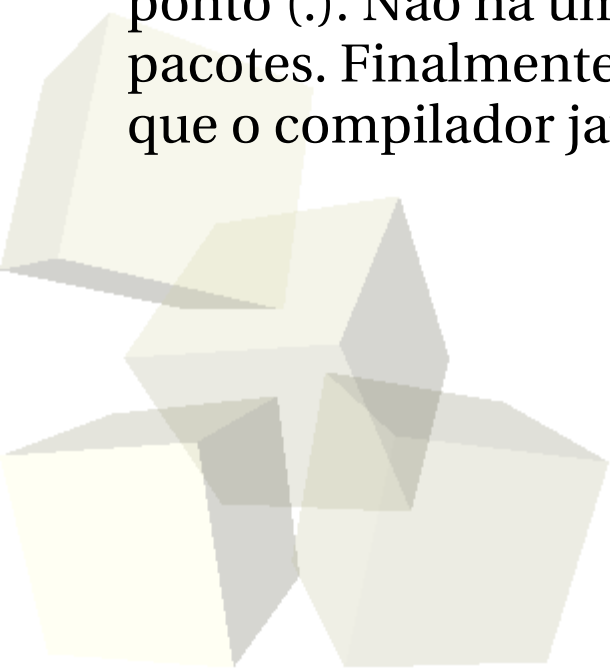
- mude os diretórios um nível acima e tente “java *teste.TestePac*” ou
- adicione a parte superior de sua hierarquia de classe de desenvolvimento à variável de ambiente *CLASSPATH*. Depois, você poderá usar “java *teste.Testepac*” de qualquer diretório e java encontrará o arquivo *.class* correto.



- **A declaração import** - Depois da declaração do package vem uma lista de declarações imports. Todas as classes embutidas de Java estão armazenadas em pacotes.
- A forma geral de declaração import é a seguinte:

***import*** *pacote1*[.*pacote2*].(*nome-classe*|\*);

Onde, *pacote1* é o nome de um pacote de nível superior1, *pacote2* é o nome de um pacote opcionalmente dentro de um pacote externo, separados por um ponto (.). Não há um limite prático para a profundidade de uma hierarquia de pacotes. Finalmente, um *nome-classe* é especificado, ou um (\*), que indica que o compilador java deve pesquisar todo este pacote .

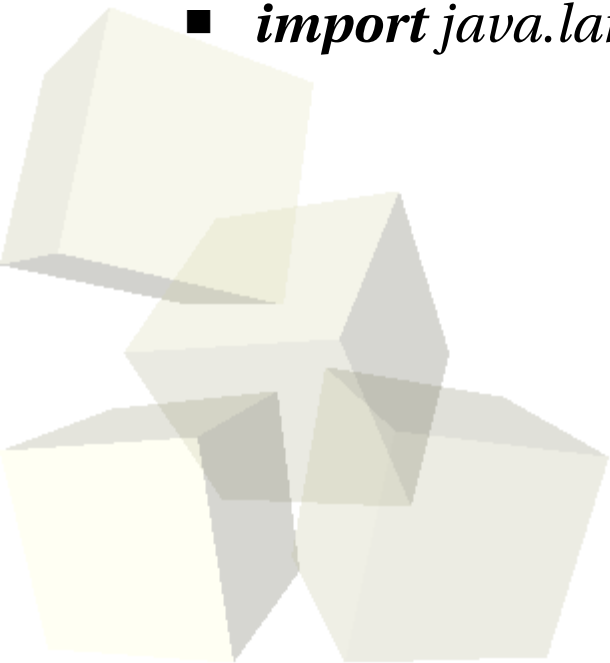






# Pacotes - package (continuação)

- *import java.util.Date*
- *import java.io.\**
- Todas as classes “embutidas” de Java incluídas no software de distribuição estão armazenadas em um pacote chamado *java*. As funções básicas da linguagem estão armazenadas em um pacote dentro do pacote *java* chamado *java.lang*. Todos os pacotes precisam ser importados mas como *java* é inútil sem *java.lang*, ele é implicitamente importado pelo compilador para todos os programas. Sendo equivalente a ter a seguinte linha de comando:
  - *import java.lang.\**



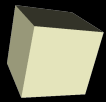


- Em qualquer lugar onde usar um nome de classe, você pode incluir seu nome totalmente qualificado, o que inclui sua hierarquia completa de pacote. Por exemplo:

```
import java.util.*;  
class MinhaData extends Date{  
}
```

*O mesmo exemplo sem import:*

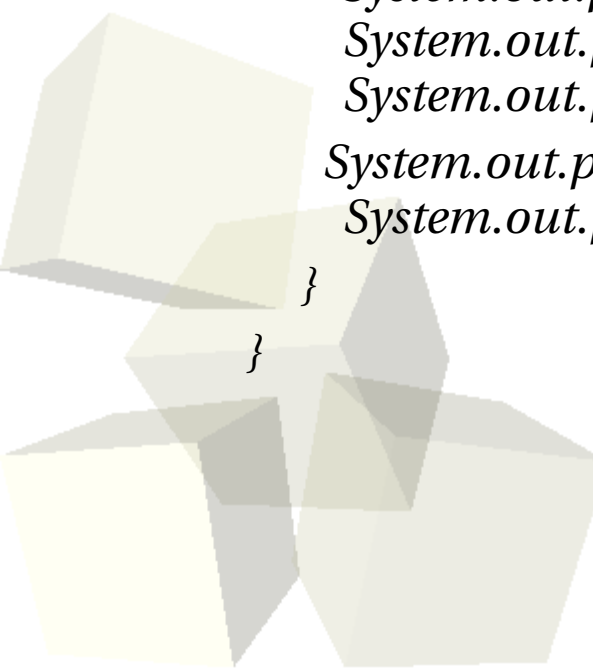
```
class MinhaData extends java.util.Date{  
}
```



# Visibilidade (Encapsulamento)

	<b>private</b>	<b>sem modificador</b>	<b>private protected</b>	<b>protected</b>	<b>public</b>
<b>mesma classe</b>	sim	sim	sim	sim	sim
<b>mesma subclasse de pacote</b>	não	sim	sim	sim	sim
<b>mesma não-sub- classe de pacote</b>	não	sim	não	sim	sim
<b>subclasse de pacote diferente</b>	não	não	sim	sim	sim
<b>não subclasse de pacote diferente</b>	não	não	não	não	sim

```
package p1;  
public class Protecao {  
    int n = 1;  
    private int n_private = 2;  
    protected int n_protected = 3;  
    public int n_public = 4;  
    public Protecao() {  
        System.out.println("Construtor da basica");  
        System.out.println("sem modificador = " + n);  
        System.out.println("privado = " + n_private);  
        System.out.println("protegido = " + n_protected);  
        System.out.println("proteg. e privado = " + n_priv_protec);  
        System.out.println("publico = " + n_public);  
    }  
}
```

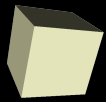




# package - Proteção de acesso (continuação)

```
class Especif extends Protecao {  
    public Especif( ) {  
        System.out.println("Construtor da  
derivada");  
        System.out.println("sem modificador  
= " + n);  
        //System.out.println("número  
privado = " + n_private); //ERRO!  
        System.out.println("protegido = " +  
n_protected);  
        System.out.println("proteg. e privado  
= " + n_priv_protec);  
        System.out.println("publico = " +  
n_public);  
    }  
}
```

```
class mesmopacote{  
    public mesmopacote( ) {  
        protecao p = new protecao( );  
        System.out.println("Construtor do mesmo  
pacote");  
        System.out.println("sem modificador = "  
+ n);  
        //System.out.println("número privado = "  
+ n_private); //ERRO!  
        System.out.println("protegido = " +  
n_protected);  
        //System.out.println("proteg. e privado = "  
+ n_priv_protec); //ERRO!  
        System.out.println("publico = " +  
n_public);  
    }  
}
```



# package - Proteção de acesso (continuação)

```
package p2;  
class Protecao2 extends p1.protecao {  
    Protecao2() {  
        System.out.println("Construtor da  
            especif em outro pacote");  
        //System.out.println("sem  
            modificador = "+ n); //ERRO!  
        //System.out.println("privado = " +  
            n_private); //ERRO!  
        System.out.println("protegido = " +  
            n_protected);  
        System.out.println("proteg. e privado = "  
            + n_priv_protec);  
        System.out.println("publico = " +  
            n_public);  
    }  
}
```

```
class outropacote{  
    outropacote() {  
        p1.protecao p = new p1.protecao();  
        System.out.println("Construtor de  
            outro pacote");  
        //System.out.println("sem modificador = "  
            + n); //ERRO!  
        //System.out.println("número privado = "  
            + n_private); //ERRO!  
        //System.out.println("protegido = " +  
            n_protected); //ERRO!  
        //System.out.println("proteg. e privado = "  
            + n_priv_protec); //ERRO!  
        System.out.println("publico = " +  
            n_public);  
    }  
}
```



# Tratamentos de Exceções

- Exceção é uma condição anormal que surge em uma sequência de código em tempo de execução.
- A forma básica de um bloco de tratamento de exceção:

```
try {  
    // bloco de código  
}catch (ExcecaoTipo1 e) {  
    //manipulador de excecao para  
    //ExcecaoTipo1  
}catch (ExcecaoTipo2 e) {  
    //manipulador de excecao para  
    //ExcecaoTipo2  
throw(e); //lançar a exceção novamente  
}finally {  
}
```

## ■ Exceções não-capturadas

```
class Exc0 {  
    public static void main(String args[]){  
        int d = 0;  
        int a = 42/d;  
    }  
}
```

O resultado quando executado o programa:

```
java.lang.ArithmeticException: /by zero  
at Exc0.main(Exc0.java:4)
```

Como tratar esses erros:

## ■ try e catch

```
class Exc0 {  
    public static void main(String args[]){  
        try{  
            int d = 0;  
            int a = 42/d;  
        } catch (ArithmeticException e) {  
            System.out.println("divisão por zero");  
        }  
    }  
}
```



## ■ Várias cláusulas catch

```
class MultiCatch {  
    public static void main (String args[ ]) {  
        try{  
            int a = args.length;  
            System.out.println("a = " + a);  
            int b = 42 / a;  
            int c[ ] = { 1 };  
            c[42] = 99;  
        } catch(ArithmeticaException e) {  
            System.out.println("div por 0: " + e);  
        } catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println("Estouro índice array" + e);  
        }  
    }  
}
```





## ⇒ Declarações try Aninhadas

```
class MultiAninh {  
    static void proced() {  
        try {  
            int c[ ] = { 1 };  
            c[42] = 99;  
        } catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println("Estouro índice array"+ e);  
        }  
    }  
    public static void main(String args[ ]) {  
        try {  
            int a = args.length;  
            System.out.println("a = " + a);  
            int b = 42 / a;  
            proced();  
        } catch(ArithmeticException e) {  
            System.out.println("div por 0: " + e);  
        } catch(ArrayIndexOutOfBoundsException e) {  
            System.out.println("Estouro índice array"+e); }  
        }  
    }
```



# Tratamentos de Exceções

- **Throw** - usado para lançar uma exceção. A forma geral é:
- **Throws** – para avisar que um método lança algum tipo de exceção

```
class ThrowDemo {  
    void demoproc( ) throws NullPointerException,  
Exception {  
        try {  
            throw new NullPointerException("teste");  
        } catch (NullPointerException e) {  
            System.out.println("Capturada dentro de demoproc");  
            throw e; //Lançada novamente  
        }  
    }  
    public static void main(String args[ ]) {  
        try{  
            ThrowDemo t = new ThrowDemo();  
            t.demoproc( );  
        } catch(Exception e) {  
            System.out.println("recapturada: "+ e);  
        }  
    }  
}
```



# Tratamentos de Exceções

- **finally** - utilizado para declarar quais os blocos que devem ser executados independente das exceções causadas ou capturadas.

```
class FinallyDemo {  
    static void procA( ) throws RuntimeException {  
        try {  
            System.out.println("dentro de procA");  
            throw new RuntimeException("demo");  
        } finally { System.out.println("finally de procA"); }  
    }  
    public static void main(String args[ ]) {  
        try {  
            procA( );  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

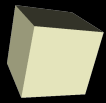


# Tratamentos de Exceções

## ⇒ SubClasses de Exceção

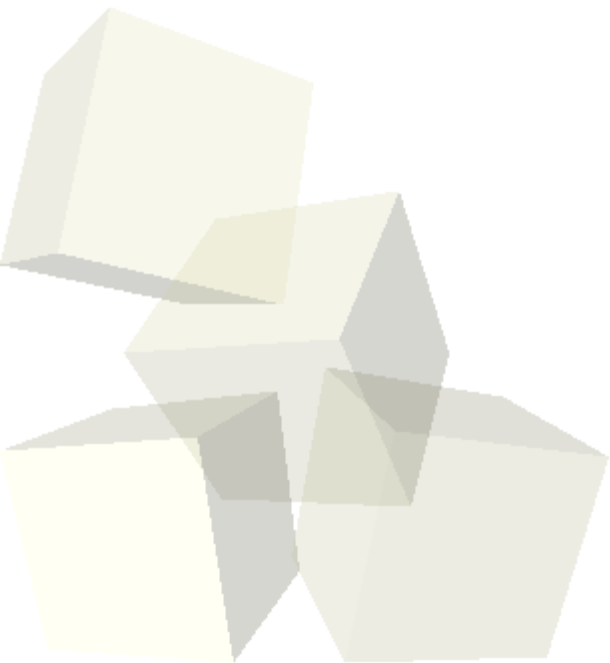
```
class MinhaExcecao extends Exception {  
    private int detalhe;  
    MinhaExcecao(int a) {  
        detalhe = a;  
    }  
    public String toString(){  
        return "MinhaExcecao[" + detalhe + "];"  
    }  
}  
  
class DemoExcecao {  
    static void calcule(int a) throws MinhaExcecao {  
        System.out.println("chamado calcule(" + a + ").");  
        if (a > 10)  
            throw new MinhaExcecao(a);  
        System.out.println("encerramento normal");  
    }  
}
```

```
public static void main  
    (String args[ ]) {  
    try {  
        calcule(1);  
        calcule(20);  
    } catch (MinhaExcecao e) {  
        System.out.println("capturada  
            + e);  
    }  
}
```



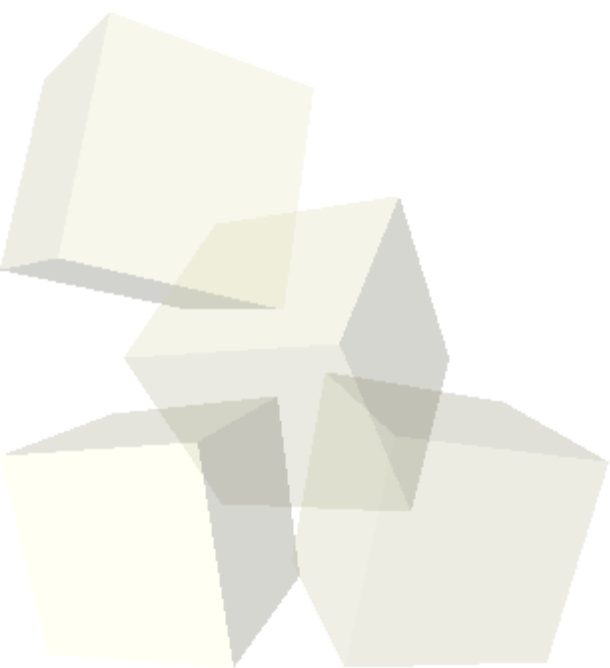
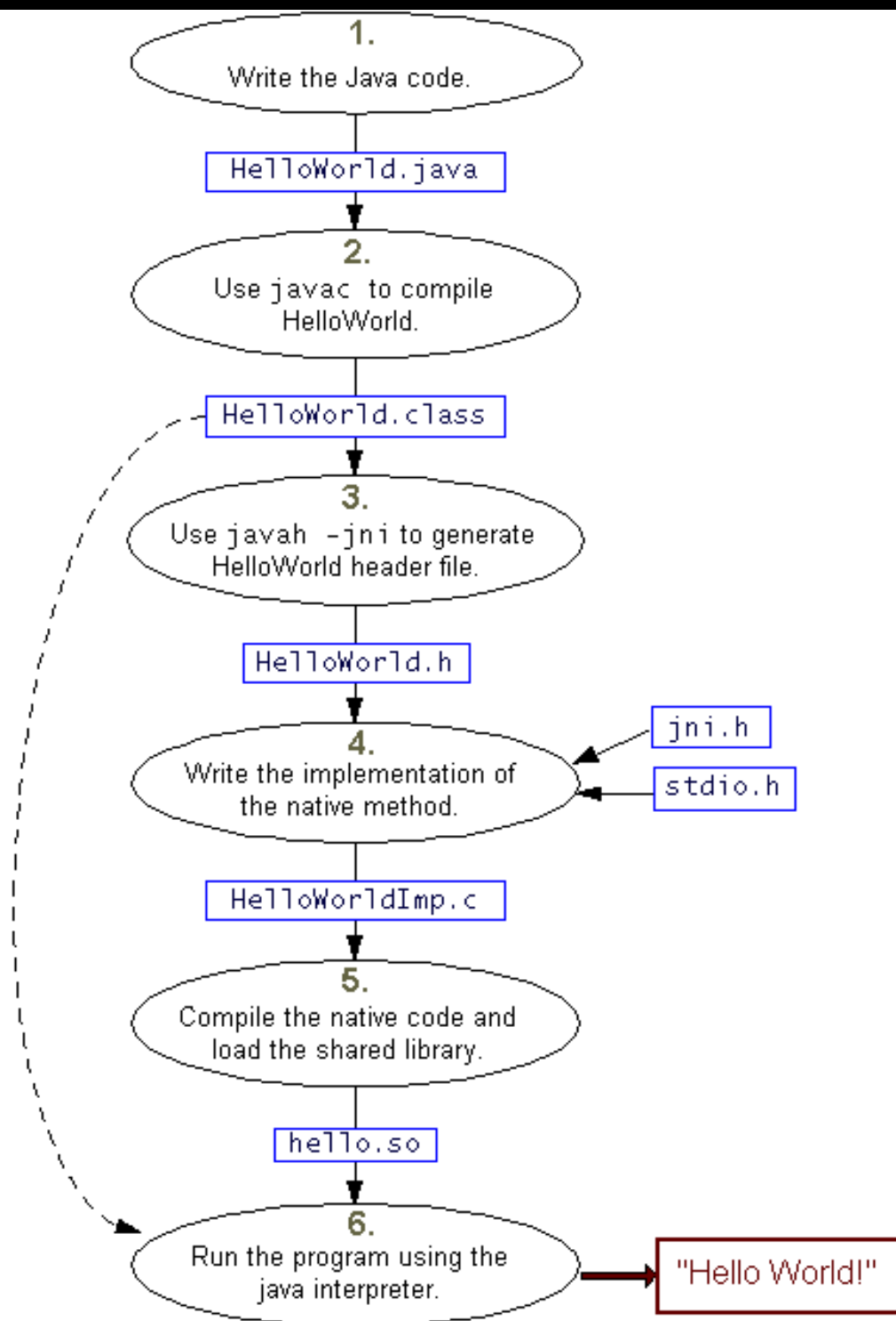
# JNI (Java Native Interface)

- Usada para fazer interface de Java com outras linguagens tais como C e C++
- Por exemplo, pode-se realizar chamadas, à partir de métodos escritos em Java, para funções escritas em C.





# JNI





# JNI – Código do HelloWorld

```
class HelloWorld {  
    public native void displayHelloWorld();  
  
    static {  
        System.loadLibrary("hello");  
    }  
  
    public static void main(String[] args) {  
        new HelloWorld().displayHelloWorld();  
    }  
}
```

Agora é preciso compilar!!!



# Criando o Arquivo de Header

```
javah -jni -classpath . HelloWorld
```

```
#include <jni.h>
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:    HelloWorld
 * Method:   displayHelloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
    (JNIEnv *, jobject);

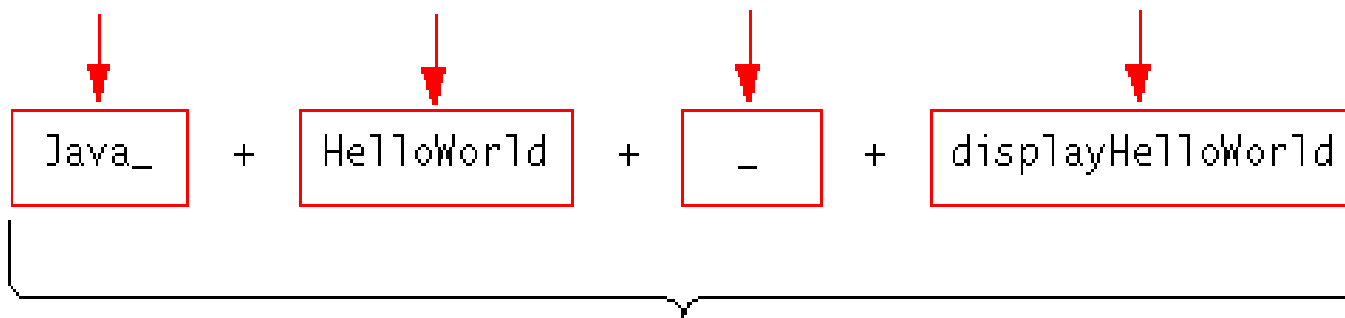
#ifdef __cplusplus
}
#endif
#endif
```





# Como é definido o nome do método?

prefix + fully qualified class name + underscore "\_" + method name



Java\_HelloWorld\_displayHelloWorld



```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

jni.h provê informações que a linguagem nativa requer para interoperar com a linguagem Java!!!



# Criando shared library

```
gcc HelloWorldImp.c -o libhello.so -shared \  
-I/usr/local/j2sdk1.5.0/include -I/usr/local/j2sdk1.5.0/include/linux
```

```
export LD_LIBRARY=`pwd`
```

Para executar:

```
java -classpath . HelloWorld
```

ou

```
export CLASSPATH=.:$CLASSPATH  
java HelloWorld  
# . deve ser o primeiro path
```





# Multithreading e Sincronização

## ■ Diferenças entre Multiprocessamento e Multithreads

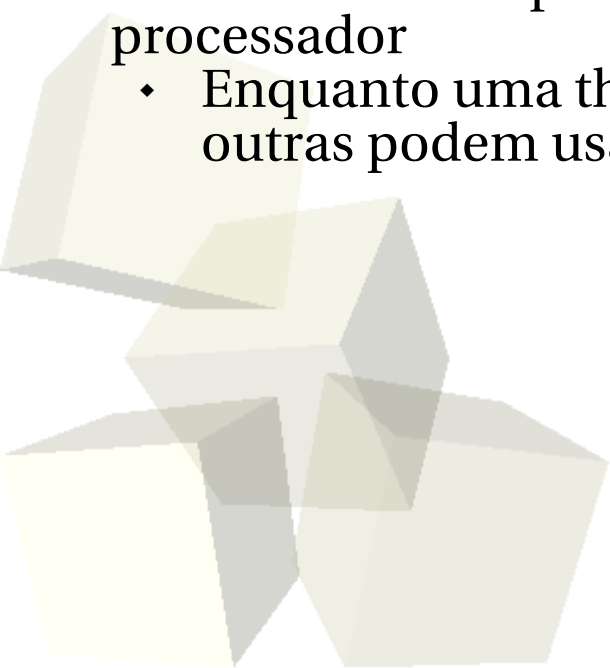
- Multiprocessamento - estão em espaço de endereçamento distinto e devem ser vistos como programas diferentes no mesmo sistema.
- Multithreads - compartilham o mesmo espaço de endereçamento e dividem cooperativamente o mesmo processo.

## ■ Permite melhor aproveitamento do processador

- Enquanto uma thread faz I/O outras podem usar o processador

## ■ Sincronização

- Como as linhas de execução utilizam um comportamento assíncrono, é necessário forçar a sincronização quando precisar dela. Para isto foi definido um conceito chamado **monitor**. Em Java, não há uma classe *monitor*, cada objeto tem seu próprio monitor, que pode chamar um dos seus métodos *sincronizados* do objeto. Depois que uma linha está dentro de um método sincronizado, nenhuma outra linha pode chamar nenhum outro método sincronizado no mesmo objeto.



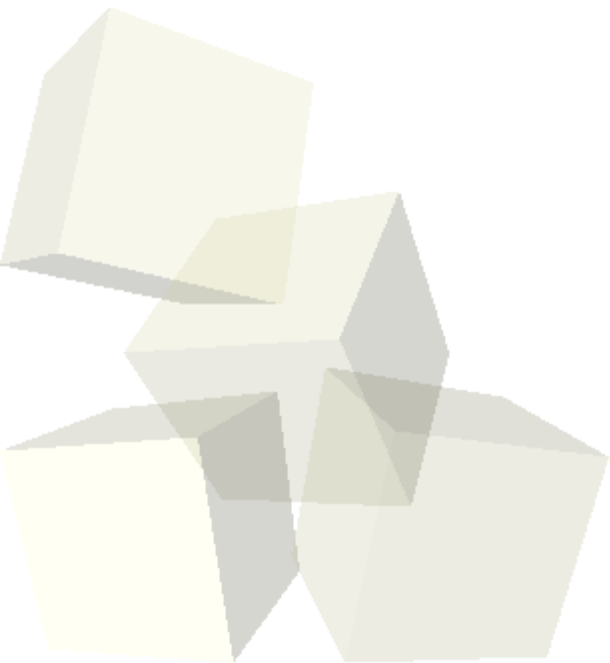


- A classe *Thread* encapsula todos os controles que você precisa ter dentro das linhas de execução. Tenha em mente a diferença entre um objeto *Thread* e linhas de execução: *Thread* é o procurador de uma linha em execução.
- Quando um programa Java é iniciado já existe uma linha sendo executada.
- Exemplo que mostra como manipular uma linha em execução no momento.

```
class Thread1 extends Thread {  
  
    public void run() {  
        while (true)  
            System.out.println("Thread1");  
    }  
}  
  
class Thread2 extends Thread {  
  
    public void run() {  
        while (true)  
            System.out.println("Thread2");  
    }  
}  
  
public class Teste {  
  
    public static void main(String args[]) {  
        Thread1 t1 = new Thread1();  
        Thread2 t2 = new Thread2();  
        t1.start();  
        t2.start();  
    }  
}
```



- Para trabalhar com mais de uma linha de execução deve-se criar instâncias de *Thread*.
- Cada instância pode executar um código distinto (nova classe Thread)





- Podendo iniciar uma linha em qualquer objeto que implemente a interface *Runnable*. Sendo a interface que abstrai a noção de querer que algum código seja “executado” de forma assíncrona. Para isto é necessário que a classe implemente um método *run*.

```
class DemoLinha implements Runnable {  
    DemoLinha() {  
        Thread ct = Thread.currentThread( );  
        System.out.println("Linhacorrente: " + ct);  
        Thread t = new Thread(this, "Linha Demo");  
        t.start();  
        try {
```

```
            Thread.sleep(3000);  
        } catch (InterruptedException e) {  
            System.out.println("interrompida");  
        }  
        System.out.println("encerrando linha  
        pricipal");  
    }  
    public void run( ) {  
        try {  
            for (int i = 5; i > 0; i--) {  
                System.out.println(" " + i);  
                Thread.sleep(1000);  
            }  
        } catch (InterruptedException e) {  
            System.out.println("linha filha  
            interrompida");  
        }  
        System.out.println("encerrando linha-filha");  
    }  
    public static void main(String args[]) {  
        new DemoLinha( ); } }
```



⇒ *Este programa ao executar:*

*Linhacorrente: Thread[main, 5, main]*

*Linha criada: Thread[linha Demo, 5, main]*

5

4

3

*encerrando linha principal*

2

1

*encerrando linha-filha*

## ⇒ **Prioridades de Linha**

Quando uma linha de prioridade baixa está sendo executada e uma linha de prioridade mais alta torna-se ativa, ou sai de uma espera de E/S, ele deve poder ser executada imediatamente.

Exemplo de duas linhas de execução com prioridade diferentes:

```
class clicador implements runnable {  
    int clique = 0;  
    private Thread t;  
    private boolean rodando = true;  
    public clicador (int p) {  
        t = new Thread(this);  
        t.setPriority(p);  
    }  
    public void run( ) {  
        while (rodando) {  
            clique++;  
        }  
    }  
    public void stop( ) {  
        rodando = false;  
    }  
    public void start( ) {  
        t.start( );  
    }  
}
```





```
class PriAltaBaixa {  
    public static void main (String args[ ]) {  
        Thread.currentThread( ).  
            setPriority(Thread.MAX_PRIORITY);  
        clicador alta = new clicador(  
            Thread.NORM_PRIORITY + 2);  
        clicador baixa = new clicador(  
            Thread.NORM_PRIORITY - 2);  
        baixa.start( );  
        alta.start( );  
        try { Thread.sleep(10000); }  
        catch (exception e) {}  
        baixa.stop( );  
        alta.stop( );  
        System.out.println(baixa.clique + " vs. " +  
            alta.clique);  
    }  
}
```

- O resultado deste programa é:
- java PriAltaBaixa  
304300 vs. 4066666



- Para se entrar em um monitor de objetos, faz-se através da chamada de um método marcado com *synchronized*.

*synchronized void metodo() {}*

```
class Chamame {  
    void chamar(String msg) { /* synchronized  
        void chamar(String msg) { */  
        System.out.print("[ " + msg);  
        try Thread.sleep(1000); catch  
            (Exception e);  
        System.out.print("]");  
    }  
}
```

```
class Chamador implements Runnable  
{  
    String msg;  
    Chamame alvo;  
    public Chamador(Chamame t, String  
        s){  
        alvo = t;  
        msg = s;  
        new Thread(this).start();  
    }  
    public void run() {  
        /* synchronized (alvo) { */  
        alvo.chamar(msg);  
    }  
}  
class Synch {  
    public static void main(String args[])  
    {  
        Chamame alvo = new  
        Chamame();  
        new Chamador(alvo, "Oi");  
        new Chamador(alvo, "Mundo");  
        new Chamador(alvo,  
            "Sincronizado");  
    }  
}
```



# Comunicação Interlinhas

## ■ Produtor e Consumidor

```
class Q {  
    int n;  
    synchronized int get() {  
        System.out.println("Obtive: " + n);  
        return n;  
    }  
    synchronized void put(int n) {  
        this.n = n;  
        System.out.println("Pus: " + n); }  
}  
class Produtor implements Runnable {  
    Q q;  
    Produtor(Q q) {  
        this.q = q;  
        new Thread(this, "Produtor").start();  
    }  
}
```

```
    public void run() {  
        int i = 0;  
        while(true) {  
            q.put(i++); } } }  
class Consumidor implements Runnable  
{  
    Q q;  
    Consumidor(Q q) {  
        this.q = q;  
        new Threads(this,  
            "Consumidor").start();  
    }  
    public void run() {  
        while(true) {  
            q.get(); } }  
}  
class PC {  
    public static void main(String args[]) {  
        Q q = new Q();  
        new Produtor(q);  
        new Consumidor(q);  
    }  
}
```



# Comunicação Interlinhas

- Produtor e Consumidor usando o mecanismo de “pooling”.

```
class Q {  
    int n;  
    boolean valorEstab = false;  
    int get() {  
        while (!valorEstab)  
            ;  
        System.out.println("Obtive: " + n); } }  
    valorEstab = false;  
    return n;  
}  
void put(int n) {  
    while (valorEstab)  
        ;  
    this.n = n;  
    valueSet = true;  
    System.out.println("Pus: " + n); }  
}  
}
```

- ⇒ Produtor e Consumidor usando *wait* e *notify*.

```
class Q {  
    int n;  
    boolean valorEstab = false;  
    synchronized int get() {  
        if (!valorEstab)  
            try wait(); catch (InterruptedException  
e);  
        System.out.println("Obtive: " + n); } }  
    valueSet = false;  
    notify();  
    return n;  
}  
synchronized void put(int n) {  
    if (valueSet)  
        try wait(); catch (InterruptedException  
e);  
    this.n = n;  
    valueSet = true;  
    System.out.println("Pus: " + n);  
    notify(); }  
}
```



# Bloqueio Mútuo

```
class A {  
    synchronized void foo(B b) {  
        String name =  
            Thread.currentThread().getName();  
        System.out.println(name + " entrou em  
            A.foo");  
        try Thread.sleep(1000); catch (Exception e);  
        System.out.println(name + " testando  
            chamarB.last()");  
        b.last();  
    }  
    synchronized void last() {  
        System.out.println(" dentro A.last");  
    }  
}
```

```
class B {  
    synchronized void bar(A a) {  
        String name =  
            Thread.currentThread().getName();  
        System.out.println(name + " entrou  
            em B.bar");  
        try Thread.sleep(1000); catch  
            (Exception e);  
        System.out.println(name + " testando  
            chamarA.last()");  
        a.last();  
    }  
    synchronized void last() {  
        System.out.println(" dentro B.last");  
    }  
}
```



# Bloqueio Mútuo

```
class Deadlock implements Runnable {  
    A a = new A();  
    B b = new B();  
    Deadlock() {  
  
        Thread.currentThread().setName("Main  
Thread");  
        new Thread(this).start();  
        a.foo(b);    //consegue bloqueio a nesta  
linha.  
        System.out.println("de novo na linha  
principal");  
    }  
    public void run() {  
  
        Thread.currentThread().setName("Raci  
ngThread");  
        b.bar(a);    //consegue bloqueio em b em  
outra linha.  
        System.out.println("de novo em outra  
linha");  
    }  
}
```

```
public static void main(String args[ ]) {  
    new Deadlock();  
}  
}
```

O resultado deste programa é:

MainThread entrou em A.foo

RacingThread entrou em B.bar

MainThread tentando chamar B.last( )

RacingThread tentando chamar A.last( )

^C -> Full Thread Dump

"RacingThread"(.. state:MONITOR\_WAIT)  
prio=5

A.last(Deadlock.java:8)

B.bar(Deadlock.java:17)

Deadlock.run(Deadlock.java:32)

java.lang.Thread.run(Thread.java:289)

.....



# UDP (User Datagram Protocol)

- Uso do protocolo UDP na camada de transporte
- Não garante a entrega da informação
- Classes importantes no Java:
  - ♦ InetAddress
  - ♦ DatagramSocket
  - ♦ DatagramPacket





## ⇒ InetAddress

Java dá suporte ao sistema de nome da Internet com a classe **InetAddress**. Essa classe não tem construtores visíveis. Então para criar um objeto é necessário um dos *métodos de fábrica* (estáticos) disponíveis, que são três: *getLocalHost*, *getByName* e *getAllByName*.

### Exemplo:

```
InetAddress Address = InetAddress.getLocalHost( );  
System.out.println(Address);  
Address= InetAddress.getByName("mailhost");  
System.out.println(Address);  
InetAddress SW[ ] = InetAddress.getAllByName("www.starwave.com");  
System.out.println(SW)
```

A classe InetAddress também tem alguns métodos não-estáticos, que são: *getHostName( )*, *getAddress( )*, *toString( )*.





# Exemplo Sender

```
import java.net.*;

class DatagramaSender extends Thread {

    public DatagramSocket ds;
    public int portaServ = 2345;
    public int portaCli = 2346;
    public int tam_buffer = 1024;

    public byte buffer[] = new byte[tam_buffer];

    public DatagramaSender() throws Exception {
        ds = new DatagramSocket(portaServ);
    }

    public void run() {
        int pos = 0;
        System.out.println("Começe a digitar!!!");
        while (true) {
            try {
                int c = System.in.read();
```



# Exemplo Sender

```
switch (c) {
    case -1:
        System.out.println("Servidor sai.");
        return;
    case '\r':
        break;
    case '\n':
        ds.send(new DatagramPacket (buffer,pos,
            InetAddress.getLocalHost(), portaCli));
        pos = 0;
        break;
    default:
        buffer[pos++] = (byte) c;
}
} catch (Exception e) {}
}

public static void main(String args[]) {
    try {
        DatagramaSender ds = new DatagramaSender();
        ds.start();
    } catch (Exception e) {}
}
```



# Exemplo Receiver

```
import java.net.*;

public class DatagramaReceiver extends Thread {
    public DatagramSocket ds;
    public int portaServ = 2345;
    public int portaCli = 2346;
    public int tam_buffer = 1024;

    public byte buffer[] = new byte[tam_buffer];

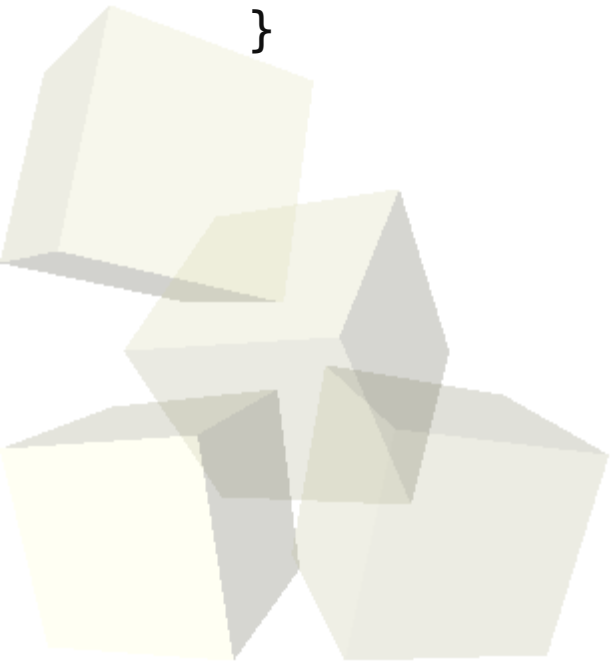
    public DatagramaReceiver() throws Exception {
        ds = new DatagramSocket(portaCli);
    }

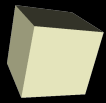
    public void run() {
        while(true) {
            try {
                DatagramPacket p = new DatagramPacket (buffer, buffer.length);
                ds.receive(p);
                System.out.println(new String(p.getData(), 0, 0, p.getLength()));
            } catch (Exception e){}
        }
    }
}
```



# Exemplo Receiver

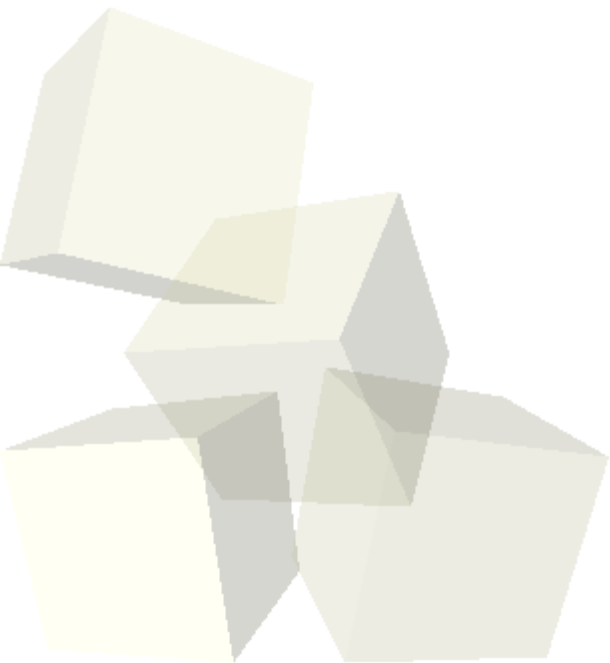
```
public static void main(String args[]) {  
    try {  
        DatagramaReceiver dc = new DatagramaReceiver();  
        dc.start();  
    } catch (Exception e) {}  
}
```





# TCP (Transfer Control Protocol)

- Protocolo na camada de Transporte
- Confiável
- Orientado à Conexões
- Principais Classes:
  - ♦ SocketServer
  - ♦ Socket





# Exemplo1 - Servidor

```
import java.net.*;
import java.io.*;

public class Servidor {

    private ServerSocket server;
    private BufferedReader br;
    private PrintStream ps;
    private Socket socket;

    public Servidor() {
        try {
            server = new ServerSocket(1234);
        } catch (Exception e) {}
    }

    public void accept() {
        try {
            socket = server.accept();
        } catch (Exception e) {}
    }
}
```



# Exemplo1 - Servidor

```
public void obterCanais() {
    try {
        br = new BufferedReader(new InputStreamReader(
                                socket.getInputStream()));
        ps = new PrintStream(socket.getOutputStream());
    } catch (Exception e) {}
}
public void enviar(String texto) {
    try {
        ps.println(texto);
    } catch (Exception e) {}
}
public String receber() {
    try {
        return br.readLine();
    } catch (Exception e) {}
    return null;
}
public void close() {
    try {
        socket.close();
    } catch (Exception e) {}
}
```



# Exemplo1 - Servidor

```
public static void main(String args[]) {  
    System.out.println("Sou o servidor...");  
    Servidor s = new Servidor();  
    System.out.println("Esperando mensagem...");  
    while (true) {  
        s.accept();  
        s.obterCanais();  
        System.out.println("Esperando...");  
  
        try { Thread.sleep(5000); } catch (Exception e) {}  
  
        System.out.println(s.receber());  
  
        s.close();  
    }  
}
```





# Exemplo1 - Cliente

```
import java.net.*;
import java.io.*;

public class Cliente {

    private BufferedReader br;
    private PrintStream ps;
    private Socket socket;

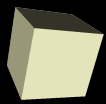
    public Cliente() {
        try {
            socket = new Socket("127.0.0.1", 1234);
        } catch (Exception e) {}
    }

    public void obterCanais() {
        try {
            br = new BufferedReader(new InputStreamReader(
                                     socket.getInputStream()));
            ps = new PrintStream(socket.getOutputStream());
        } catch (Exception e) {}
    }
}
```



# Exemplo1 - Cliente

```
public void enviar(String texto) {  
    try {  
        ps.println(texto);  
    } catch (Exception e) {}  
}  
public String receber() {  
    try {  
        return br.readLine();  
    } catch (Exception e) {}  
    return null;  
}  
public void close() {  
    try {  
        socket.close();  
    } catch (Exception e) {}  
}  
public static void main(String args[]) {  
    Cliente c = new Cliente();  
    c.obterCanais();  
    System.out.println("Sou o cliente...");  
    System.out.println("Vou enviar...");  
  
    c.enviar("MENSAGEM");  
    c.close();  
}  
}
```



# Exemplo2 – Servidor usando Threads (Atendendo Múltiplos Clientes)

```
import java.net.*;
import java.io.*;

public class Servidor {
    private ServerSocket server;
    public Servidor() {
        try {
            server = new ServerSocket(1234);
        } catch (Exception e) {}
    }
    public void accept() {
        try {
            while (true) {
                TrataCliente tc = new TrataCliente(server.accept());
                tc.start();
            }
        } catch (Exception e) {}
    }
    public static void main(String args[]) {
        System.out.println("Sou o servidor...");
        Servidor s = new Servidor();
        s.accept();
    }
}
```



## Exemplo2 – TrataCliente

```
import java.net.*;
import java.io.*;
public class TrataCliente extends Thread {
    private BufferedReader br;
    private PrintStream ps;
    private Socket socket;

    public TrataCliente(Socket socket) {
        this.socket = socket;
        this.obterCanais();
    }

    public void obterCanais() {
        try {
            br = new BufferedReader(new InputStreamReader(
                                   socket.getInputStream()));
            ps = new PrintStream(socket.getOutputStream());
        } catch (Exception e) {}
    }

    public void enviar(String texto) {
        try {
            ps.println(texto);
        } catch (Exception e) {}
    }
}
```



# Exemplo2 - TrataCliente

```
public String receber() {  
    try {  
        return br.readLine();  
    } catch (Exception e) {}  
    return null;  
}
```

```
public void close() {  
    try {  
        socket.close();  
    } catch (Exception e) {}  
}
```

```
public void run() {  
    System.out.println("Tratando cliente...");  
    System.out.println("Recebendo Mensagem...");  
    try { Thread.sleep(10000); } catch (Exception e) {}  
    System.out.println(this.receber());  
    this.close();  
    this.stop();  
}
```



```
import java.net.*;  
import java.io.*;
```

```
public class Cliente {
```

```
    private BufferedReader br;  
    private PrintStream ps;  
    private Socket socket;
```

```
    public Cliente() {  
        try {  
            socket = new Socket("127.0.0.1", 1234);  
        } catch (Exception e) {}  
    }
```

```
    public void obterCanais() {  
        try {  
            br = new BufferedReader(new InputStreamReader(  
                                    socket.getInputStream()));  
            ps = new PrintStream(socket.getOutputStream());  
        } catch (Exception e) {}  
    }
```

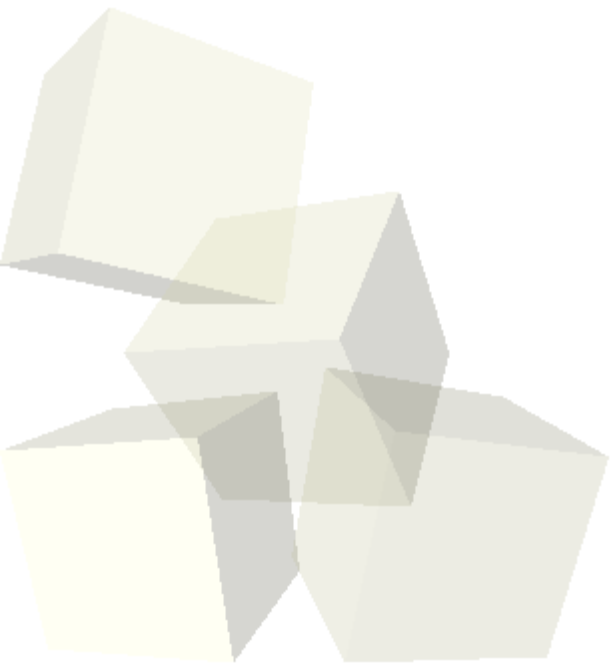


```
public void enviar(String texto) {
    try {
        ps.println(texto);
    } catch (Exception e) {}
}
public String receber() {
    try {
        return br.readLine();
    } catch (Exception e) {}
    return null;
}
public void close() {
    try {
        socket.close();
    } catch (Exception e) {}
}
public static void main(String args[]) {
    Cliente c = new Cliente();
    c.obterCanais();
    System.out.println("Sou o cliente...");
    System.out.println("Vou enviar...");
    c.enviar("MENSAGEM");
    c.close();
}
```



# Corrigindo Problemas

- Eliminando os try-catch intermediários
- Próximo exemplo troca Objetos entre Computadores







```
import java.net.*;
import java.io.*;

public class Servidor {

    private ServerSocket server;
    private ObjectInputStream ois;
    private Socket socket;

    public Servidor() throws Exception {
        server = new ServerSocket(1234);
    }

    public void accept() throws Exception {
        socket = server.accept();
    }

    public void obterCanais() throws Exception {
        ois = new ObjectInputStream(socket.getInputStream());
    }

    public Object receber() throws Exception {
        return ois.readObject();
    }
}
```

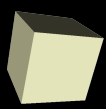


```
public void close() throws Exception {
    socket.close();
}

public static void main(String args[]) {
    try {
        System.out.println("Sou o servidor...");
        Servidor s = new Servidor();
        System.out.println("Esperando mensagem...");
        while (true) {
            s.accept();
            s.obterCanais();
            System.out.println("Esperando...");

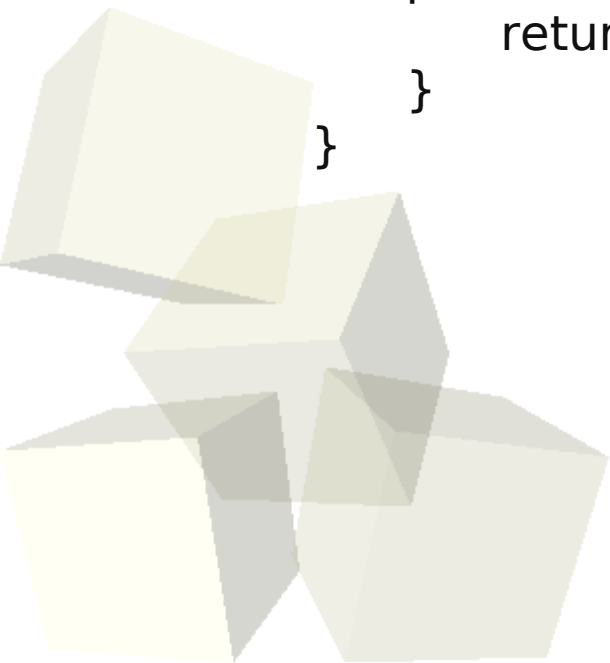
            Objeto o = (Objeto) s.receber();
            System.out.println(o.soma());

            s.close();
        }
    } catch (Exception e) {}
}
```



# Classe de Troca: deve existir no CLASSPATH dos dois lados

```
class Objeto extends Object implements Serializable {  
    private int i;  
    private int j;  
  
    public Objeto(int i, int j) {  
        this.i = i;  
        this.j = j;  
    }  
  
    public int soma() {  
        return i + j;  
    }  
}
```





```
import java.net.*;
import java.io.*;

public class Cliente {

    private ObjectOutputStream oos;
    private Socket socket;

    public Cliente() throws Exception {
        socket = new Socket("127.0.0.1", 1234);
    }

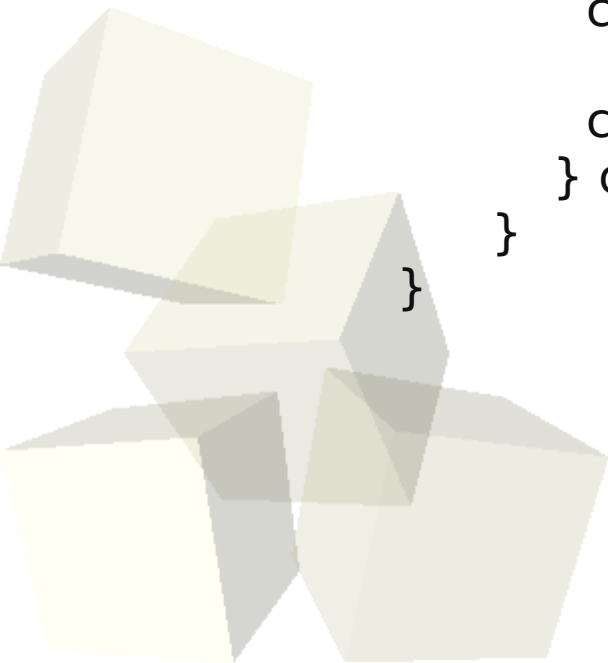
    public void obterCanais() throws Exception {
        oos = new ObjectOutputStream(socket.getOutputStream());
    }

    public void enviar(Objeto o) throws Exception {
        oos.writeObject(o);
    }

    public void close() throws Exception {
        socket.close();
    }
}
```

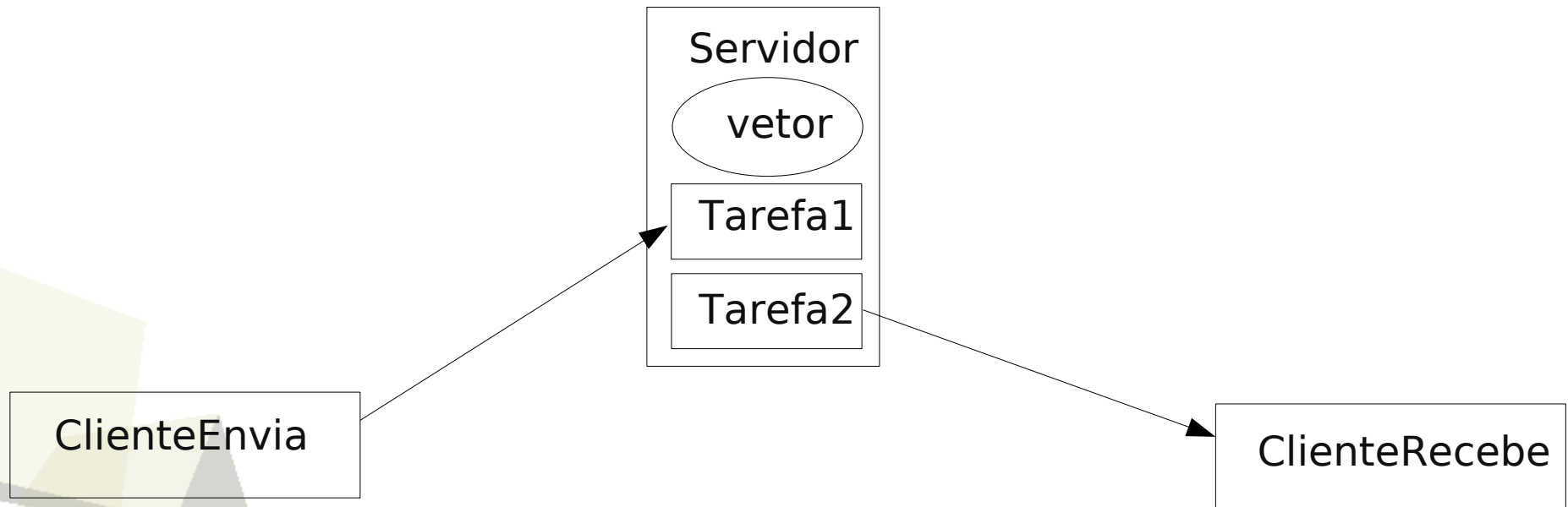


```
public static void main(String args[]) {  
    try {  
        Cliente c = new Cliente();  
        c.obterCanais();  
        System.out.println("Sou o cliente...");  
        System.out.println("Vou enviar...");  
  
        Objeto o = new Objeto(10, 20);  
        c.enviar(o);  
  
        c.close();  
    } catch (Exception e) {}  
}
```





# Exemplo (Próximo a Jini)





```
import java.util.*;
import java.net.*;
import java.io.*;

public class Servidor {

    public Vector vetor;

    public Servidor() {
        vetor = new Vector();
    }

    public static void main(String args[]) {
        Servidor s = new Servidor();
        Tarefa1 t1 = new Tarefa1(s);
        Tarefa2 t2 = new Tarefa2(s);

        t1.start();
        t2.start();
    }
}
```

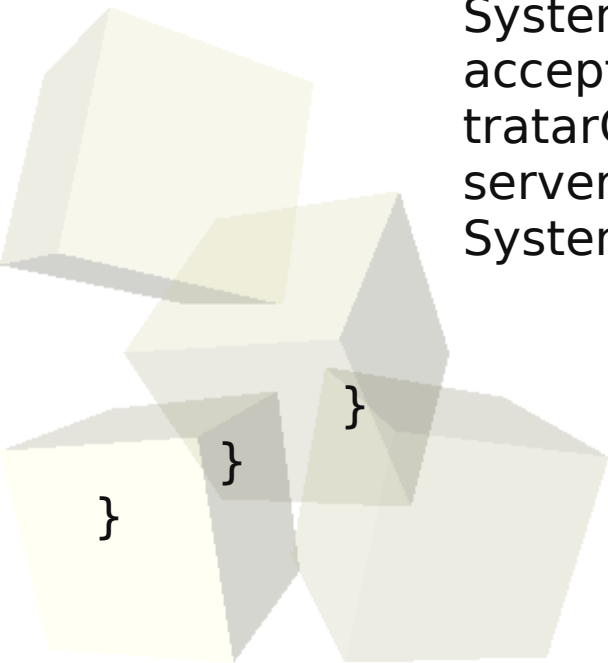


```
class Tarefa1 extends Thread {  
    ServerSocket s;  
    Socket socket;  
    ObjectInputStream ois;  
    Servidor server;  
    public Tarefa1(Servidor server) {  
        try {  
            this.server = server;  
            s = new ServerSocket(1234);  
        } catch (Exception e) {}  
    }  
  
    public void accept () {  
        try {  
            socket = s.accept();  
        } catch (Exception e){}  
    }  
  
    public void tratarCanais() {  
        try {  
            ois = new ObjectInputStream(  
                socket.getInputStream());  
        } catch (Exception e) {}  
    }  
}
```



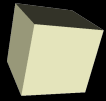


```
public Object receber() {  
    try {  
        return ois.readObject();  
    } catch (Exception e) {}  
    return null;  
}  
public void close() {  
    try {  
        socket.close();  
    } catch (Exception e) {}  
}  
public void run() {  
    while (true) {  
        System.out.println("Aguardando Objeto...");  
        accept();  
        tratarCanais();  
        server.vetor.add(receber());  
        System.out.println("Objeto armazenado... "+  
            "Temos "+server.vetor.size()+  
            " objetos em nosso vetor.");  
    }  
}
```





```
class Tarefa2 extends Thread {  
    ServerSocket s;  
    Socket socket;  
    ObjectOutputStream oos;  
    Servidor server;  
    public Tarefa2(Servidor server) {  
        try {  
            this.server = server;  
            s = new ServerSocket(2345);  
        } catch (Exception e) {}  
    }  
  
    public void accept () {  
        try {  
            socket = s.accept();  
        } catch (Exception e){}  
    }  
  
    public void tratarCanais() {  
        try {  
            oos = new ObjectOutputStream(  
                socket.getOutputStream());  
        } catch (Exception e) {}  
    }  
}
```



```
public void enviar(Object o) {
    try {
        oos.writeObject(o);
    } catch (Exception e) {}
}
public void close() {
    try {
        socket.close();
    } catch (Exception e) {}
}
public void run() {
    while (true) {
        System.out.println("Aguardando conexoes de "+
                           +"clientes que requisitam Objetos...");

        accept();
        tratarCanais();
        if (server.vetor.size() > 0)
        {
            enviar(server.vetor.elementAt(0));
            server.vetor.removeElementAt(0);
        }

        System.out.println("Objeto enviado...");
    }
}
}
```



# Objeto Trocado

class ObjRmt extends Thread implements Serializable {

```
    int a;  
    int b;  
    double c;  
    boolean fnl = false;
```

```
    public ObjRmt(int a, int b) {  
        this.a = a;  
        this.b = b;  
    }
```

```
    public void run() {  
        this.c += (a * b) / 0.333;  
        fnl = true;  
    }
```

```
    public boolean getFinal() {  
        return fnl;  
    }
```

```
}
```




```
import java.net.*;
import java.io.*;

public class ClienteEnvia {

    private ObjectOutputStream oos;
    private Socket socket;

    public ClienteEnvia() {
        try {
            socket = new Socket("127.0.0.1", 1234);
        } catch (Exception e) {}
    }

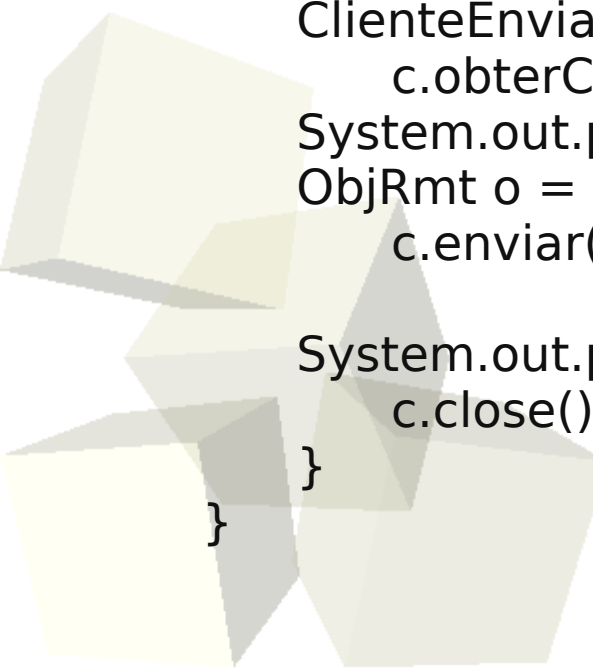
    public void obterCanais() {
        try {
            oos = new ObjectOutputStream(socket.getOutputStream());
        } catch (Exception e) {}
    }
}
```



```
public void enviar(Object o) {  
    try {  
        oos.writeObject(o);  
    } catch (Exception e) {}  
}
```

```
public void close() {  
    try {  
        socket.close();  
    } catch (Exception e) {}  
}
```

```
public static void main(String args[]) {  
    ClienteEnvia c = new ClienteEnvia();  
    c.obterCanais();  
    System.out.println("Enviando Objeto...");  
    ObjRmt o = new ObjRmt(15, 25);  
    c.enviar(o);  
  
    System.out.println("Objeto enviado");  
    c.close();  
}
```





```
import java.net.*;
import java.io.*;

public class ClienteRecebe {

    private ObjectInputStream ois;
    private Socket socket;

    public ClienteRecebe() {
        try {
            socket = new Socket("127.0.0.1", 2345);
        } catch (Exception e) {}
    }

    public void obterCanais() {
        try {
            ois = new ObjectInputStream(socket.getInputStream());
        } catch (Exception e) {}
    }
}
```



# ClienteRecebe

```
public Object receber() {  
    try {  
        return ois.readObject();  
    } catch (Exception e) {}  
    return null;  
}  
  
public void close() {  
    try {  
        socket.close();  
    } catch (Exception e) {}  
}  
  
public static void main(String args[]) {  
    ClienteRecebe c = new ClienteRecebe();  
    c.obterCanais();  
    System.out.println("Aguardando Objeto...");  
    ObjRmt o = (ObjRmt) c.receber();  
    System.out.println(o.a+" - "+o.b);  
    c.close();  
}  
}
```





# RMI (Remote Method Invocation)

- Uma aplicação RMI consiste de dois programas separados: Servidor e Cliente
- Servidor
  - ♦ Cria alguns objetos e os oferece aos clientes
- Cliente
  - ♦ Obtém a referência para um dos objetos do Servidor e invoca seus métodos
- Tal suporte é conhecido como *distributed object application*



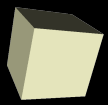
# Escrevendo o Servidor RMI

```
package compute;
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
    Object executeTask(Task t) throws RemoteException;
}
```

```
package compute;
import java.io.Serializable;

public interface Task extends Serializable {
    Object execute();
}
```



```
package engine;
import java.rmi.*;
import java.rmi.server.*;
import compute.*;
public class ComputeEngine extends UnicastRemoteObject
    implements Compute {
    public ComputeEngine() throws RemoteException {super();}
    public Object executeTask(Task t) {return t.execute();}
    public static void main(String[] args) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        String name = "//kerberos/Compute";
        try {
            Compute engine = new ComputeEngine();
            Naming.rebind(name, engine);
            System.out.println("ComputeEngine bound");
        } catch (Exception e) {
            System.err.println("ComputeEngine exception: " +
                e.getMessage());
            e.printStackTrace();
        }
    }
}
```



```
package client;
import java.rmi.*;
import java.math.*;
import compute.*;
public class Client {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            String name = "/" + args[0] + "/Compute";
            Compute comp = (Compute) Naming.lookup(name);
            Tarefa task = new Tarefa(args[1]);
            BigDecimal pi = (BigDecimal) (comp.executeTask(task));
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("Client exception: " +
                               e.getMessage());
            e.printStackTrace();
        }
    }
}
```



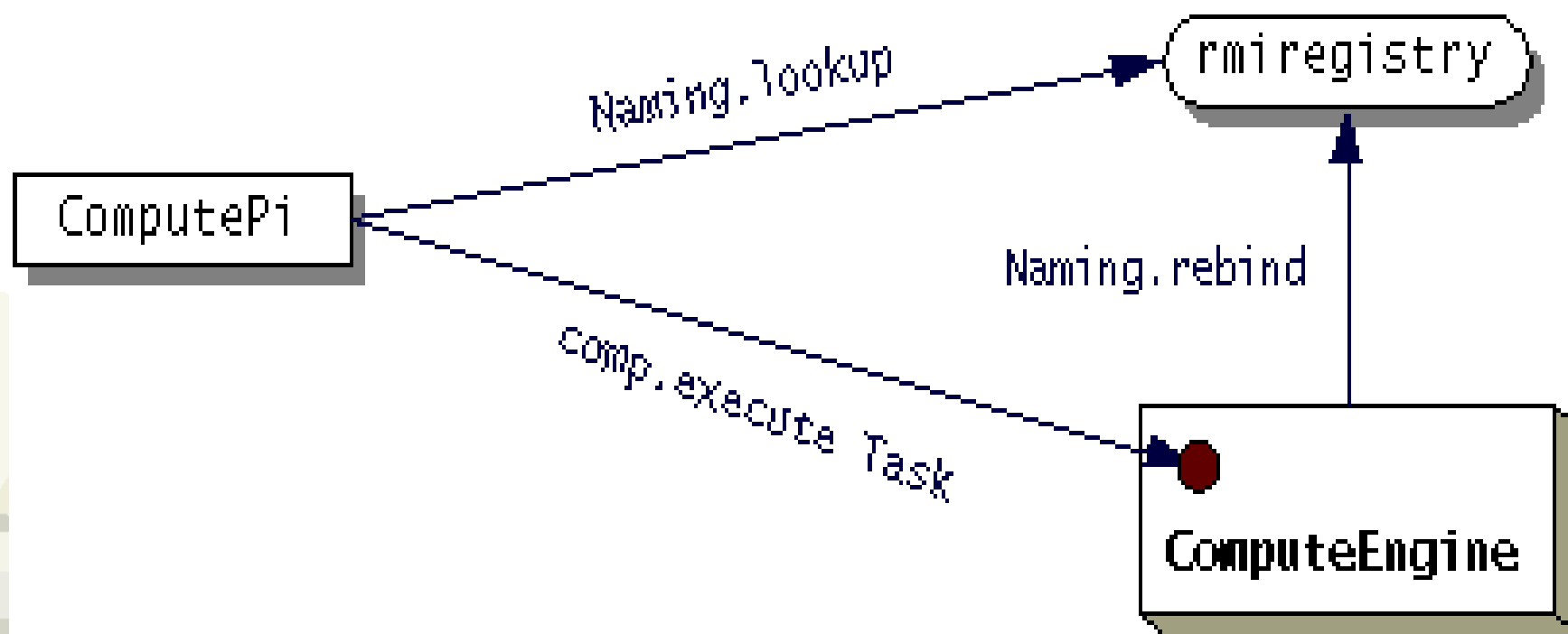
```
package client;
public class Tarefa implements Task {

    public String task;
    public Tarefa(String task) {
        this.task = task;
    }

    public void execute() {
        System.out.println(task);
    }

}
```







Compilando e gerando Package para Interfaces:

```
javac compute/Compute.java  
javac compute/Task.java  
jar cvf compute.jar compute/*.class
```

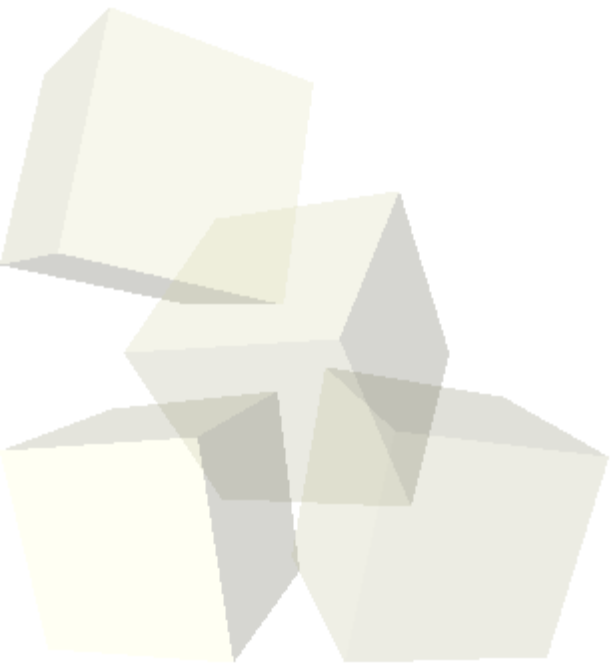
Compilando Servidor:

```
export CLASSPATH=$CLASSPATH:/path_compute/compute.jar  
javac engine/ComputeEngine.java  
rmic -d . engine.ComputeEngine  
mkdir /var/www/html/classes/engine/  
cp engine/ComputeEngine_*.class /var/www/html/classes/engine/  
cd /var/www/html/classes/  
jar xvf /path_compute/compute.jar  
cp /path_compute/compute.jar .
```



Compilando o Cliente:

```
export CLASSPATH=$CLASSPATH:/path_compute/compute.jar  
javac client/Client.java  
javac -d /var/www/html/classes client/Tarefa.java
```







```
grant {  
    permission java.net.SocketPermission "*:1024-65535",  
        "connect,accept";  
    permission java.net.SocketPermission "*:80", "connect";  
};
```

Há permissão de conectar e aceitar conexões em portas acima de 1024 até 65535

Há permissão de conectar em qualquer servidor na porta 80



Iniciando Servidor de Registro (por default usa porta 1099):

```
unset CLASSPATH  
rmiregistry&
```

Registrando Objetos:

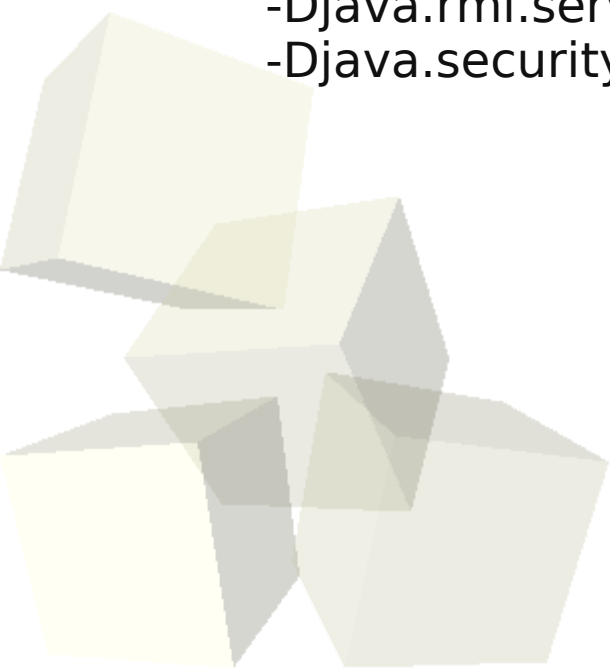
```
export CLASSPATH=/path_compute/./var/www/html/classes/compute.jar  
java -Djava.rmi.server.codebase=http://kerberos/classes/  
-Djava.rmi.server.hostname=kerberos  
-Djava.security.policy=java.policy engine.ComputeEngine
```



Executando Cliente:

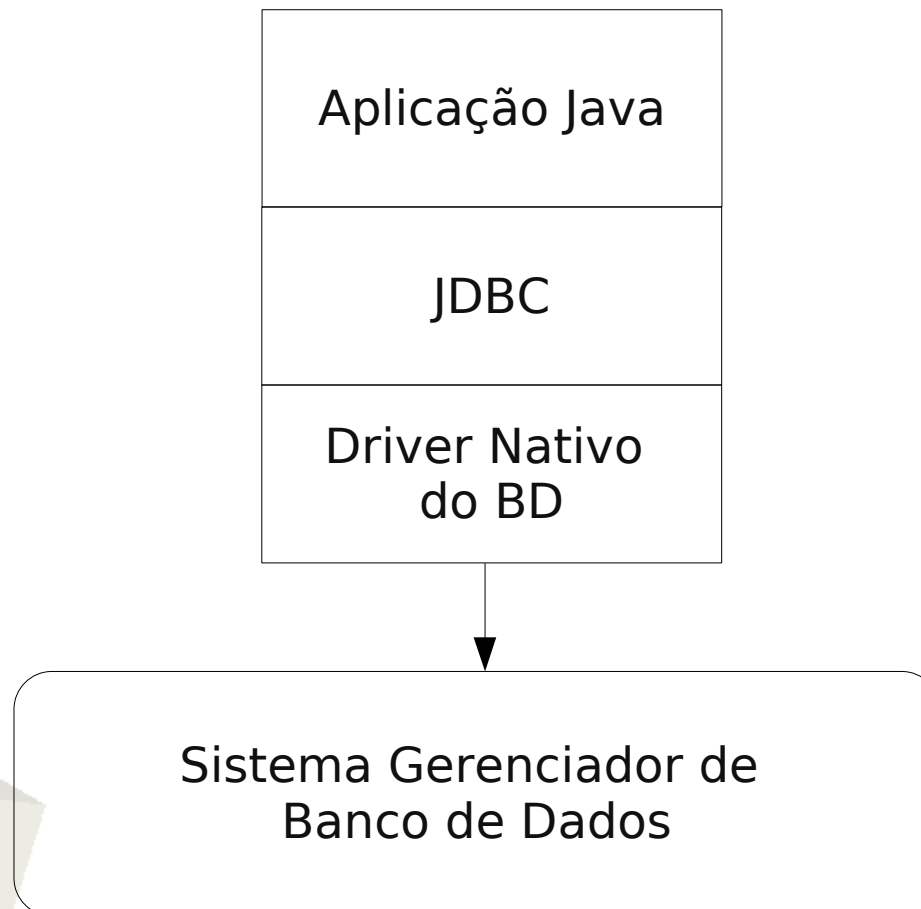
```
export CLASSPATH=/path_compute/./path_compute/compute.jar
```

```
java -Djava.rmi.server.codebase=http://kerberos/classes/  
-Djava.rmi.server.hostname=kerberos  
-Djava.security.policy=java.policy client.Client kerberos TEXTO
```



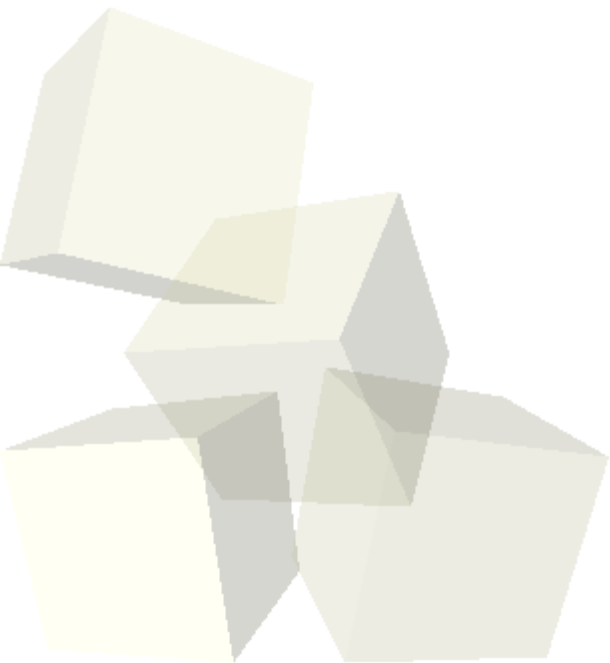


# JDBC (Java Database Connectivity)



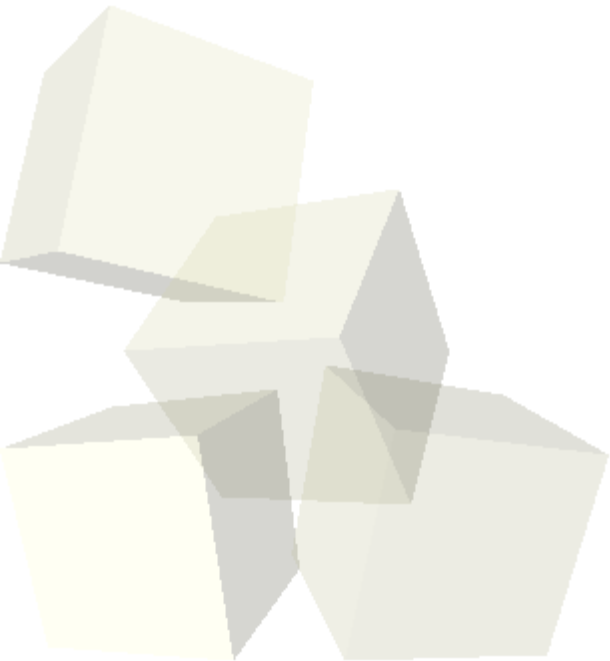


- Configurar um SGBD
  - ♦ PostgreSQL
- Instalação
- Criando um Usuário
- Configurações de acesso `/var/lib/pgsql`





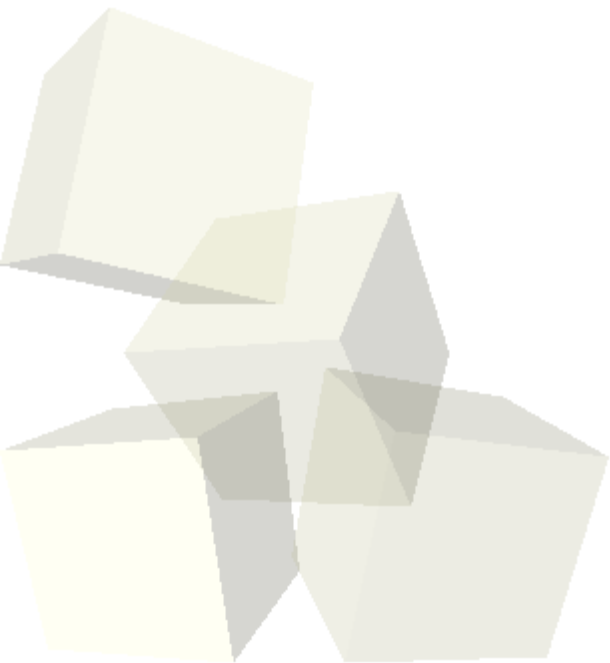
- Utilizando psql para criar base de dados
- Criar algumas tabelas
- Utilizar selects





# Terceiro Passo

- Para utilizar JDBC precisamos de um Driver para o banco de dados
  - ♦ Aquisição desse driver





# Construindo a Classe Base

```
import java.sql.*;
public class BDados {
    private String driver;
    private String url;
    String login;
    String senha;
    Connection conn;
    public BDados(String driver, String url, String login, String senha)
        throws Exception {
        this.driver = driver;
        this.url = url;
        this.login = login;
        this.senha = senha;
        Class.forName(driver);
        conn = DriverManager.getConnection(url, login, senha);
    }
    public ResultSet query(String str) throws Exception {
        ResultSet rs = null;
        Statement stmt = conn.createStatement();
        rs = stmt.executeQuery(str);
        return rs;
    }
}
```



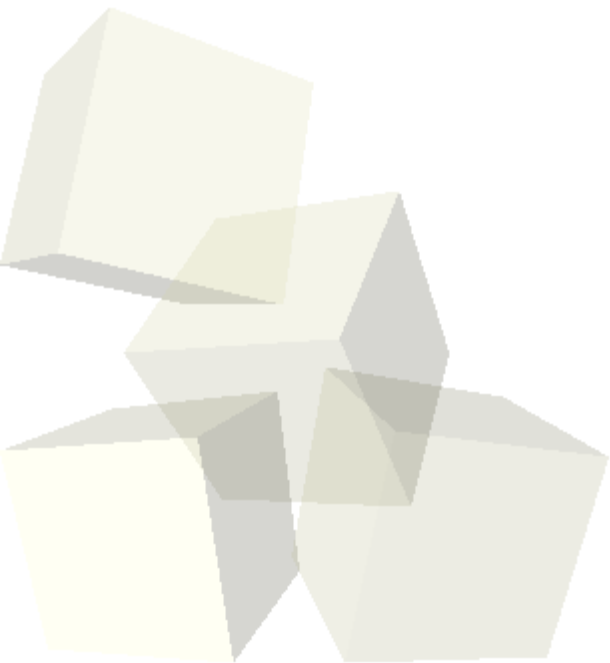


# Construindo a Classe Base

```
public int update(String str) throws Exception {  
    int r = 0;  
    Statement stmt = conn.createStatement();  
    r = stmt.executeUpdate(str);  
    return r;  
}  
public void close() throws Exception {  
    conn.close();  
}  
public void beginWork() throws Exception {  
    conn.setAutoCommit(false);  
}  
public void commitWork() throws Exception {  
    conn.commit();  
    conn.setAutoCommit(true);  
}  
public void rollbackWork() throws Exception {  
    conn.rollback();  
    conn.setAutoCommit(true);  
}  
}
```

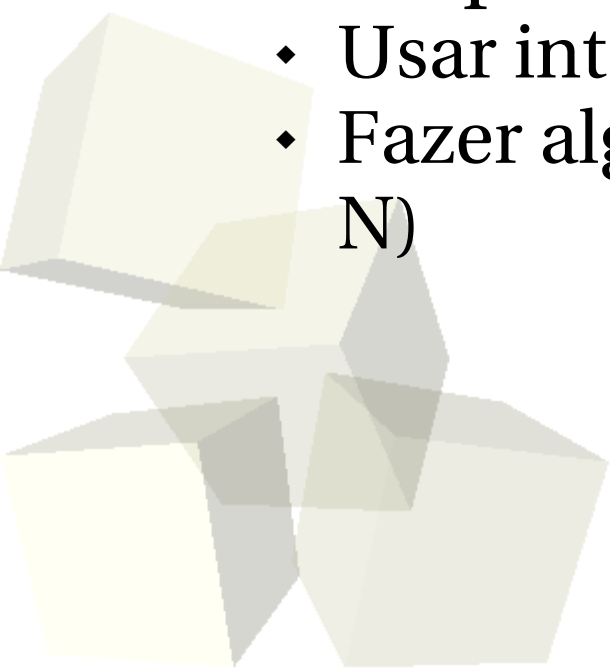


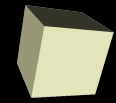
- Adicionar método main e manipular um pouco do Banco de Dados dentro de BDados...



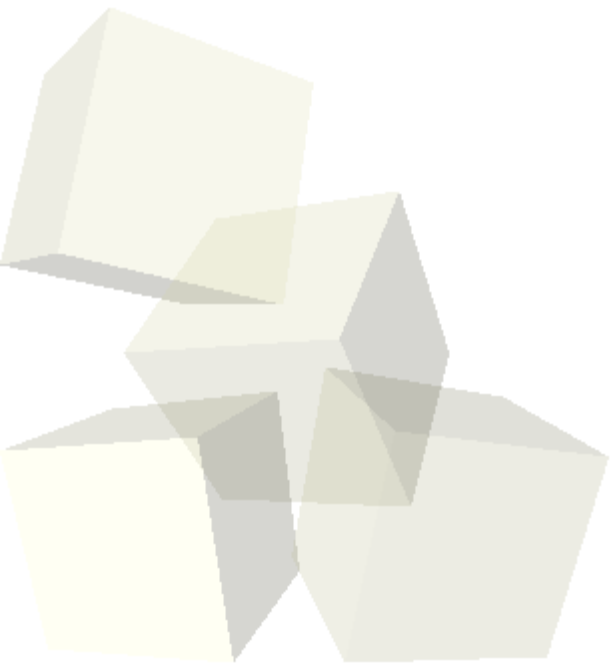


- Criação de classes que mascaram internamente o acesso ao banco de dados
- Acesso ao banco fica transparente ao desenvolvedor
- Exemplos...
  - ♦ Usar interface para definir drivers etc
  - ♦ Fazer algumas classes, relacionamentos (1-1; 1-N)



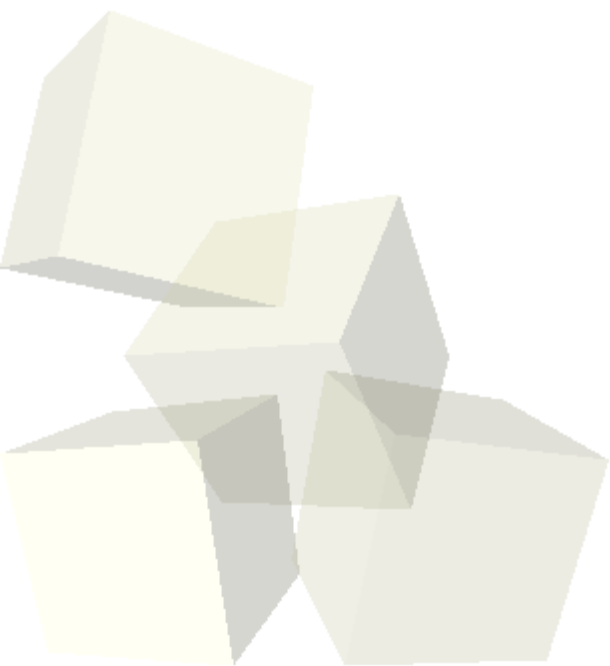


# Comunicação em Grupo (Multicast)



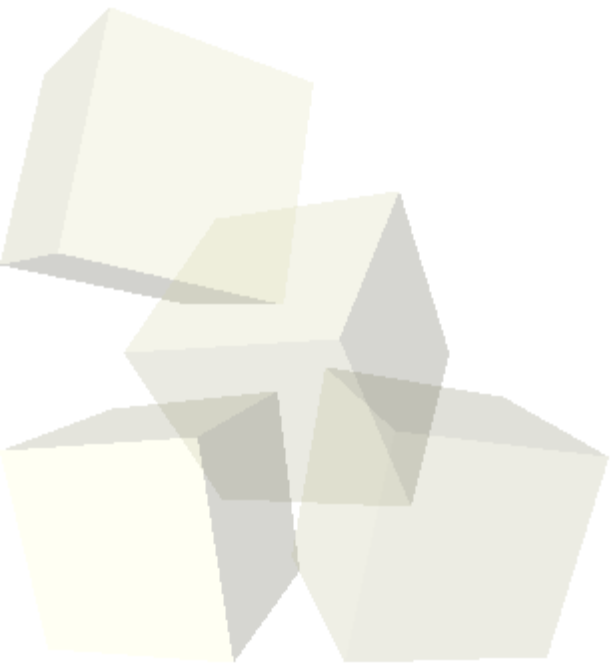


# Internacionalização - i18n





# Reflection API





# Java/IDL (Usando Corba)

