Programação por Objectos

Java

Parte 4: Classes de ambiente

Pacote java.lang

- O **pacote java.lang** é automaticamente importado:
  - Interfaces:
    - **`Cloneable`**
    - **`Runnable`**
  - Classes:
    - **`Class`** e **`Object`**
    - **`Boolean`**, **`Number`** (e subclasses), **`Character`**, **`Void`**
    - **`Math`**
    - **`Process`**, **`Thread`**, **`System`** e **`Runtime`**
    - **`String`** e **`StringBuffer`**
    - **`Throwable`** e **`Exception`** (e subclasses)

Classe `Object` (1)

- A classe `Object` é a raíz na hierarquia de herança de qualquer classe em Java.
  - Todas as classes estendem directa ou indirectamente `Object`.

Classe `Object` (2)

- **Métodos da classe `Object`**:
  - **`public int hashCode()`**
    Retorna valor da função de dispersão.
  - **`public String toString()`**
    Retorna cadeia de caracteres que descreve o objecto.
  - **`public boolean equals(Object obj)`**
    Retorna igualdade entre objectos.

    Nota:
    - **Igualdade entre objectos**: duas referências para o mesmo objecto.
    - **Equivalência entre objectos**: dois objectos com o mesmo estado (mesmo valor dos atributos).

Classe `Object` (3)

- Ambos os métodos `equals` e `hashCode` devem ser redefinidos se o programador pretende oferecer, em vez de igualdade, equivalência entre objectos.
  - **Por omissão, `equals` implementa igualdade entre objectos** (objectos distintos devolvem `false`).

    Nota: O operador == e != retorna sempre igualdade entre objectos. Ou seja, se o programador redefinir o método `equals` continua a poder testar igualdade entre objectos através do auxílio do operador == e !=.

  - **Por omissão, dois objectos distintos devolvem um `hashCode` diferente**.

Classe `Object` (4)

- **Se o método `equals` for redefinido para implementar equivalência entre objectos, então o método `hashCode` também deve ser redefinido concordantemente, ou seja, deve ser redefinido por forma a que dois objectos equivalentes devolvam o mesmo código de dispersão.**
  - O método `hashCode` é chamado automaticamente quando referências do objeto forem usadas em coleções do tipo *hash* (por exemplo, `Hashtable`, `HashMap`).
  - O método `equals` é usado como critério de desempate, portanto, se redefinir `hashCode` deve redefinir concordantemente `equals`.

Classe `Object` (5)

- Normalmente as classes redefinem o método `toString`.
- O método tem várias utilizações:
  - Depuração.
  - Geração de mensagem de apresentação.

Classe `Object` (5)

- **Métodos da classe `Object` (cont)**:
  - **`protected void finalize()`**
    Chamado pelo *garbage collector* quando o objecto deixa de ser referenciado.

    Nota: No Java, um objecto existe enquanto for referenciado. O *garbage collector* destrói objectos não referenciados.

  - **`protected Object clone()`**
    `throws CloneNotSupportedException`
    Cria cópia integral do objecto, que passa a ser autónomo. Contudo, se a classe deste `Object` não implementa a interface `Cloneable` então é lançada a excepção `CloneNotSupportedException`.

    Nota: Para qualquer objecto `obj`, tem-se

```
obj.clone() != obj;
```

Tipos primitivos (1)

- **Tipos de dados primitivos**:
  - **boolean** 1-bit (`true` ou `false`)
  - **char** 16-bit Unicode UTF-16 (sem sinal)
  - **byte** 8-bit inteiro com sinal
  - **short** 16-bit inteiro com sinal
  - **int** 32-bit inteiro com sinal
  - **long** 64-bit inteiro com sinal
  - **float** 32-bit IEEE 754 vírgula flutuante
  - **double** 64-bit IEEE 754 vírgula flutuante

Tipos primitivos (2)

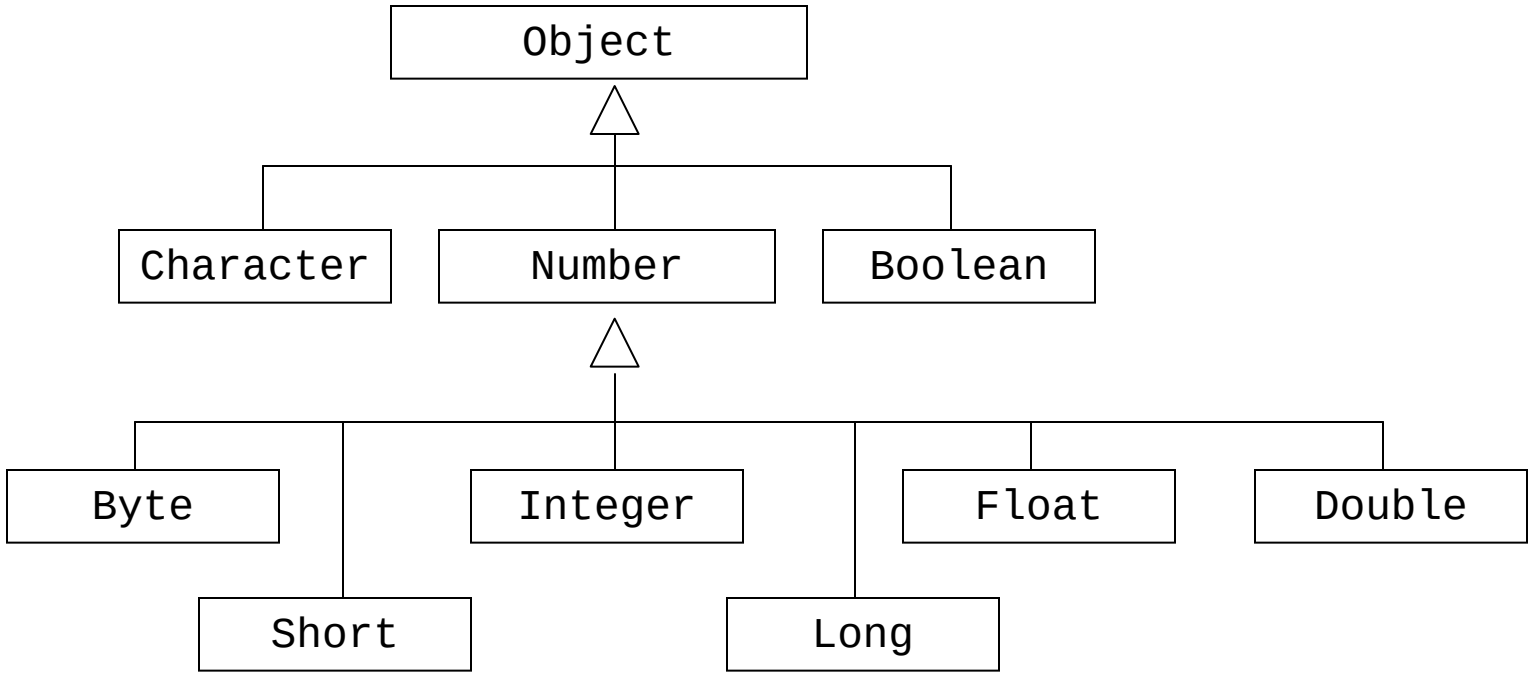
- No Java, para cada tipo primitivo existe no pacote `java.lang` uma **classe de embrulho** correspondente.
- Estas classes de embrulho, **`Boolean`**, **`Character`**, **`Byte`**, **`Short`**, **`Integer`**, **`Long`**, **`Float`** e **`Double`** definem constantes e métodos úteis.
- Os tipos primitivos de dados oferecem:
  - Acesso mais eficiente do que o acesso a objectos.
  - Ocupam sempre o mesmo espaço, independente da máquina onde corre o programa.

Tipos primitivos (3)

- **Conversão de tipos**:
  - O Java efectua **conversão implícita** de tipos primitivos, na ordem: `byte->short->int->long->float->double`.
  - Expressão envolvendo tipos distintos resulta num valor do tipo superior (por exemplo, `5+3.0` resulta no valor `8.0`).
  - Quando a conversão implícita não é possível, normalmente uma **conversão explícita** (*casting*) é usada (por exemplo, conversão de `float` para `int` elimina parte fraccionária e `(int)-72.3` resulta no valor `-72`).

Classes de embrulho (1)

- **Hierarquia das classes de embrulho**:



Classes de embrulho (2)

- As instâncias de uma classe de embrulho contêm um valor do correspondente tipo primitivo.
- O Java disponibiliza **conversão automática** entre tipo primitvo e correspondente classe de embrulho:
  - **Embrulhar** (*boxing*)*:* converter tipo primitivo para classe de embrulho.
  - **Desembrulhar** (*unboxing*): converter classe de embrulho para tipo primitivo.

```
Integer val = 3;
```

Necessário conhecer apenas a semântica.

Classes de embrulho (3)

- No que se segue, `Type` é usado para a classe de embrulho correspondente ao tipo primitivo `type`.
- **Todas as classes de embrulho possuem os seguintes métodos**:
  - **`public static Type valueOf(type t)`**
    retorna um objecto do tipo `Type` com o valor `t`.
  - **`public static Type valueOf(String str)`**
    retorna um objecto do tipo `Type` com o valor parsado de `str` (excepto para Character).
  - **`public type typeValue()`**
    retorna o valor do tipo primitivo `type` correspondente ao objecto de embrulho.

```
Integer.valueOf(6).intValue();
```

Necessário conhecer apenas a semântica.

Classes de embrulho (4)

- **`public static type parseType(String str)`**
  converte a string `str` para um valor do tipo primitivo `type`.
- **`public static String toString(type val)`**
  retorna a string representativa do valor `val` de tipo primitivo `type`.

- **Todas as classes de embrulho, à excepção da classe `Boolean`, definem três atributos**:
  - **`public static final type MIN_VALUE`**
    o valor mínimo representável pelo tipo de dados `type`.
  - **`public static fine type MAX_VALUE`**
    o valor máximo representável pelo tipo de dados `type`.
  - **`public static final int SIZE`**
    o número de bits usado para representar um valor deste tipo.

Necessário conhecer apenas a semântica.

Classes de embrulho (5)

Tipo primitivo	Classe de embrulho	Recolha de valor
`boolean`	`Boolean`	`booleanValue()`
`char`	`Character`	`charValue()`
`byte`	`Byte`	`byteValue()`
`short`	`Short`	`shortValue()`
`int`	`Integer`	`intValue()`
`long`	`Long`	`longValue()`
`float`	`Float`	`floatValue()`
`double`	`Double`	`doubleValue()`

Nota:
nem todos os nomes coincidem!

```
Integer I = new Integer(3);
int i = I.intValue(); // i passa a ter 3
String srt_i = Integer.toString(i); // str_i passa a ter "3"
```

Necessário conhecer apenas a semântica.

Classe `Character` (1)

Métodos estáticos	Descrição
`boolean isLowerCase(char)`	Indica se é minúscula
`boolean isUpperCase(char)`	Indica se é maiúscula
`boolean isDigit(char)`	Indica se é dígito decimal
`boolean isSpace(char)`	Indica se é `'\t'`, `'\n'`, `'\f'` ou `' '`
`char toLowerCase(char)`	Converte para minúscula
`char toUpperCase(char)`	Converte para maiúscula
`int digit(char,int)`	Valor numérico na base indicada

```
char c = Character.toUpperCase('g');
```

Necessário conhecer apenas a semântica.

Classe `Character` (2)

- **A classe `Character` possui ainda o seguinte método**:
  - **`public static int getType(char)`**
    retorna o código do caractere Unicode, que pode ser uma das seguintes contantes:
    - **`CURRENCY_SYMBOL`**
    - **`LOWERCASE_LETTER`**
    - **`UPPERCASE_LETTER`**
    - **`MATH_SYMBOL`**
    - **`SPACE_SEPARATOR`**
    - **`...`**

Necessário conhecer apenas a semântica.

Classes `Byte`, `Short`, `Integer` e `Float`

- **As classes `Byte`, `Short`, `Integer` e `Float` possuem ainda o seguinte método**:
  - **`public static type parseType(String str, int radix)`**
    converte a string `str` para um valor do tipo primitivo `type` segundo a base indicada em `radix` (decimal, por omissão).

Operadores (1)

- **Operadores aritméticos**:
  - `+` adição
  - `-` subtracção
  - `*` multiplicação
  - `/` divisão
  - `%` resto divisão inteira

- Os operadores aritméticos podem ser aplicados a qualquer tipo primitivo numérico e a caracteres.

Operadores (2)

- **Operadores de incremento/decremento**:
  - **++** incremento
  - **- -** decremento

- Os operadores de incremento/decremento podem ser aplicados a tipos primitivos numéricos e a caracteres (próximo/anterior código Unicode).

Operadores (3)

```
int var = 5;
```

Instrução	Resultado no terminal	Valor após instrução
`System.out.println(var++);`	5	6
`System.out.println(++var);`	6	6
`System.out.println(var--);`	5	4
`System.out.println(--var);`	4	4
`System.out.println(var%3);`	2	5

Operadores (4)

- **Operadores relacionais**:
  - `>` maior que
  - `>=` maior ou igual a
  - `<` menor que
  - `<=` menor ou igual a
- **Operadores de igualdade**:
  - `==` igual a
  - `!=` diferente de

- Os operadores relacionais e de igualdade devolvem um valor Booleano, e podem ser aplicados aos tipos primitivos numéricos e a caracteres.

Operadores (5)

- Os operadores de igualdade podem ser aplicados a tipos primitivos Booleanos.
- Os operadores de igualdade podem ainda ser usados para testar igualdade entre referências:
  - Esta igualdade refere-se a uma igualdade entre objectos e não a uma equivalência entre objectos:
    - **Igualdade entre objectos**: duas referências para o mesmo objecto.
    - **Equivalência entre objectos**: dois objectos com o mesmo estado (mesmo valor dos atributos).

Operadores (6)

- **Operadores lógicos**:
  - `!` negação
  - `&` conjunção
  - `|` disjunção
  - `^` disjunção exclusiva
  - `&&` conjunção condicional (com *lazzy evaluation*)
  - `||` disjunção condicional (com *lazzy evaluation*)

- Os operadores lógicos combinam expressões Booleanas e resultam em valores Booleanos.

Operadores (7)

- O operador **`instanceof`** avalia o tipo (classe ou intrface) de uma referência:
  - **Ref `instanceof` Ident**
    verifica se a referência Ref é do tipo Ident.

Operadores (8)

- **Operadores de bits**:
  - `&` conjunção (AND)
  - `|` disjunção (OR)
  - `^` disjunção exclusiva (XOR)
  - `~` negação
  - `<<` deslocamento para a esquerda, com 0's à direita
  - `>>` deslocamento para a direita, preservando sinal
  - `>>>` deslocamento para a direita, com 0's à esquerda

- Os operadores de bits podem ser usados em tipos primitivos inteiros, incluindo caracteres.

Operadores (9)

- O **operador condicional ? :** devolve uma de duas expressões dependendo da avaliação de uma expressão Booleana:
  - **Expr-Bool ? Expr1 : Expr2**
    se a expressão booleana Expr-Bool for verdadeira devolve Expr1 senão devolve Expr2.

Operadores (10)

- **Operadores de atribuição**:

  = atribuição

  op= atribuição composta

- O operando à esquerda dos operadores de atribuição deve ser sempre uma variável. O operando à direita é uma expressão.
- O operador op pode ser qualquer operador aritmético, lógico ou de bits.

Operadores (11)

- **Operador para concatenação de strings**: **+**

```
String s1 = “boo”;
String s2 = s1+”hoo”;
s2 += “!”;
System.out.println(s2);
```

- O **operador new** cria uma instância duma classe ou duma tabela.

Operadores (12)

- **Prioridade dos operadores** (máxima para mínima):

1.	Operadores unários	`++ -- + - ~ !`
2.	Criação ou *cast*	`new (type)`
3.	Multiplicativos	`* / %`
4.	Aditivos	`+ -`
5.	Deslocamento	`<< >> >>>`
6.	Relacional	`< > >= <= instanceof`
7.	Igualdade	`== !=`
8.	Conjunção	`&`
9.	Disjunção exclusiva	`^`
10.	Disjunção	`|`
11.	Conjunção condicional	`&&`
12.	Disjunção condicional	`||`
13.	Condicional	`?:`
14.	Atribuição	`= += -= *= /= %= >>= <<= >>>= &= ^= |=`

```
++x>3&&!b
```
é equivalente a
```
((++x)>3)&&(!b)
```

Operadores (13)

- Quando dois operadores com a mesma prioridade aparecem numa expressão, a **associatividade do operador** determina qual o operador que vai ser avaliado primeiro.
  - **Associatividade à esquerda**: `expr1 op expr2 op expr3` é equivalente a `(expr1 op expr2) op expr3`.
  - **Associatividade à direita**: `expr1 op expr2 op expr3` é equivalente a `expr1 op (expr2 op expr3)`.
- Os operadores de atribuição são associativos à direita. Todos os restantes operadores binários são associativos à esquerda.
- O operador condicional é associativo à direita.

Tabelas (1)

- Uma **tabela** (*array*) é um objecto autónomo contendo um número fixo de células, todas contendo dados do mesmo tipo base.
  - As tabelas são objectos que estendem implicitamente a classe `Object`.
  - Os tipos base podem ser primitivos ou referências (incluindo referências para outras tabelas).
  - J2SE disponibiliza contentores de tabelas de capacidade variável, por exemplo, `Vector`, `Stack`, ...

Tabelas (2)

Sintaxe

Tipo_base Ident [ ] = new Tipo_base [comprimento]

- A dimensão da tabela é omitida na sua declaração, sendo apenas dada quando é criada com o operador `new`.
- A dimensão de uma tabela é fixada na sua criação e não pode ser modificada.
- Os parêntesis rectos na declaração da tabela podem ser colocados depois do Tipo_base (Tipo_base [ ] Ident em vez de Tipo_base Ident [ ]).

Tabelas (3)

- **Tabelas multidimensionais**:
  - Declaradas com vários [ ].
  - Instanciação apenas exige a primeira dimensão (mais à esquerda).
  - Especificar mais do que uma dimensão poupa em número de operadores `new` a usar.

```
float[][] mat = new float[4][4];
```

```
float[][] mat = new float[4][];
for (int i=0; i < mat.length; i++)
   mat[i] = new float[4];
```

Tabelas (4)

- Numa tabela multidimensional, cada tabela pode ter uma dimensão diferente, o que permite criar tabelas multidimensionais de vários tipos:
  - Triangulares
  - Rectangulares
  - …

Tabelas (5)

- A dimensão da tabela é armazenada no atributo `public final int length`
- O acesso aos elementos da tabela é feito depois de instanciada, com cada um dos índices indicado entre parêntesis rectos (Ident[pos]).
- O primeiro elemento da tabela tem índice 0, e o último elemento da tabela tem índice `length-1`.
- O acesso a índices fora da gama gera excepção `ArrayIndexOutOfBoundException`.

```
int[] ia = new int[3];
... //inicialização de ia
for (int i=0; i < ia.length; i++)
   System.out.println(i + ": " + ia[i]);
```

Tabelas (6)

- Uma tabela pode ter dimensão `0`.
  - Uma referência para uma tabela de dimensão `0` é diferente de uma referência para `null`.
  - Útil no retorno de métodos.
- Os qualificadores usuais podem ser usados na declaração de atributos/variáveis do tipo tabela.
  - Os qualificadores aplicam-se ao atributo/variável tabela e não aos seu elementos.
  - Quando uma tabela é declarada **final** significa que a sua referência não pode ser modificada após a sua inicialização. Não significa que os seus elementos não podem ser modificados!

Tabelas (7)

- **Inicialização de tabelas**:
  - Quando uma tabela é criada, cada elemento é inicializado com um valor por omissão (dependendo do seu tipo).
  - A inicialização pode ser feita de duas formas:
    1. Listagem de todas a células entre { }:
       - Não é necessário criar a tabela explicitamente com o operador new.
       - A dimensão da tabela é determinada pelo número de elementos incializados.

```
String[] animais = {“Leão”, “Tigre”, “Urso”};
```

Tabelas (8)

- Pode usar-se o operador `new` explicitamente, mas nesse caso a dimensão tem de ser omitida (porque, mais uma vez, esta é determinada pela lista de elementos inicializados).

```
String[] animais = new String[]{“Leão”, “Tigre”, “Urso”};
```

- É possível que o último elemento de uma lista de inicialização seja seguido por uma vírgula.
- Tabelas multidimensionais podem ser inicializadas por aninhamento de listagem.

Tabelas (8)

2. Por atribuição directa dos seus elementos:

```
String[] animais = new String[3];
animais[0] = “Leão”;
animais[1] = “Tigre”;
Animais[2] = “Urso”;
```

Necessário conhecer apenas a semântica.

Tabelas (9)

- **A classe `System` oferece um método que permite copiar os valores de uma tabela para outra**:
  - `public static void arraycopy (Object src, int srcPos, Object dst, int dstPos, int count)`

    copia o conteúdo da tabela `src`, começando em `src[srcPos]`, para a tabela `dst`, começando em `dst[dstPos]`; são copiados exactamente `count` elementos.

Tabelas (10)

- **As tabelas como extensão da classe `Object`**:
  - As tabelas não definem métodos próprios, apenas herdam os métodos da classe `Object`.
    - O método **`equals`** é sempre baseado em igualdade e não em equivalência.
  - O método **`deepEquals`** da classe utilitária **`java.util.Arrays`** permite comparar tabelas por equivalência.
    - Verifica a equivalência entre dois `Object` recursivamente, tendo em consideração equivalência de tabelas multidimensionais.

Tabelas (11)

```
String[] animais = {"Leão", "Tigre", "Urso", };
String[] aux = new String[animais.length];
System.arraycopy(animais, 0, aux, 0, animais.length);

for (int i=0; i<aux.length; i++)
   System.out.println(i + ": " + aux[i]);

System.out.println(aux.equals(animais));
System.out.println(java.util.Arrays.deepEquals(aux, animais));
```

No terminal é impresso

```
0: Leão
1: Tigre
2: Urso
false
true
```

Tabelas (12)

```
int[][] trianguloPascal1 = {
   { 1 },{ 1, 1 },{ 1, 2, 1 },{ 1, 3, 3, 1},{ 1, 4, 6, 4, 1}};

int[][] trianguloPascal2 = new int[5][];
trianguloPascal2[0] = new int[]{ 1 };
trianguloPascal2[1] = new int[]{ 1, 1 };
trianguloPascal2[2] = new int[]{ 1, 2, 1 };
trianguloPascal2[3] = new int[]{ 1, 3, 3, 1 };
trianguloPascal2[4] = new int[]{ 1, 4, 6, 4, 1 };

System.out.println(trianguloPascal1.equals(trianguloPascal2));
System.out.println(java.util.Arrays.deepEquals(
   trianguloPascal1, trianguloPascal2));
```

No terminal é impresso

```
false
true
```

Classe `String` (1)

- Uma **cadeia de caracteres** (*string*) é um objecto autónomo/pré-definido contendo sequências de caracteres.
  - As cadeias de caracteres são instâncias da classe **`String`**.
  - As cadeias de caracteres não podem ser alteradas, pelo que têm de ser determinadas na instanciação.
    - No entanto, o identificador pode mudar a referência para outra cadeia de caracteres.
    - Se for mesmo necessário alterar o conteúdo, usar a classe **`StringBuffer`**.
  - A sequência de caracteres é delimitada por aspas (**`"`** e **`"`**).
  - O operador **`+`** concatena duas cadeias de caracteres.

Classe `String` (2)

- **Construção de cadeias de caracteres**:
  - Implicitamente, através do uso de um literal, ou com o auxílio dos operadores `+` e `+=` sobre dois objectos `String`.
  - Explicitamente, com o auxílio do operador `new` (apenas alguns dos construtores, ver documentação):
    - `public String()`
      Cria uma nova cadeia de caracteres vazia (“”).
    - `public String(String valor)`
      Construtor por cópia, cria uma nova cadeia de caracteres com o mesmo valor que a cadeia de caracteres recebida.

Classe `String` (3)

- **`public String (char[] valor)`**
  Cria uma cadeia de caracteres cujo valor representa a sequência de caracteres da tabela de caracteres recebida.
- **`public String(char[] valor, int pos, int num)`**
  Cria uma cadeia de caracteres, que contém a sequência de caracteres, a partir da posição pos, da tabela recebida. Apenas num elementos são considerados.

Classe `String` (4)

```
String s1 = "Bom";
String s2 = s1 + " dia";
String vazia = "";
```

```
String vazia = new String();
String s1 = new String(“Bom dia”);
char valor[] = {‘B’,‘o’,‘m’,‘ ’,‘d’,‘i’,‘a’};
String s2 = new String(valor);
String s3 = new String(valor,4,3);
```

Necessário conhecer apenas a semântica.

Classe `String` (5)

- **Métodos públicos da classe `String`:**
  1. Propriedades da cadeia:
  - **`int length()`**
    Comprimento da cadeia.
  - **`int compareTo(String str)`**
    Devolve um inteiro que é menor que 0, igual a 0, ou maior que 0, quando a cadeia de carateres na qual o método foi chamado é menor que `str`, igual a `str`, ou maior que `str`. A ordem usada é a ordem dos caracteres Unicode.

Necessário conhecer apenas a semântica.

Classe `String` (6)

2. Acesso a partes da cadeia:

- **`char charAt(int)`**
  Caractere na posição.
- **`char[] toCharArray()`**
  Devolve tabela de caracteres.
- **`int indexOf(char)`**
  Primeira posição em que ocorre caractere.
- **`int lastIndexOf(char)`**
  Última posição em que ocorre caractere.
- **`String substring(int,int)`**
  Subcadeia entre posições.
- **`String substring(int)`**
  Subcadeia a partir de posição.

O primeiro caractere numa cadeia tem posição 0.

Necessário conhecer apenas a semântica.

Classe `String` (7)

3. Alterações em cadeias:

- **`String replace(char oldChar,char newChar)`**
  Obtém nova cadeia alterando todas as ocorrências do 1º caractere.
- **`String toLowerCase()`**
  Obtém nova cadeia em minúsculas.
- **`String toUpperCase()`**
  Obtém nova cadeia em maiúsculas.
- **`String trim()`**
  Obtém nova cadeia sem espaços em branco no início e no fim.
- **`String concat(String)`**
  Obtém nova cadeia estendida no fim com o parâmetro.

Classe `String` (8)

```
String s = "/home/asmc/aula-po.ppt";
...
int inicio, fim;
inicio = s.lastIndexOf('/');
fim = s.lastIndexOf('.');
System.out.println(s.substring(inicio+1,fim));
```

No terminal é impresso `aula-po`

Necessário conhecer apenas a semântica.

Classe `String` (9)

- **Métodos de conversão entre cadeia de caracteres e tipo primitivo**:

Tipo	Para cadeia	De cadeia
boolean	`String.valueOf(boolean)`	`Boolean.parseBoolean(String)`
int	`String.valueOf(int)`	`Integer.parseInt(String,int)`
long	`String.valueOf(long)`	`Long.parseLong(String,int)`
float	`String.valueOf(float)`	`Float.parseFloat(String)`
double	`String.valueOf(double)`	`Double.parseDouble(String)`

Necessário conhecer apenas a semântica.

Classe `String` (10)

- **Conversão entre cadeias de caracteres e tabela de carateres**:
  - A classe `String` disponibiliza a conversão de cadeia de caracteres para tabela de caracteres:
    - **`public char[] toCharArray()`**
  - A classe `System` disponibiliza a conversão de tabela de caracteres para cadeia de caracteres:
    - **`public static void arraycopy( Object src, int srcPos, Object dst, int dstPos, int count)`**
      copia o conteúdo da tabela `src`, começando em `src[srcPos]`, para a tabela `dst`, começando em `dst[dstPos]`; serão copiados exactamente `count` elementos.

Classe `String` (11)

```
public static String squeezeOut(String from, char toss){
   char chars[]=from.toCharArray(); // transfere cadeia para tabela
   int len=chars.length;            // recolhe comprimento

   for (int i=0; i<len; i++) {
      if (chars[i]==toss) {
         --len; // cadeia final tem menos 1 caractere
         System.arraycopy(
            chars, i+1, chars, i, len-i); // desloca parte direita
          --i; // para continuar a procura na mesma posição
      }
   }
   return new String(chars, 0, len);
}
```

`System.out.println(squeezeOut("Programação por Objetos", 'o'));`

imprime no terminal `Prgramaçã pr Objets`

Classe `String` (12)

- **A classe `String` redefine o método `equals` de `Object` para devolver `true` sse duas cadeias de caracteres têm o mesmo conteúdo.**
- **Também redefine o método `hashCode`, de tal forma que duas cadeias de caracteres com o mesmo conteúdo tem o mesmo `hashCode`.**

```
String s1 = new String("abc"), s2 = "abc";
```

Expressão	Resultado	Justificação
`s1==s2`	`false`	Objectos distintos
`s1.equals(s2)`	`true`	Valores iguais

Necessário conhecer apenas a semântica.

Classe Math (1)

- Classe disponibilizada no J2SE, define constantes matemáticas e implementa métodos de cálculo.

Constante	Significado
PI	π
E	e

```
System.out.println("Pi=" + Math.PI);
```

Necessário conhecer apenas a semântica.

Classe Math (2)

- Todos os métodos são estáticos
  (num é `int`, `long`, `float` ou `double`)

Métodos	Descrição
`double sin(double)`	Seno
`double powers(double,double)`	Potência
`num abs(num)`	Valor absoluto
`num max(num,num)`	Valor máximo
`num min(num,num)`	Valor mínimo
`int round(float)` `long round(double)`	Arredondamento
`double sqrt(double)`	Raiz quadrada