



METODOLOGIA

- › Interprete o documento calmamente e com atenção.
- › Acompanhe a execução do exercício no seu computador.
- › Não hesite em consultar o formador para o esclarecimento de qualquer questão.
- › Não prossiga para o ponto seguinte sem ter compreendido totalmente o ponto anterior.
- › Caso seja necessário, execute várias vezes o exercício até ter compreendido totalmente o processo.

CONTEÚDO PROGRAMÁTICO

1. [Introdução](#)
2. [StringBuilder](#)
3. [ArrayList](#)
4. [Date](#)

1. Introdução

Temos vindo a construir vários tipos de classes que fornecem funcionalidades específicas. Mas existem já no Java classes que fornecem soluções simples para problemas comuns que surgem durante o desenvolvimento de programas. Observámos isso mesmo para as **Strings**, em que a respetiva classe disponibiliza já vários métodos que nos permitem fazer várias operações típicas relacionadas com texto. Vamos agora ver outros exemplos.

2. StringBuilder

Já foi referido que o tipo String no Java é um pouco diferente do resto dos tipos. Este tem a particularidade de ser imutável, o que significa que cada vez que é feita uma alteração numa variável do tipo String é criada uma nova String.

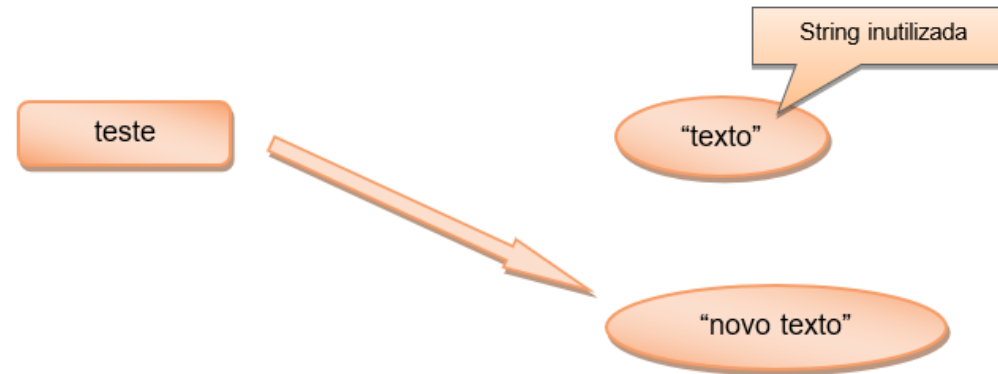
Considere as duas seguintes linhas de código:

```
String teste = "texto";  
teste = "novo texto";
```

Na primeira linha de código existe a variável `teste` com o valor "texto":



Na segunda linha de código, a **String** `texto` encontra-se inutilizada em memória e a variável **teste** tem um novo valor:



Ao contrário do que somos tentados a pensar, a **String** `teste` não é alterada da primeira para a segunda linha de código, mas sim criada uma nova **String** com o texto "novo texto" e guardada na variável `teste`. Isto não se passa com a classe `StringBuilder`, que serve também para guardar e manipular **Strings**, o que a torna um pouco mais eficiente e flexível pois o conteúdo da variável é modificado internamente.

Vamos criar um programa para testar esta situação e explorar um pouco mais a classe `StringBuilder`.

› **Crie** a classe `StringBuilderTestes.java` e **introduza** o código:

```
import java.util.Scanner;
```

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

        Scanner teclado = new Scanner(System.in);
        System.out.println("Insira uma frase");
        StringBuilder texto = new StringBuilder(teclado.nextLine());

        System.out.println("A frase que escreveu e - " + texto);
    }
}
```

› Compile e teste a classe

📄 O funcionamento é muito semelhante ao de uma **String**, existindo também uma coleção de métodos para manipular internamente os dados:

- `append` – Permite juntar ao texto um determinado valor (muitas das vezes também texto).
- `insert` – Permite inserir um ou vários caracteres em determinada parte do texto.
- `setCharAt` – Através deste método é possível definir o caractere de uma determinada posição.
- `reverse` – Inverte o texto.
- `delete` – Remove um ou vários caracteres a partir de uma posição no texto.

📄 Nenhum dos métodos acima existe na classe `String`.

Vamos ver um exemplo em que utilizamos alguns destes métodos.

› Reescreva o código do método `main` da classe `StringBuilderTestes` de acordo com o seguinte:

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

    Scanner teclado = new Scanner(System.in);
    System.out.println("Insira uma frase");
    StringBuilder texto = new StringBuilder(teclado.nextLine());

    System.out.println("A sua frase ao contrario - " + texto.reverse());
    texto.reverse(); //reverter ao modo normal
}
```

```

System.out.println("Escreva uma palavra para juntar ao fim da frase");

String palavra = teclado.nextLine();

//junta a string palavra ao fim do texto do stringBuilder
texto.append(" " + palavra);
System.out.println("A sua frase com a palavra no fim - " + texto);

//transformar o primeiro caratere em maiusculo
char caratereMaiusculo = Character.toUpperCase(texto.charAt(0));
texto.setCharAt(0, caratereMaiusculo);

for (int i = 1; i < texto.length(); ++i){
    if (texto.charAt(i-1) == ' '){
        //se o caratere atrás do corrente e um espaço
        //transforma o caratere corrente em maiúsculo
        caratereMaiusculo = Character.toUpperCase(texto.charAt(i));
        texto.setCharAt(i, caratereMaiusculo);
    }
}

System.out.println("A sua string com as iniciais em maiusculas - " + texto);
}

```

› **Compile e teste** o código

› **Experimente** alguns valores de forma a perceber o efeito do programa

O ciclo `for` visa transformar o texto de forma a que todas as palavras comecem em maiúsculas. Para este efeito é percorrido todo o texto, letra a letra, e avaliado se a letra anterior é um espaço. Caso isso aconteça transforma-se essa letra em maiúscula. Repare como o ciclo começa na posição **1**:

```
for (int i = 1 ; i < texto.length(); ++i){
```

Isto porque o primeiro caratere é sempre modificado para maiúsculas. No código do `for` é testado se o caratere anterior ao corrente é um espaço:

```
if (texto.charAt(i-1) == ' '){
```

Quando isto acontece é transformado na sua versão maiúscula e guardado na posição correta. O método para converter apenas um caratere em maiúscula é um método estático existente na classe

`Character`, de nome `toUpperCase`.

```
Character.toUpperCase(carater);
```

Como vimos anteriormente, um método estático é utilizado sem a necessidade de criação de um novo objeto, uma vez que se refere à própria classe e não a uma instância.

3. ArrayList

A classe `ArrayList` representa um array do tipo que nós especificarmos. Mas qual a diferença entre esta classe e um array normal? A primeira diferença, e também a mais relevante, deve-se ao facto de o tamanho não ser fixo. Até então, para criar um array de inteiros usámos sempre uma instrução semelhante:

```
int[] arraynumeros = new int[10];
```

Como podemos verificar, é necessário especificar um tamanho quando iniciamos um array (10 no exemplo anterior). Isto pode ser restritivo em várias situações. Imaginemos que pretendemos um array de "Carros" para registar cada carro que passa numa portagem, mas não temos qualquer perceção do tamanho que este deverá atingir. Qual o tamanho a definir? Utilizando esta notação de array, teríamos que tentar prever um número máximo, correndo o risco do programa dar erro quando se ultrapassasse o tamanho definido. Através do `ArrayList` não existe esta limitação.

Vamos ver a sua construção:

```
ArrayList<Integer> arraynumeros = new ArrayList<Integer>();
```

Para esta classe, os tipos que podemos definir são diferentes dos que temos usado, representando, no entanto, a mesma informação. O tipo encontra-se dentro do sinal `<` e `>`. Tendo isto em mente para construir um `ArrayList` de **Strings** deve utilizar-se a seguinte instrução:

```
ArrayList<String> listatextos = new ArrayList<String>();
```

Repare como não é especificado nenhum tamanho. Isto porque este `ArrayList` **permite-nos adicionar elementos sem ter um tamanho fixo**. Ao contrário de um array normal, disponibiliza vários métodos com os quais é possível manipular o seu conteúdo. Os principais são os de inserção e remoção de elementos. Para inserir um elemento no objeto `arraynumeros`, exemplificado em cima, utiliza-se o método `add`:

```
arraynumeros.add(5);
```

Vamos exemplificar criando outra classe de testes para a classe `Carro`.

- **Crie** a classe `TesteCarro2.java` no package `Carro`

➤ **Insira** o seguinte código:

```

package carro;
import java.util.ArrayList;
import java.util.Scanner;

public class TesteCarro2 {
    public static void listarCarros(ArrayList<Carro> lista){
        System.out.println("Lista de Carros:");

        for(int i = 0; i < lista.size(); ++i){
            Carro carrocorrente = lista.get(i);
            carrocorrente.mostrar();
        }
    }

    public static void main(String[] args) {
        Scanner teclado = new Scanner(System.in);
        int opcao = 0;
        ArrayList<Carro> lista = new ArrayList<Carro>();

        while(opcao != 3){
            System.out.println("\n1 - Inserir Carro");
            System.out.println("2 - Listar Carros");
            System.out.println("3 - Sair");
            opcao = teclado.nextInt();

            switch(opcao){
                case 1:
                    System.out.println("Escolha o tipo de Carro que pretende inserir");
                    System.out.println("1 - Jipe");
                    System.out.println("2 - Descapotavel");
                    int tipocarro = teclado.nextInt();

                    teclado.nextLine();

                    System.out.println("Insira a marca");
                    String marca = teclado.nextLine();

                    System.out.println("Insira a matricula");
                    String matricula = teclado.nextLine();

```

```

        System.out.println("Insira a cor");
        String cor = teclado.nextLine();

        if(tipocarro == 1){
            lista.add(new Jipe(matricula,marca,cor,false));
        }
        else if(tipocarro == 2){
            lista.add(new Descapotavel(matricula,marca,cor));
        }

        break;
        case 2:
            listarCarros(lista);
            break;
    }
}
}
}

```

› Compile e teste a classe

Comecemos por perceber o código que foi inserido. Inicialmente são apresentadas na consola três opções: "Inserir Carro", "Listar Carros" e "Sair". Para inserir são pedidas as informações do carro e o seu tipo. Consoante o tipo, é criado o tipo de carro correspondente e adicionado à lista.

1

```

if(tipocarro == 1){
    lista.add(new Jipe(matricula, marca, cor, false));
}
else if(tipocarro == 2){
    lista.add(new Descapotavel(matricula, marca, cor));
}

```

A instrução usada está compactada em termos de sintaxe. A primeira **instrução de adição 1** poderia estar na seguinte forma:

```

Jipe jipetemp = new Jipe(matricula, marca, cor, false);
lista.add(jipetemp);

```

Neste caso, primeiro é criado o objeto com os dados inseridos pelo utilizador e, só de seguida é adicionado à lista de carros.

Para o método mostrar da listagem dos carros é utilizado um ciclo for que percorre o array de carros até ao fim:

```
for(int i = 0; i < lista.size(); ++i){
```

Note que neste caso, a expressão `lista.size()` devolve resultados diferentes consoante existem mais ou menos carros na lista. Para cada iteração do ciclo é obtido o respetivo carro e executado o método `mostrar`. Lembre-se que a informação apresentada por este método depende do tipo de carro.

Vamos agora adicionar a funcionalidade de remover carros:

› **Modifique** o código que se encontra destacado:

📄 Os blocos de código omitidos não sofrem qualquer alteração.

```
while(opcao != 4){
    System.out.println("\n1 - Inserir Carro");
    System.out.println("2 - Listar Carros");
    System.out.println("3 - Remover Carros");
    System.out.println("4 - Sair");

    opcao = teclado.nextInt();

    switch(opcao){
        case 1:
            .
            .
            .

        case 2:
            listarCarros(lista);
            break;
        case 3:
            System.out.println("Insira o numero do carro a apagar");
            int carroapagar = teclado.nextInt() - 1;

            if((carroapagar < lista.size()) && (carroapagar >= 0)){
                lista.remove(carroapagar);
            }

            break;
    }
}
```


> **Compile e execute** classe

Foi adicionada mais uma entrada no bloco `switch`, que corresponde à opção do menu para apagar carros. Esta interpreta o número introduzido e decrementa uma unidade.

```
int carroapagar = teclado.nextInt() - 1;
```

Isto é feito devido à clássica utilização de índices em arrays. Relembrando o que foi mencionado em módulos anteriores, um array de 10 posições vai de 0 a 9. Por este motivo, a primeira posição de um array é sempre zero. No entanto, para um utilizador este conceito é estranho, pelo que o deixamos introduzir o valor normal e ajustamos para o índice no array. Ex: o primeiro carro, no qual seria lógico pensarmos que era o carro 1, corresponde ao primeiro índice do array que é o 0. Para convertermos um no outro basta subtrair uma unidade.

A inserção é feita apenas após testar se o número do carro a apagar se encontra dentro dos índices disponíveis do array:


```
if((carroapagar < lista.size()) && (carroapagar >= 0)){
```

De seguida temos a tabela com os tipos disponíveis para utilizar num `ArrayList`:

Tipo original	Tipo a usar	Exemplo
byte	Byte	ArrayList<Byte> lista = new ArrayList<Byte>();
short	Short	ArrayList<Short> lista = new ArrayList< Short >();
int	Integer	ArrayList<Integer> lista = new ArrayList<Integer>();
long	Long	ArrayList<Long> lista = new ArrayList<Long>();
float	Float	ArrayList<Float> lista = new ArrayList<Float>();
double	Double	ArrayList<Double> lista = new ArrayList<Double>();
char	Character	ArrayList<Character> lista = new ArrayList<Character>();
boolean	Boolean	ArrayList<Boolean> lista = new ArrayList<Boolean>();

4. Date


À medida que desenvolvemos classes que representam situações da vida real, mais tarde ou mais cedo deparamo-nos com a necessidade de representar e guardar datas. Um exemplo disto seria guardar numa classe `Pessoa` a data de nascimento. O Java possui já a classe `Date` que contém métodos para efetuar operações sobre datas. No entanto, a maior parte dos métodos desta classe estão marcados como **deprecated** devido às atualizações.

 Um método **deprecated** está disponível na linguagem mas encontra-se descontinuado e deve deixar de ser utilizado. Significa que em versões futuras da linguagem poderá até deixar de existir.

Por este motivo é necessário usar ainda outra classe para o tratamento de datas. Trata-se da classe `Calendar` que, para além das datas, permite manipular definições de calendário.

O código para criar um objeto do tipo "Date" para a data de "01-01-2000" seria:


```
Calendar cal = Calendar.getInstance();
cal.set(2000, 0, 1);
Date data = cal.getTime();
```

 Repare que o mês começa em **0**, correspondendo a Janeiro.

Vamos utilizar a classe `Pessoa.java` para exemplificar a utilização desta classe:

› **Adicione** antes da declaração da classe as seguintes instruções:

```
import java.util.Calendar;
import java.util.Date;
```

 Sem estas importações não é possível de usar as classes `Date` e `Calendar` existentes no Java.

› **Acrescente** à classe `Pessoa` o seguinte atributo e método:

```
private Date datanascimento;

public void definirDataNasc(int dia, int mes, int ano){
    Calendar cal = Calendar.getInstance();
    cal.set(ano, mes-1, dia);
    datanascimento = cal.getTime();
}
```

Pode também ser vantajoso ter um método que através de uma **String** crie uma data. Esta ação de transformar um texto noutra tipo chama-se conversão ou *parsing*. Para podermos fazer o parse de uma data utilizamos ainda outra classe: `SimpleDateFormat`, que recebe o formato da data no seu construtor. Após construído um objeto deste tipo, podemos utilizar os seus métodos para mostrar uma data no formato definido ou extrair uma data a partir de uma **String**.

Precisamos agora de definir um método que converta numa data o texto introduzido pelo utilizador e a guarde no objeto.

➤ **Adicione** mais um método à classe `Pessoa`:

```
/**
 * Define a datanascimento convertendo uma string para a respetiva data
 * @param data texto que contem a data a interpretar
 * @throws ParseException
 */
public void definirDataNasc(String data) throws ParseException{
    SimpleDateFormat formato = new SimpleDateFormat("dd-MM-yyyy");
    datanascimento= formato.parse(data);
}
```

É o método `parse` que gera uma data através de uma **String**:

```
datanascimento = formato.parse(data);
```

A interpretação respeita o formato especificado na linha anterior. Este método `definirDataNasc` possui a palavra reservada do Java `throws` que iremos abordar no módulo de exceções.

Repare que este segundo método `definirDataNasc` é um *overload* do primeiro. Com estes dois métodos conseguimos agora definir a data de um objeto "pessoa" de duas formas diferentes. Precisamos agora de obter informações a partir desta data, por exemplo a idade da pessoa.

➤ **Adicione** outro método à classe `Pessoa`:

```
/**
 * Devolve a idade através da diferença da data corrente
 * e da data de nascimento
 * @return idade em anos
 */
public long obterIdade(){
    Date agora = new Date();
    long diferenca = agora.getTime() - datanascimento.getTime();
}
```

```
}  
    return diferenca/1000/60/60/24/30/12;  
}
```

Quando criamos uma data sem parâmetros no construtor, é assumida a data atual. Depois é obtida a diferença das datas, em **milissegundos**, através do método `getTime` já usado atrás:

```
long diferenca = agora.getTime() - datanascimento.getTime();
```

Com a diferença em milissegundos é necessário converter para anos, através de uma série de divisões. Dividindo o valor por 1000 ficamos com segundos, dividindo novamente por 60 transforma-se em minutos, outra vez por 60 gera horas e assim sucessivamente ate ficarem anos.

Vamos agora fazer o código de teste destes métodos:

➤ Crie uma classe `TestePessoa.java` e insira o código seguinte:

```
import java.text.ParseException;  
import java.util.Scanner;  
  
public class TestePessoa {  
    public static void main(String[] args) throws ParseException {  
        Scanner teclado = new Scanner(System.in);  
  
        System.out.println("Insira o seu nome");  
        String nome = teclado.nextLine();  
  
        System.out.println("Insira a sua morada");  
        String morada = teclado.nextLine();  
  
        System.out.println("Insira o seu telefone");  
        long telefone = teclado.nextLong();  
  
        Pessoa p = new Pessoa(nome, morada, telefone);  
  
        teclado.nextLine();  
        System.out.println("Insira a sua data de nascimento no formato dd-mm-aaaa");  
        String datanasc = teclado.nextLine();  
  
        p.definirDataNasc(datanasc);  
  
        long idade = p.obterIdade();  
        System.out.println("Voce tem " + idade + " anos");  
    }  
}
```

```
}  
}
```

- › **Implemente** um **construtor** para a classe `Pessoa` que receba um **nome**, uma **morada** e um **telefone**

Este construtor é utilizado na linha:

```
Pessoa p = new Pessoa(nome, morada, telefone);
```

- › Finalmente **teste** esta classe

📄 Em caso de alguma dúvida não hesite em solicitar a ajuda do formador.

Podemos também criar um construtor que recebe a data de nascimento simplificando a criação de um objeto "Pessoa".

- › **Adicione** o seguinte **construtor** à classe `Pessoa`:

```
public Pessoa(String nome_pessoa, String morada_pessoa, long telefone_pessoa, String datanascimento_pessoa) throws ParseException {  
    nome = nome_pessoa;  
    morada = morada_pessoa;  
    telefone = telefone_pessoa;  
  
    definirDataNasc(datanascimento_pessoa);  
}
```

- › Agora **altere** na classe `TestePessoa` as linhas destacadas:

```
System.out.println("Insira o seu telefone");  
long telefone = teclado.nextLong();  
  
teclado.nextLine();  
System.out.println("Insira a sua data de nascimento no formato dd-mm-aaaa");  
String datanasc = teclado.nextLine();
```

```
Pessoa p = new Pessoa(nome, morada, telefone, datanasc);
```

```
long idade = p.obterIdade();
```

› **Compile e teste** a classe

📄 Confirme que o resultado da execução continua igual.

📄 Ao definir o código da definição da data através do texto no construtor, facilitamos a construção de um objeto deste tipo.