Java SE 8 Programmer I

O guia para sua certificação Oracle Certified Associate

Casa do Código

SÉRIE CAELUM

GUILHERME SILVEIRA
MÁRIO AMARAL

Edição

Adriano Almeida
Vivian Matsui

Revisão

Bianca Hubert
Vivian Matsui

[2018]

Casa do Código

Livros para o programador

Rua Vergueiro, 3185 - 8º andar

04101-300 – Vila Mariana – São Paulo – SP – Brasil

www.casadocodigo.com.br

SOBRE O GRUPO CAELUM

Este livro possui a curadoria da Casa do Código e foi estruturado e criado com todo o carinho para que você possa aprender algo novo e acrescentar conhecimentos ao seu portfólio e à sua carreira.

A Casa do Código faz parte do Grupo Caelum, um grupo focado na educação e ensino de tecnologia, design e negócios.

Se você gosta de aprender, convidamos você a conhecer a Alura (www.alura.com.br), que é o braço de cursos online do Grupo. Acesse o site deles e veja as centenas de cursos disponíveis para você fazer da sua casa também, no seu computador. Muitos instrutores da Alura são também autores aqui da Casa do Código.

O mesmo vale para os cursos da Caelum (www.caelum.com.br), que é o lado de cursos presenciais, onde você pode aprender junto dos instrutores em tempo real e usando toda a infraestrutura fornecida pela empresa. Veja também as opções disponíveis lá.

ISBN

Impresso e PDF: 978-85-5519-031-5

EPUB: 978-85-5519-032-2

MOBI:978-85-5519-033-9

Você pode discutir sobre este livro no Fórum da Casa do Código: http://forum.casadocodigo.com.br/.

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

AGRADECIMENTOS

"Às famílias que me acolhem no dia a dia, Azevedo Silveira e Bae Song." — Guilherme Silveira

"Aos familiares, amigos e, especialmente, à minha esposa Gabriela. Amo todos vocês." — Mário Amaral

---

Escrever um livro é difícil, descrever pequenos detalhes de uma linguagem é um desafio maior do que poderíamos imaginar.

Fica um agradecimento ao Gabriel Ferreira, Márcio Marcelli, Leonardo Cordeiro e Alexandre Gamma, pelas valiosas revisões dos textos e exercícios. Agradecimento especial ao Leonardo Wolter, por sua revisão completa, além de diversas sugestões e melhorias.

Um abraço a todos da Caelum, do Alura e da Casa do Código, que nos incentivam na busca contínua de conhecimento com a finalidade de melhoria da qualidade de ensino e aprendizado de desenvolvimento de software no Brasil.

CERTIFICAÇÃO?

As certificações Java são, pelo bem ou pelo mal, muito reconhecidas no mercado. Em sua última versão, a principal certificação é feita em duas provas. Este livro vai guiá-lo por questões e assuntos abordados para a primeira prova, a Java SE 7 Programmer I (1Z0-803) ou sua versão Java SE 8 Programmer I (1Z0-808), de maneira profunda e desafiadora.

O livro vai percorrer cada tema, com detalhes e exercícios, para você chegar à prova confiante. Decorar regras seria uma maneira de estudar, mas não estimulante. Por que não compila? Por que não executa como esperado? Mais do que um guia para que você tenha sucesso na prova, nossa intenção é mostrar como a linguagem funciona por trás.

Ao terminar essa longa caminhada, você será capaz de entender melhor a linguagem, assim como poder dizer com exatidão os motivos de determinadas construções e idiomismos.

Faço a prova 7 ou 8?

A prova do Java 8 cobra conteúdo a mais do que a do Java 7. Se seu objetivo é aprender mais ou conhecer todos os detalhes da linguagem em sua versão mais recente, a resposta é clara: Java 8. Caso deseje tirar uma certificação, a versão não fará muita diferença.

O conteúdo aqui apresentado facilita o estudo para ambas as provas, indicando explicitamente quais seções são cobradas somente na prova Java SE 8 Programmer I.

Como estudar

Lembre-se de usar a linha de comando do Java. Não use o Eclipse ou qualquer outra IDE: os erros que o compilador da linha de comando mostra podem ser diferentes dos da IDE, e você não quer que isso atrapalhe seu desempenho.

Lembre-se de ficar atento. Na prova não ficará claro qual o assunto que está sendo testado e você deve se concentrar em todo o código, não só em um assunto ou outro.

Esse processo é longo e a recomendação é que agende a prova agora mesmo no site da Oracle, para que não haja pausa desde o primeiro dia de leitura, até o último dia de leitura, a execução de diversos simulados e a prova em si.

Não deixe de testar todo o código em que não sentir confiança. Os exercícios são gerados de propósito para causar insegurança no candidato, para levá-lo para um lado, sendo que o problema pode estar em outro. E faça muitos exercícios e simulados.

Os exercícios são dados em inglês e com nomes de classes e variáveis escolhidos propositalmente. Os nomes não dizem muito o que elas fazem e, por vezes, são enganosos. Fique atento a esses detalhes durante sua prova, e acostume-se a eles durante os exercícios que fará por aqui.

Não hesite, tire suas dúvidas no site do GUJ e nos avise de sua certificação via Twitter ou Facebook:

- http://www.guj.com.br
- http://www.twitter.com/casadocodigo
- http://www.facebook.com/casadocodigo

Você pode também encontrar alguns simulados e indicações neste site: http://www.coderanch.com/how-to/java/OcajpFaq.

Bom estudo, boa prova, boa sorte e, acima de tudo, bem-vindo ao grupo daqueles que não só usam uma linguagem, mas a dominam.

Seções da prova

Os assuntos cobrados e abordados aqui são:

1. Java Basics
    - Define the scope of variables
    - Define the structure of a Java class
    - Create executable Java applications with a main method
    - Import other Java packages to make them accessible in your code
2. Working With Java Data Types
    - Declare and initialize variables
    - Differentiate between object reference variables and primitive variables
    - Read or write to object fields
    - Explain an Object's Lifecycle (creation, "dereference" and garbage collection)
    - Call methods on objects
    - Manipulate data using the StringBuilder class and its methods
    - Creating and manipulating Strings
3. Using Operators and Decision Constructs

- Use Java operators
- Use parenthesis to override operator precedence
- Test equality between Strings and other objects using == and equals ()
- Create if and if/else constructs
- Use a switch statement

4. Creating and Using Arrays

    - Declare, instantiate, initialize and use a one-dimensional array
    - Declare, instantiate, initialize and use multi-dimensional array
    - Declare and use an ArrayList

5. Using Loop Constructs

    - Create and use while loops
    - Create and use for loops including the enhanced for loop
    - Create and use do/while loops
    - Compare loop constructs
    - Use break and continue

6. Working with Methods and Encapsulation

    - Create methods with arguments and return values
    - Apply the static keyword to methods and fields
    - Create an overloaded method
    - Differentiate between default and user defined constructors
    - Create and overload constructors
    - Apply access modifiers

- Apply encapsulation principles to a class
- Determine the effect upon object references and primitive values when they are passed into methods that change the values

7. Working with Inheritance

    - Implement inheritance
    - Develop code that demonstrates the use of polymorphism
    - Differentiate between the type of a reference and the type of an object
    - Determine when casting is necessary
    - Use super and this to access objects and constructors
    - Use abstract classes and interfaces

8. Handling Exceptions

    - Differentiate among checked exceptions, RuntimeExceptions and Errors
    - Create a try-catch block and determine how exceptions alter normal program flow
    - Describe what Exceptions are used for in Java
    - Invoke a method that throws an exception
    - Recognize common exception classes and categories

A prova do Java 8 possui algumas seções que mudaram de posições, mas continuam na prova. O conteúdo a seguir são as seções novas, que também são abordadas no livro:

- Java Basics

    - Run a Java program from the command line;

including console output
    - Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc

- Working with Java Data Types
    - Develop code that uses wrapper classes such as Boolean, Double, and Integer.

- Working with Selected Classes from the Java API
    - Create and manipulate calendar data using classes from java.time.LocalDateTime, java.time.LocalDate, java.time.LocalTime, java.time.format.DateTimeFormatter, java.time.Period
    - Write a simple Lambda expression that consumes a Lambda Predicate expression

Sumário

Versão: 22.0.21

CAPÍTULO 1

O BÁSICO DE JAVA

1.1 DEFINA O ESCOPO DE VARIÁVEIS

O escopo é o que determina em que pontos do código uma variável pode ser usada.

Variáveis locais

Chamamos de locais as variáveis declaradas dentro de blocos, como dentro de métodos ou construtores. Antes de continuar, vamos estabelecer uma regra básica: o ciclo de vida de uma variável local vai do ponto onde ela foi declarada, até o fim do **bloco** onde ela foi declarada.

Mas o que é um bloco? Podemos entender como bloco um trecho de código entre chaves. Pode ser um método ou um construtor:

```
public void m1() { // method - opening
    int x = 10; // method local variable

} // method - closing
```

Ou ainda o corpo de um `if` , de um `for` etc.:

```
public void m1() { // method - opening
    int x = 10; // method local variable
```

```
    if (x >= 10) { // if - opening
        int y = 50; // if local variable
        System.out.print(y);

    } // if - closing

} // method - closing
```

Analisando esse código, temos uma variável `x`, que é declarada no começo do método. Ela pode ser utilizada durante todo o corpo do método.

Dentro do `if`, declaramos a variável `y`, e `y` só pode ser usada dentro do corpo do `if`, delimitado pelas chaves. Se tentarmos usar `y` fora do corpo do `if`, teremos um erro de compilação, pois a variável saiu do seu escopo.

Tome cuidado especial com loops `for`. As variáveis declaradas na área de inicialização do loop só podem ser usadas no corpo do loop. O exemplo a seguir mostra a tentativa de usar uma variável cujo escopo não pode ser acessado:

```
for (int i = 0, j = 0; i < 10; i++)
    j++;

System.out.println(j); // compilation error
```

Parâmetros de métodos também podem ser considerados variáveis locais ao método, ou seja, só podem ser usados dentro do método onde foram declarados:

```
    class Test {

        public void m1(String s) {
            System.out.print(s);
        }

        public void m2() {
            System.out.println(s); // compilation error
```

```
        }
    }
```

Variáveis de instância

Variáveis de instância ou variáveis de objeto são os atributos dos objetos. São declaradas dentro da classe, mas fora de qualquer método ou construtor. Podem ser acessadas por qualquer membro da classe e ficam em escopo enquanto o objeto existir:

```
class Person {
    // instance or object variable
    String name;

    public void setName(String n) {
        // explicit (this) instance variable access
        this.name = n;
    }
}
```

Variáveis estáticas (class variables)

Podemos declarar variáveis que são compartilhadas por todas as instâncias de uma classe usando a palavra chave `static`. Essas variáveis estão no escopo da classe, e lá ficarão enquanto a classe estiver carregada na memória (enquanto o programa estiver rodando, na grande maioria dos casos). O código a seguir define uma variável estática:

```
class Person {
    static int count = 15;
}
```

No caso de variáveis `static`, não precisamos ter uma referência para usá-las e podemos acessá-las diretamente a partir da classe, desde que respeitando as regras de visibilidade da variável. O código a seguir mostra o acesso a mesma variável

através da referência de uma instância e a referência da classe.

```
class Person {
    static int id = 15;
}

class Test {
    public static void main(String[] args) {
        Person p = new Person();
        // instance reference access: 15
        System.out.println(p.id);
        // class reference access: 15
        System.out.println(Person.id);
    }
}
```

Uma vez que variáveis estáticas podem ser acessadas dessas duas maneiras, tome cuidado com nomes enganosos de suas variáveis, como o caso do `id` anterior.

Variáveis com o mesmo nome

Logicamente, não é possível declarar duas variáveis no mesmo escopo com o mesmo nome:

```
public void method() {
    int a = 0;
    int a = 10;// compile error
}
```

Mas, eventualmente, podemos ter variáveis em escopos diferentes que podem ser declaradas com o mesmo nome. Em casos em que possam haver ambiguidades na hora de declará-las, o próprio compilador vai emitir um erro, evitando a confusão. Por exemplo, não podemos declarar variáveis de classe e de instância com o mesmo nome:

```
class Bla {
    static int a;
```

```
    int a; // compile error
}
...

System.out.println(new Bla().a); // which one?
```

Se a definição da classe `Bla` compilasse, o JVM ficaria perdido em qual das duas referências estamos tentando usar. A decisão dos criadores da linguagem foi que a variável estática e de instância não podem ter o mesmo nome, portanto, erro de compilação.

Também não podemos declarar variáveis locais com o mesmo nome de parâmetros:

```
public void method(String par) {
    int par = 0; // compilation error

    System.out.println(par); // which one?
}
```

Apesar de parecer estranho, é permitido declarar variáveis locais ou parâmetros com o mesmo nome de variáveis de instância ou de classe. Essa técnica é chamada de *shadowing*. Nesses casos, é possível resolver a ambiguidade: para variáveis de classe, podemos referenciar pela própria classe; para variáveis de instância, usamos a palavra chave `this` :

```
class Person {

    static int x = 0;
    int y = 0;

    public static void setX(int x) {
        Person.x = x; // type (class) explicit access
    }

    public void setY(int y) {
        this.y = y; // instance (this) explicit access
    }
}
```

Quando não usamos o `this` ou o nome da classe para usar a variável, o compilador sempre utilizará a variável de "menor" escopo:

```
class X {
    int a = 100;

    public void method() {
        int a = 200; // shadowing
        System.out.println(a); // 200
    }
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o código a seguir:

```
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            System.out.println(i);
        }
        System.out.println(i);
    }
}
```

a) Erro de compilação.

b) Compila e roda, imprimindo de 0 até 19 e depois 19.

c) Compila e roda, imprimindo de 0 até 19, depois ocorre um erro de execução.

2) Escolha a opção adequada ao tentar compilar e rodar o código a seguir:

```
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 20; i++)
```

```
            System.out.println(i);
            System.out.println(i);
        System.out.println("finished");
    }
}
```

a) Erro de compilação.

b) Compila e roda, imprimindo de 0 até 19 e depois 19.

c) Compila e roda, imprimindo de 0 até 19, depois 19, depois `finished`.

d) Compila e roda, imprimindo de 0 até 19, depois ocorre um erro de execução.

3) Escolha a opção adequada ao tentar compilar e rodar o código a seguir:

```
class Test {
    public static void main(String[] args) {
        for (int i = 0; i < 20; i++) {
            System.out.println(i);
        }
        int i = 15;
        System.out.println(i);
    }
}
```

a) Erro de compilação na linha 6. A variável `i` não pode ser redeclarada.

b) Erro de compilação na linha 7. A variável `i` é ambígua.

c) Compila e roda, imprimindo de 0 até 19 e depois 15.

d) Compila e roda, imprimindo de 0 até 19, depois ocorre um erro de execução na linha 6.

e) Compila e roda, imprimindo de 0 até 19 e depois 19 novamente.

4) Escolha a opção adequada ao tentar compilar e rodar o código a seguir:

```
class Test {
    static int x = 15;

    public static void main(String[] x) {
        x = 200;
        System.out.println(x);
    }
}
```

a) O código compila e roda, imprimindo 200.

b) O código compila e roda, imprimindo 15.

c) O código não compila.

d) O código compila, mas dá erro em execução.

5) Escolha a opção adequada ao tentar compilar e rodar o código a seguir:

```
class Test {
    static int i = 3;

    public static void main(String[] a) {
        for (new Test().i = 10; new Test().i < 100;
             new Test().i++) {
            System.out.println(i);
        }
    }
}
```

a) Não compila a linha 4.

b) Não compila a linha 5.

c) Compila e imprime 100 vezes o número 3.

d) Compila e imprime os números de 10 até 99.

1.2 DEFINA A ESTRUTURA DE UMA CLASSE JAVA

Nesta seção, vamos entender a estrutura de um arquivo Java, no qual inserir as declarações de pacotes e imports. Veremos também como declarar classes e interfaces.

Para entender a estrutura de uma classe, vamos ver o arquivo `Person.java`:

```
// 0 or 1 package
package br.com.caelum.certification;

// 0 or more imports
import java.util.Date;

// 0 or more types
class Person {
    // class content
}
```

Pacotes

Pacotes servem para separar e organizar as diversas classes que temos em nossos sistemas. Todas as classes pertencem a um pacote, sendo que, caso o pacote não seja explicitamente declarado, a classe fará parte do que chamamos de **pacote padrão**, ou **default package**. Todas as classes no *default package* se enxergam e podem ser utilizadas entre si. Classes no pacote default não podem ser importadas para uso em outros pacotes:

```
// no package => package "default"
```

```
class Person {
    //...
}
```

Para definir qual o pacote que a classe pertence, usamos a palavra-chave `package`, seguida do nome do pacote. Só pode existir um único `package` definido por arquivo, e ele deve ser a primeira instrução do arquivo. Após a definição do `package`, devemos finalizar a instrução com um `;` (ponto e vírgula). Podem existir comentários antes da definição de um pacote:

```
// package declaration
package br.com.caelum.certification;

class Person {
    //...
}
```

Comentários não são considerados parte do código, portanto, podem existir em qualquer lugar do arquivo Java, sem restrições. Para inserir comentário em nosso código, temos as seguintes formas:

```
// single line comment

/*
   multiple line
   comment
 */
class /* middle of line comment */ Person {

    /**
     *  JavaDoc, starts with slash, then two *
     *  and it is multiple line
     */
    public void method() { // single line comment again
    }
}
```

PARA SABER MAIS: JAVADOC

Javadoc é um tipo especial de comentário que pode ser usado para gerar uma documentação HTML a partir de nosso código. Para saber mais, acesse http://bit.ly/oracle-javadoc.

Classe

Uma classe é a forma no Java onde definimos os atributos e comportamentos de um objeto. A declaração de uma classe pode ser bem simples, apenas a palavra `class` seguida do nome e de `{}` (chaves):

```
class Person {}
```

Existem outros modificadores que podem ser usados na definição de uma classe, mas veremos essas outras opções mais à frente, onde discutiremos esses modificadores com mais detalhes.

Vale lembrar de que Java é *case sensitive*, e `Class` é o nome de uma classe e não podemos usá-lo para definir uma nova classe.

Dentro de uma classe, podemos ter variáveis, métodos e construtores. Essas estruturas são chamadas de **membros** da classe:

```
class Person {

    String firstname;
    String lastname;

    Person(String firstname, String lastname) {
        this.firstname = firstname;
        this.lastname = lastname;
```

```
    }

    public String getFullName() {
        return this.firstname + this.lastname;
    }
}
```

NOMES DOS MEMBROS

Podemos ter membros de tipos diferentes com o mesmo nome. Fique atento, o código a seguir compila normalmente:

```
class B {
    String b;

    B() {
    }

    String b() {
        return null;
    }
}
```

Variaveis

Usando como exemplo a classe `Person` definida anteriormente, `firstname` e `lastname` são variáveis. A declaração de variáveis é bem simples, sempre o **tipo** seguido do **nome** da variável.

Dizemos que essas são variáveis de instância, pois existe uma cópia delas para cada objeto `Person` criado em nosso programa. Cada cópia guarda o estado de uma certa instância desses objetos.

Existem ainda variáveis que não guardam valores ou

referências para uma determinada instância, mas sim um valor compartilhado por todas as instâncias de objetos. Essas são variáveis **estáticas**, definidas com a palavra-chave `static`. Veremos mais à frente sobre esse tipo de membro.

Métodos

A declaração de métodos é um pouquinho diferente, pois precisamos do **tipo do retorno**, seguido do **nome do método** e de parênteses, sendo que pode ou não haver parâmetros de entrada desse método. Cada parâmetro é uma declaração de variável em si. Essa linha do método, na qual está definido o retorno, teremos a **assinatura do método** no nome e nos parâmetros. Cuidado, pois a **assinatura de um método** inclui somente o nome do método e os tipos dos parâmetros.

Assim como variáveis, métodos também podem ser `static`, como veremos mais adiante.

Construtores

Uma classe pode possuir zero ou vários construtores. Nossa classe `Person` possuía um construtor que recebe como parâmetros o nome e o sobrenome da pessoa. A principal diferença entre a declaração de um método e um construtor é que um **construtor não tem retorno e possui o mesmo nome da classe**.

Métodos com o mesmo nome da classe

Cuidados com métodos que parecem construtores:

```
class Executor {

    Executor() { // constructor
    }

    void Executor() { // method
    }

}
```

Note que um construtor pode ter um `return` vazio:

```
class X {
    int j = -100;

    X(int i) {
        if (i > 1)
            return;
        j = i;
    }
}
```

Caso o valor recebido no construtor (`i`) seja maior do que 1, o valor de `j` será -100; caso contrário, `j` passa a ter o mesmo valor de `i` .

Interfaces

Além de classes, também podemos declarar *interfaces* em nossos arquivos Java. Para definir uma interface, usamos a palavra reservada `interface` :

```
interface Authenticable {

    final int PASSWORD_LENGTH = 8;

    void authenticate(String login, String password);
}
```

Em uma interface, devemos apenas definir a assinatura do método, sem a sua implementação. Além da assinatura de métodos, também é possível declarar constantes em interfaces.

Multíplas estruturas em um arquivo

Em Java, é possível definir mais de uma classe/interface em um mesmo arquivo, embora devamos seguir algumas regras:

- Podem ser definidos em qualquer ordem;
- Se existir alguma classe/interface pública, o nome do arquivo deve ser o mesmo dessa classe/interface;
- Só pode existir uma classe/interface pública por arquivo;
- Se não houver nenhuma classe/interface pública, o arquivo pode ter qualquer nome.

Logo, são válidos:

```
// file1.java
interface First {}

class Second {}

// Third.java
public class Third {}

interface Fourth {}
```

Pacotes e imports em arquivos com múltiplas estruturas

As regras de pacotes e imports valem também para arquivos com múltiplas estruturas definidas. Caso exista a definição de um pacote, ela vale para todas as classes/interfaces definidas nesse arquivo, e o mesmo vale para *imports*.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir sem nenhum parâmetro na linha de comando, como `java D`:

```
package a.b.c;

import java.util.*;

class D {
    public static void main(String[] args) {
        ArrayList<String> existing = new ArrayList<String>();

        for (String arg : args) {
            if (new E().exists(arg))
                existing.add(arg);
        }
    }
}

import java.io.*;

class E {
    public boolean exists(String name) {
        File f = new File(name);
        return f.exists();
    }
}
```

a) O arquivo não compila.

b) O arquivo compila, mas dá erro de execução, já que o array é nulo.

c) O arquivo compila, mas dá erro de execução, já que o array tem tamanho zero.

d) Roda e imprime `false`.

e) Roda e imprime `true`.

f) Roda e não imprime nada.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class Test {
    int Test = 305;

    void Test() {
        System.out.println(Test);
    }

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

a) O código não compila: erros nas linhas 2, 4, 5 e 6.

b) O código não compila: erro na linha 5.

c) O código não compila: erros nas linhas 2, 4 e 6.

d) O código compila e, ao rodar, imprime `305`.

e) O código compila e não imprime nada.

f) O código compila e, ao rodar, imprime uma linha em branco.

3) Escolha a opção adequada ao tentar compilar o arquivo a seguir:

```
package br.com.teste;

import java.util.ArrayList;
```

a) Erro na linha 1: definimos o pacote, mas nenhum tipo.

b) Erro na linha 3: importamos algo desnecessário ao arquivo.

c) Compila sem erros.

1.3 CRIE APLICAÇÕES JAVA EXECUTÁVEIS E RODE NA LINHA DE COMANDO

Nesta seção, entenderemos as diferenças entre classes normais e classes que podem ser executadas pela linha de comando.

Uma classe executável é uma classe que possui um método inicial para a execução do programa — o método `main`, que será chamado pela JVM. Classes sem o método `main` não são classes executáveis e não podem ser usadas como ponto inicial da aplicação.

Método main

O tal método de entrada deve seguir algumas regras para ser executado pela JVM:

- Ser público (`public`);

- Ser estático (`static`);
- Não ter retorno (`void`);
- Ter o nome *main*;
- Receber como parâmetro um `array` ou `varargs` de String (`String[]` ou `String...`).

Os seguintes exemplos são métodos `main` válidos:

```
// parameter: array
public static void main (String[] args) {}

// parameter: varargs
public static void main (String... args) {}

// static public/public static are ok
static public void main(String[] args) {}

// parameter name does not matter
public static void main (String...
listOfArgumentsOurUserSentUs){}

// parameter: array variation
public static void main (String args[]) {}
```

Executando uma classe pela linha de comando

Para executar uma classe com `main` pela linha de comando, devemos compilar o arquivo com o comando `javac` e executar a classe com o comando `java` . O exemplo a seguir usa o arquivo `HelloWorld.java` :

```
public class HelloWorld {

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

Compilamos e executamos no console com os seguintes

comandos:

```
$ javac HelloWorld.java
$
$ java HelloWorld
Hello World!
```

Repare que, para compilar a classe, passamos como parâmetro para o comando `javac` o nome do arquivo, enquanto para executar, passamos apenas o nome da classe (`HelloWorld`) para o comando `java` .

Passando parâmetros pelo linha de comando

Ao executarmos uma classe pela linha de comando, podemos passar parâmetros para o método `main` . Esses valores serão recebidos no array do método `main` . Por exemplo, vamos passar um nome para a classe `HelloWorld` :

```
public class HelloWorld{

    public static void main(String[] args) {
        // reading the first (0) position
        String arg = args[0];
        System.out.println("Hello " + arg + "!");
    }
}
```

Para informar o valor do parâmetro, é só informá-lo **após** o nome da classe que está sendo executada:

```
java HelloWorld Mario
Hello Mario!
```

Você pode passar quantos parâmetros quiser, basta separá-los por espaço. Cada parâmetro informado será armazenado em uma posição do array, na mesma ordem em que foi informado.

Compilação e execução

Para criar um programa Java, é preciso escrever um código-fonte e, através de um compilador, gerar o executável (bytecode). O compilador do JDK (*Java Development Kit*) é o *javac*. Para a prova de certificação, devemos conhecer o comportamento desse compilador.

A execução do bytecode é feita pela **JVM** (*Java Virtual Machine*). O comando `java` invoca a máquina virtual para executar um programa java. Ao baixarmos o Java, podemos escolher baixar o JDK, que já vem com o JRE, ou somente o JRE (*Java Runtime Environment*), que inclui a *Virtual Machine*.

Algumas questões da prova abordam aspectos fundamentais do processo de compilação e de execução. É necessário saber como os comandos `javac` e o `java` procuram os arquivos.

javac

Imagine o arquivo `Exam.java` dentro do diretório `projeto`:

```
class Exam {
    double timeLimit;
}
```

```
$ javac Exam.java
```

O bytecode da classe `Exam` gerado na compilação é colocado no arquivo `Exam.class`, dentro do nosso diretório de trabalho (no meu caso, `projeto`). O resultado é um diretório chamado `projeto`, com dois arquivos dentro dele: `Exam.java` e `Exam.class`.

Os projetos profissionais usam o recurso de pacotes para

melhor organizar os fontes e os bytecodes. Vejamos qual é o comportamento do `javac` com a utilização de pacotes. Colocamos o arquivo `Exam.java` no diretório `certification`:

```
package certification;
class Exam {
    double timeLimit;
}
```

```
[certification]$ javac certification/Exam.java
```

Nesse exemplo, o arquivo `Exam.class` é colocado no diretório `certification`, junto com seu fonte, o `Exam.java`.

> **Escolhendo a versão do Java na hora de compilar**
>
> Na hora da compilação, é possível definir em que versão do Java o código-fonte foi escrito. Isso é feito com a opção `-source` do comando `javac` (`javac MyClass.java -source 1.3`).

java

Vamos utilizar um exemplo para mostrar o funcionamento do comando `java`, criando o arquivo `Test.java` no mesmo diretório, no mesmo pacote:

```
package certification;
class Test {
    public static void main(String[] args) {
        Exam p = new Exam();
        p.timeLimit = 210;
        System.out.println(p.timeLimit);
    }
}
```

Compilamos e rodamos:

```
$ javac certification/Test.java
$ java certification.Test
```

E a saída é:

```
210.0
```

E o resultado é a existência do arquivo `Test.class`, no diretório `certification`.

Somente o arquivo `Test.java` foi passado para o compilador. Nesse arquivo, a classe `Test` usa a classe `Exam` que se encontra em outro arquivo, `Exam.java`. Dessa forma, o compilador vai compilar automaticamente o arquivo `Exam.java` se necessário.

Para executar, é preciso passar o nome completo da classe desejada para a máquina virtual. O sufixo `.class` não faz parte do nome da classe, então ele não aparece na invocação da máquina virtual pelo comando `java`.

Propriedades na linha de comando

A prova ainda cobra conhecimentos sobre como executar um programa Java passando **parâmetros** ou **propriedades** para a JVM. Essas propriedades são identificadas pelo `-D` antes delas. **Este *-D* não faz parte da chave**.

```
java -Dkey1=abc -Dkey2=def Foo xpto bar
```

`key1=abc` e `key2=def` são parâmetros/propriedades, e `xpto` e `bar` são argumentos recebidos pelo método `main`.

Classpath

Para compilar ou executar, é necessário que os comandos `javac` e `java` possam encontrar as classes referenciadas pela aplicação Java.

A prova de certificação exige o conhecimento do algoritmo de busca das classes. As classes feitas pelo programador são encontradas através do **classpath** (caminho das classes).

O classpath é formado por **diretórios**, `jars` e `zips` que contenham as classes e pacotes da nossa aplicação. Por padrão, o classpath está configurado para o diretório corrente (`.`).

Configurando o classpath

Há duas maneiras de configurar o classpath:

1. **Configurando a variável de ambiente `CLASSPATH` no sistema operacional.**

   Basta seguir as opções do SO em questão e definir a variável. Isso é considerado uma má prática no dia a dia, porque é um classpath global, que vai valer para qualquer programa Java executado na máquina.

2. **Com as opções `-cp` ou `-classpath` dos comandos `javac` ou `java`.**

   É a forma mais usada. Imagine que queremos usar alguma biblioteca junto com nosso programa:

   ```
   $ javac -cp /path/to/library.jar Test.java
   $ java -cp /path/to/library.jar Test
   ```

   E podemos passar tanto caminhos de outras pastas como de `JARs` ou `zips`. Para passar mais de uma coisa no

classpath, usamos o separador de parâmetros no SO (no Windows é ponto e vírgula, no Linux/ Mac/ Solaris/ Unix são dois pontos):

```
$ javac -cp /path/to/library.jar:/another/path/
certification/Test.java
$ java -cp /path/to/library.jar:/another/path/
certification.Test
```

PARA SABER MAIS: ARQUIVOS JAR

Para facilitar a distribuição de bibliotecas de classes ou de aplicativos, o JDK disponibiliza uma ferramenta para a compactação das classes Java.

Um arquivo `JAR` nada mais é que a pasta de nossas classes no formato `ZIP`, mas com extensão `.jar`.

Para criar um `jar` incluindo a pasta `certification` que fizemos antes:

```
jar -cf library.jar certification
```

Agora podemos executar nossa classe usando esse `jar`:

```
java -cp library.jar certification.Test
```

PARA SABER MAIS: META-INF/MANIFEST.MF

Ao criar o JAR usando o comando `jar` do JDK, ele cria automaticamente a pasta `META-INF`, que é usada para configurações relativas a ele. E dentro dela, cria também o arquivo `Manifest.mf`.

Esse arquivo pode ser usado para algumas configurações. Por exemplo, é possível dizer qual classe do nosso `jar` é a principal (**Main-Class**) e que deve ser executada. Basta criar um arquivo chamado `Manifest.mf` com a seguinte instrução, indicando a classe com o método `main`:

```
Main-Class: certification.Test
```

Um ponto que exige atenção extra é que o manifest **deve** possuir uma **última linha vazia**. Sem esta linha, o manifest não é adicionado e não conseguimos executar o `jar`.

Depois gerá-lo passando esse arquivo:

```
jar -cfm bib.jar mymanifest certification
```

Na hora de rodar um `jar` com `Main-Class`, basta usar:

```
java -jar bib.jar
```

Exercícios

1) Qual é uma assinatura válida do método `main` para executar um programa Java?

a) `public static void main(String... args)`

b) `public static int main(String[] args)`

c) `public static Void main(String []args)`

d) `protected static void main(String[] args)`

e) `public static void main(int argc, String[] args)`

2) Escolha a opção adequada para compilar e rodar o arquivo `A.java` , existente no diretório `b` :

```
package b;
class A {
    public static void main(String[] args) {
        System.out.println("running");
    }
}
```

a) `javac A e java A`

b) `javac A.java e java A`

c) `javac b/A.java e java A`

d) `javac b/A.java e java b.A`

e) `javac b.A.java e java b.A`

f) `javac b.A e java b.A`

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        System.out.println(args); // A
        System.out.println(args.length); // B
        System.out.println(args[0]); // C
    }
}
```

a) Não compila: array não possui membro `length`.

b) Não compila: o método `println` não consegue imprimir um array.

c) Ao rodar sem argumentos, ocorre uma `ArrayIndexOutOfBoundsException` na linha C.

d) Ao rodar sem argumentos, ocorre uma `NullPointerException` na linha B.

e) Ao rodar sem argumentos, são impressos os valores "1" e "A".

f) Ao rodar com o argumento "certification", são impressos os valores "2" e "A".

4) Escolha a opção adequada para rodar a classe `A.java` presente no diretório `b`, que foi compactado em um arquivo chamado `program.jar`, sendo que não existe nenhum arquivo de manifesto:

```
package b;
class A {
    public static void main(String[] args) {
        System.out.println(args[0]);
    }
}
```

a) `java jar program.jar`

b) `java jar program.jar b.A`

c) `java -jar program.jar`

d) `java -jar program.jar b.A`

e) `java -cp program.jar`

f) `java -cp program.jar b.A`

5) Escolha a opção adequada para compilar a classe `A.java`, definida como no pacote `b` presente no diretório `b`, e adicionar também o arquivo `program.jar` na busca de classes durante a compilação. Lembre-se de que `.` (ponto) significa o diretório atual.

a) `javac -cp b.A.java -cp program.jar`

b) `javac -jar program.jar b.A.java`

c) `javac -cp program.jar:b A.java`

d) `javac -cp program.jar:. b.A.java`

e) `javac -cp . -cp program.jar`

f) `javac -jar program.jar:. b/A.java`

g) `javac -cp program.jar:b b/A.java`

h) `javac -cp program.jar:. b/A.java`

1.4 IMPORTE PACOTES JAVA E DEIXE-OS ACESSÍVEIS AO SEU CÓDIGO

Se duas classes estão no mesmo pacote, elas se "enxergam" entre si, sem a necessidade de colocar o nome do pacote. Por exemplo, imagine que as classes `Person` e `Address` estejam no mesmo pacote:

```
package model;
```

```
class Address {
    String address;
    String number;
    String city;
    //...
}
```

E o outro arquivo:

```
package model;

class Person {
    Address address; // Person using an address
}
```

Para usar uma classe que está em outro pacote, temos duas opções: podemos referenciá-la usando o que chamamos de *Full Qualified Name*, ou seja, o nome do pacote seguido do nome da classe. O código ficaria assim:

```
package finance;

class Order {
    model.Person client; // Referencing the type Person
                         // from another package
}
```

Tentamos compilar, mas ele não deixa, porque uma classe, por padrão, só pode ser acessada dentro do próprio pacote, e a nossa classe `Person` está no pacote `model`. Portanto, definiremos nossa classe `Person` como pública.

Veremos com mais calma os modificadores de acesso na seção que cobra isso na prova. Por enquanto, basta lembrar de que classes públicas podem ser acessadas por outros pacotes, já classes padrão não podem.

```
package model;
```

```
public class Person { // anyone can reference me!
    Address address; // Person referencing the type
                     //in the same package
}
```

Outra opção é importar a classe `Product` e referenciá-la apenas pelo nome simples dentro de nosso código. Para fazer o *import*, usamos a palavra `import`, seguida do `Full Qualified Name` da classe. A instrução de `import` deve aparecer na classe logo após o `package` (se este existir), e antes da definição da classe. É possível importar mais de uma classe por vez:

```
package model;

// import type Product from package order
import order.Product;
// another import
import java.util.Date;

class Order {
    Person client; // same package
    Product item; // imported reference
    Date creationDate; // imported reference
}
```

Também é possível importar todas as classes de um determinado pacote, basta usar um `*` (asterisco) após o nome do pacote. No exemplo a seguir, importamos todos os tipos do pacote `order`:

```
import order.*;
```

Importando classes com mesmo nome

Quando precisamos usar duas classes com o mesmo nome, mas de pacotes diferentes, só podemos importar uma delas; a outra deve ser referenciada pelo `Full Qualified Name`. Tentativas de

importar as duas classes resultarão em erros de compilação:

```
import java.util.Date;
import java.sql.Date; // Compilation error: Date/Date?

class Test {
    Date d1;
    Date d2;
}
```

O correto seria:

```
import java.util.Date;

class Test {
    Date d1;            // java.util
    java.sql.Date d2; // java.sql
}
```

Caso tenhamos um `import` específico e um genérico, o Java usa o específico:

```
import java.util.*;
import java.sql.Date;

class Test {
    Date d1; // java.sql
    Date d2; // java.sql
}
```

Por padrão, todas as classes do pacote `java.lang` são importadas. Justamente por esse motivo é opcional escrevermos `import java.lang.String` ou `java.lang.String` por extenso, como em:

```
import java.lang.String; // optional

class Test {
    String name;
}
```

E também em:

```
class Test {
    java.lang.String name;
}
```

Devemos tomar muito cuidado com o caso de uma outra classe com o mesmo nome, no mesmo pacote:

```
class String {
    int value;
}
class Test {
    java.lang.String s1; // java.lang.String
    String s2; // String with value
}
```

Um ponto importante é que nenhuma classe de pacote que não seja o padrão pode importar uma classe do pacote padrão. Considere o arquivo `Manager.java`, compilável, a seguir:

```
class Manager {
}
```

O arquivo `model/Bank.java` jamais compilará:

```
package model;
class Bank {
    Manager manager; // compilation error
}
```

Pacotes

Nesta seção, entenderemos mais a fundo como funciona a declaração de pacotes, e como isso influencia nos `imports` das classes.

Como já discutimos anteriormente, pacotes servem para organizar suas classes e interfaces. Eles permitem agrupar

componentes que tenham alguma relação entre si, além de garantir algum nível de controle de acesso a membros. Além de serem uma divisão lógica para as suas classes, os pacotes também definem uma separação física entre os arquivos de seu projeto, já que espelham a estrutura de diretórios dos arquivos do projeto.

Subpacotes e estrutura de diretórios

Pacotes são usados pela JVM como uma maneira de encontrar as classes no sistema de arquivos, logo, a estrutura de diretórios do projeto deve ser a mesma da estrutura de pacotes. Vamos usar como exemplo a classe `Person`:

```
package project.model;

public class Person {}
```

O arquivo `Person.java` deve estar localizado dentro do diretório `model`, que deve estar dentro do diretório `project`, isto é, em:

```
project/model/Person.java
```

Dizemos que `model` é um subpacote de `project`, já que está dentro dele. Usamos o caractere `.` (ponto) como separador de pacotes e subpacotes. Podemos ter vários subpacotes, como `project.utils` e `project.converters`, por exemplo:

```
project/model/Person.java
project/utils/DateUtils.java
project/converters/IntConverter.java
project/converters/StringConverter.java
```

Convenções de nomes para pacotes

Existem algumas convenções para nomes de pacotes. Elas não

são obrigatórias, mas geralmente são seguidas para facilitar o entendimento e organização do código:

- O nome do pacote deve ser todo em letras minúsculas;
- Um pacote deve começar com o site da empresa, ao contrário;
- Após o site, deve vir o projeto;
- Após o projeto, a estrutura é livre.

Import usando classes de outros pacotes

Existem diversas maneiras de referenciar uma classe de pacote diferente em nosso código. Vamos analisar essas opções:

Full Qualified Name

Podemos referenciar uma classe em nosso código usando o que chamamos de **Full Qualified Name** (ou **FQN**). Ele é composto pelo **pacote completo** mais o **nome da classe**, por exemplo:

```
class Person {
    // Full Qualified Name
    java.util.Calendar birthday;
}
```

import

Usar o **FQN** nem sempre deixa o código legível, portanto, em vez de usar o nome completo da classe, podemos importá-la e usar apenas o nome simples da classe:

```
import java.util.Calendar;

class Person {
    Calendar birthday;
}
```

É permitido também importar todas as classes de um pacote de uma vez, usando o `*` (asterisco) no lugar do nome da classe:

```
import java.util.*;

class Person {
    Calendar birthday; // java.util.Calendar
    List<String> nicknames; // java.util.List
}
```

Caso existam duas classes com o mesmo nome, mas de pacotes diferentes, só podemos importar uma delas. A outra deve ser referenciada pelo *FQN*:

```
import java.util.Date;

class Foo {
    Date some; // java.util.Date
    java.sql.Date other; // java.sql.Date
}
```

Múltiplos imports com *

Caso importemos dois ou mais pacotes que contenham classes com o mesmo nome, será obrigatório especificar qual das classes queremos utilizar, usando o *FQN*. Ao tentar usar apenas o nome simples da classe, teremos um erro de compilação:

```
import java.util.*;
import java.sql.*;

public class Test {
    private Date d; // compilation error on this line
}
```

Enquanto isso, caso você tente importar duas classes com o mesmo nome, o erro é na linha do import:

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

```
import java.sql.Date; // compilation error

public class Test {
    private Date d; // no error
}
```

Import de subpacotes

Em Java, não podemos importar todas as classes de subpacotes usando `*`. Veja a seguinte situação, considerando que cada classe foi definida em seu próprio arquivo:

```
package myproject.certification;

public class Question {}

package myproject.bank;

public class QuestionDao {}

package myproject;

public class Exam {}
```

Agora o exemplo a seguir importa `myproject.*`, como o `*` não importa subpacotes, somente a classe `Exam` é importada.

```
package myproject.test;

import myproject.*; // imports Exam only

public class Test {}
```

O único modo de importar todas as classes é explicitamente importando cada subpacote:

```
package myproject.test;

import myproject.*;
import myproject.certification.*;
import myproject.bank.*;
public class Test {}
```

import static

Desde o Java 5, é possível importar apenas métodos e atributos estáticos de uma classe, usando a palavra-chave `static` juntamente com o `import`. Podemos importar um a um, ou simplesmente importar todos usando `*`. Considere o exemplo a seguir com diversos membros estáticos:

```
package model;

public class Utils {

    // static attributes
    public static int AGE = 33;

    // static methods
    public static void method1() {}
    public static void method1(int a) {}

}
```

O código a seguir importa referências para todos os membros estáticos visíveis na classe `Utils`:

```
import static model.Utils.*; // import static

public class Tests {

    public static void main(String[] args) {
        int x = AGE;
        method1();
        method1(x);
    }
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o `Test`. Arquivo no diretório atual:

```
import model.A;
class Test {
    public static void main(String[] args) {
        new A("guilherme").print();
    }
}
```

Arquivo no diretório `model`:

```
package model;

class A {
    private String name;
    A(String name) {
        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

a) Não compila: erro na classe `Test`.

b) Não compila: erro na classe `A`.

c) Erro de execução: método `main`.

d) Roda e imprime "Guilherme".

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import model.basic.Client;
import model.advanced.Client;

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

No arquivo `model/basic/Client.java`:

```
public class Client{}
```

No arquivo `model/advanced/Client.java` :

```
public class Client{}
```

a) O código do primeiro arquivo não compila, erro ao tentar importar duas classes com o mesmo nome.

b) O código do terceiro arquivo não compila, erro ao tentar definir uma classe com o nome de uma classe que já existe.

c) O código todo compila, mas ao rodar dá erro por ter importado duas classes com o mesmo nome.

d) O código todo compila e roda imprimindo `Welcome!` . Uma vez que nenhuma das classes importadas é usada no código, não existe ambiguidade.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import model.basic.Client;
import model.advanced.*;

class Test {
    public static void main(String[] args) {
        System.out.println("Welcome " + new Client().name);
    }
}
```

No arquivo `model/basic/Client.java` :

```
public class Client{
    public String name="guilherme";
}
```

No arquivo `model/advanced/Client.java` :

```
public class Client{
```

```
    public String name = "mario";
}
```

a) O código do primeiro arquivo não compila, é dá erro ao tentar importar duas classes com o mesmo nome.

b) O código compila, mas, ao rodar, dá erro por ter importado duas classes com o mesmo nome.

c) O código compila e roda imprimindo `Welcome guilherme` .

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import model.basic.Client;
import model.basic.Client;

class Test {
    public static void main(String[] args) {
        System.out.println("Welcome " + new Client().name);
    }
}
```

No arquivo `model/basic/Client.java` :

```
public class Client{
    public String name="guilherme";
}
```

a) O código do primeiro arquivo não compila, erro ao tentar importar duas classes com o mesmo nome.

b) O código compila, mas, ao rodar, dá erro por ter importado duas classes com o mesmo nome.

c) O código compila e roda imprimindo `Welcome guilherme` , uma vez que não há ambiguidade.

5) Escolha a opção adequada ao tentar compilar os arquivos a seguir:

`a/A.java` :

```
package a;
class A {
    b.B variable;
}
```

`a/C.java` :

```
package a;
class C {
    b.B variable;
}
```

`a/b/B.java` :

```
package a.b;
class B {
}
```

a) Erro de compilação somente no arquivo A.

b) Erro de compilação somente no arquivo B.

c) Erro de compilação somente no arquivo C.

d) Erro de compilação nos arquivos A e B.

e) Erro de compilação nos arquivos A e C.

f) Erro de compilação nos arquivos B e C.

g) Compila com sucesso.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package A;
class B{
    public static void main(String[] a) {
        System.out.println("running");
    }
}
```

a) Não compila: a variável do método `main` deve se chamar `args`.

b) Não compila: pacote com letra maiúscula.

c) Compila mas não roda: a classe B não é pública.

d) Compila e roda.

7) Escolha a opção adequada ao tentar compilar os arquivos a seguir:

`a/A.java`:

```
package a;
public class A {
    public static final int VALUE = 15;
    public void run(int x) {
        System.out.println(x);
    }
}
```

`b/B.java`:

```
package b;
import static a.A.*;
class B{
    void m() {
        A a = new A();
        a.run(VALUE);
    }
}
```

a) B não compila: erro na linha 2.

b) B não compila: erro na linha 5.

c) B não compila: erro na linha 6.

d) Tudo compila.

8) Escolha a opção adequada ao tentar compilar os arquivos a seguir:

`a/A.java` :

```
package a;
public class A {
    public static final int VALUE = 15;
    public void run(int x) {
        System.out.println(x);
    }
}
```

`b/B.java` :

```
package b;
import a.A;
static import a.A.*;
class B{
    void m() {
        A a = new A();
        a.run(VALUE);
    }
}
```

a) B não compila: erro na linha 3.

b) B não compila: erro na linha 5.

d) B não compila: erro na linha 6.

d) Tudo compila.

9) Escolha a opção adequada ao tentar compilar os arquivos a

seguir:

`A.java` :

```
public class A {
    public static final int VALUE = 15;
    public void run(int x) {
        System.out.println(x);
    }
}
```

`b/B.java` :

```
package b;
import static A.*;
class B{
    void m() {
        A a = new A();
        a.run(VALUE);
    }
}
```

a) Não compila

b) Tudo compila.

CAPÍTULO 2

TRABALHANDO COM TIPOS DE DADOS EM JAVA

2.1 DECLARAR E INICIALIZAR VARIÁVEIS

Qualquer programa de computador precisa manter informações de alguma forma. As linguagens de programação permitem a criação de variáveis para que possamos armazenar informações. Por exemplo, se precisarmos guardar a idade de uma pessoa, podemos utilizar uma variável que seja capaz de manter números inteiros.

Quando precisamos de uma nova variável, devemos declarar que queremos criá-la. A declaração de variável no Java, obrigatoriamente, deve
informar o **tipo** e o **nome** que desejamos para ela. Por isso, essa linguagem é dita *explicitamente tipada* (todas as variáveis precisam ter o seu tipo definido).

O código a seguir mostra a declaração de uma variável chamada `age` do tipo primitivo `int`:

```
int age;
```

Nem toda linguagem exige que as variáveis sejam iniciadas

antes de serem utilizadas. Mas, no Java, a **inicialização é obrigatória** e pode ser implícita ou explícita. É de fundamental importância saber que, para usar uma variável, é necessário que ela tenha sido iniciada explicitamente ou implicitamente em algum momento antes da sua utilização.

Variáveis locais (declaradas dentro de métodos/construtores) devem ser explicitamente iniciadas antes de serem usadas, ou teremos um erro de compilação:

```
public void method() {
    int age;

    System.out.println(age); // compilation error
}
```

Já o código a seguir mostra a inicialização explícita de nossa variável:

```
public void method() {
    int age;

    age = 10; // explicit initialization

    System.out.println(age); // ok
}
```

Podemos declarar e iniciar a variável na mesma instrução:

```
double pi = 3.14;
```

Se eu tenho um `if` , a inicialização deve ser feita em todos os caminhos possíveis para que não haja um erro de compilação no momento da utilização da variável:

```
void method(int a) {
    double x;
    if(a > 1) {
        x = 6;
```

```
    }
    System.out.println(x); // compile error
}
```

O código a seguir mostra como corrigir a situação:

```
void method(int a) {
    double x;
    if(a > 1) {
        x = 6;
    } else {
        x = 0;
    }
    System.out.println(x); // ok
}
```

Ou ainda inicializando durante a declaração:

```
void method(int a) {
    double x = 0;
    if(a > 1) {
        x = 6;
    }
    System.out.println(x); // ok
}
```

Quando a variável é membro de uma classe, ela é iniciada implicitamente junto com o objeto com um valor *default*. Esse processo pode ser chamado de inicialização implícita (*implicit initialization*).

```
class Exam {
     double timeLimit; // implicit initialization: 0.0
}

Exam exam = new Exam();

System.out.println(exam.timeLimit);
```

Outro momento em que ocorre a inicialização implícita é na criação de arrays:

```
int[] numbers = new int[10];
System.out.println(numbers[0]); // 0
```

Tanto quando declaradas como variáveis membro, ou quando arrays são criadas, os valores default para as variáveis são:

- Primitivos numéricos inteiros — **0**
- Primitivos numéricos com ponto flutuante — **0.0**
- Boolean — **false**
- `Char` — **vazio**, equivalente a 0
- Referências — **null**

Os tipos das variáveis do Java podem ser classificados em duas categorias: primitivos e não primitivos (referências).

Tipos primitivos

Todos os tipos primitivos do Java já estão definidos e não é possível criar novos tipos primitivos. São oito os tipos primitivos do Java: `byte`, `short`, `char`, `int`, `long`, `float`, `double` e `boolean`.

O `boolean` é o único primitivo não numérico. Todos os demais armazenam números: `double` e `float` são ponto flutuante, e os demais, todos inteiros (incluindo `char`). Apesar de representar um caractere, o tipo `char` armazena seu valor como um número positivo. Em Java, não é possível declarar variáveis com ou sem sinal (*unsigned*), todos os números (exceto `char`) podem ser positivos e negativos.

Cada tipo primitivo abrange um conjunto de valores. Por exemplo, o tipo `byte` abrange os números inteiros de -128 até 127. Isso depende do tamanho em bytes do tipo sendo usado.

Os tipos inteiros têm os seguintes tamanhos:

- `byte` — 1 byte (8 bits, de -128 a 127);
- `short` — 2 bytes (16 bits, de –32.768 a 32.767);
- `char` — 2 bytes (só positivo), (16 bits, de 0 a 65.535);
- `int` — 4 bytes (32 bits, de –2.147.483.648 a 2.147.483.647);
- `long` — 8 bytes (64 bits, de –9.223.372.036.854.775.808 a 9.223.372.036.854.775.807).

Decorar o tamanho dos primitivos para prova

Não há a necessidade de decorar o intervalo e tamanho de todos os tipos de primitivos para a prova. O único intervalo cobrado é o do `byte`.

É importante também saber que o `char`, apesar de ter o mesmo tamanho de um `short`, não consegue armazenar todos os números que cabem em um `short`, já que o `char` só armazena números positivos.

PARA SABER MAIS: CALCULANDO O INTERVALO DE VALORES

Dado o número de bits **N** do tipo primitivo inteiro, para saber os valores que ele aceita, usamos a seguinte conta:

```
-2^(n-1) a 2^(n-1) -1
```

A única exceção é o tipo `char`, por ser apenas positivo, tem intervalo:

```
0 a 2^(16) -1
```

Os tipos de ponto flutuante têm os seguintes tamanhos em notação científica:

- `float` — 4 bytes (32 bits, de +/–1.4 * 10 ^ 45 a +/–3.4028235 * 10 ^ 38);
- `double` — 8 bytes (64 bits, de +/–4.9 * 10 ^ 324 a +/–1.7976931348623157 * 10 ^ 308).

Todos os números de ponto flutuante também podem assumir os seguintes valores:

- +/– infinity
- +/- 0
- NaN (Not a Number)

Literais

Ao escrever nosso código, muitas vezes o programador coloca os valores das variáveis diretamente no código-fonte. Quando isso ocorre, dizemos que o valor foi literalmente escrito no código, ou

seja, é um **valor literal**.

Todos os valores primitivos maiores que `int` podem ser expressos literalmente. Por outro lado, as referências (valores não primitivos) não podem ser expressas de maneira literal (não conseguimos colocar direto os endereços de memória dos objetos).

Ao inicializar uma variável, podemos explicitar que queremos que ela seja do tipo `double` ou `long` usando a letra específica:

```
// compila pois 737821237891232 é um double válido
System.out.println(737821237891232d);

// compila pois 737821237891232 é um long válido
System.out.println(737821237891232l);

// não compila pois 737821237891232 é um valor maior que
// o int aceita
System.out.println(737821237891232);
```

Da mesma maneira, o compilador é um pouco esperto e percebe se você tenta quebrar o limite de um `int` muito facilmente:

```
// compila pois 7378212378912321 é um long válido
long l = 7378212378912321;

// não compila, pois nem todo long cabe em um int
int i = l;
```

Alguns outros exemplos de inicialização com literal são:

```
// boolean
System.out.println(true); // booleano verdadeiro
System.out.println(false); // booleano falso

// números simples são considerados inteiros
System.out.println(1); // int
```

```
// números com casa decimal são considerados double.
// Também podemos colocar uma letra "D" ou "d" no final
System.out.println(1.0); //double
System.out.println(1.0D); //double

// números inteiros com a letra "L" ou "l"
// no final são considerados long.
System.out.println(1L); //long

// números com casa decimal com a letra "F" ou "f"
// no final são considerados float.
System.out.println(1.0F); //float
```

Bases diferentes

No caso dos números inteiros, podemos declarar usando bases diferentes. O Java suporta a base **decimal** e mais as bases **octal**, **hexadecimal** e **binária**.

Um número na base octal tem de começar com um zero à esquerda e pode usar apenas os algarismos de 0 a 7:

```
int i = 0761; // octal

System.out.println(i); // 497
```

E na hexadecimal, começa com `0x` ou `0X` e usa os algarismos de 0 a 15. Como não existe um algarismo "15", usamos letras para representar algarismos de "10" a "15", no caso, "A" a "F", maiúsculas ou minúsculas:

```
int j = 0xAB3400; // hexadecimal
System.out.println(j); // 11219968
```

Já na base binária (`binary`), começamos com `0b` ou `0B` , e só podemos usar "0" e "1":

```
int b1 = 0b100001011; // binary
System.out.println(b1); // 267
```

```
int b2 = 0B100001010; // binary
System.out.println(b2); // 266
```

Não é necessário aprender a fazer a conversão entre as diferentes bases e a decimal. Apenas saber quais são os valores possíveis em cada base, para identificar erros de compilação como o que segue. Na base octal, o algarismo 9 não existe, portanto, o código a seguir não compila:

```
int i = 0769; // compile error
int j = 011; // 9
```

Notação científica

Ao declarar `doubles` ou `floats`, podemos usar a notação científica:

```
double d = 3.1E2;
System.out.println(d); // 310.0

float e = 2e3f;
System.out.println(e); // 2000.0

float f = 1E4F;
System.out.println(f); // 10000.0
```

Usando underlines em literais

A partir do Java 7, existe a possibilidade de usarmos *underlines* (_) quando estamos declarando literais para facilitar a leitura do código:

```
int a = 123_456_789;
```

Existem algumas regras sobre onde esses *underlines* podem ser posicionados nos literais e, caso sejam colocados em locais errados, resultam em erros de compilação. A regra básica é que eles só

podem ser posicionados com **valores numéricos em ambos os lados**. Vamos ver alguns exemplos:

```
int v1 = 0_100_267_760;         // ok
int v2 = 0_x_4_13;              // erro, _ antes e depois do x
int v3 = 0b_x10_BA_75;          // erro, _ depois do b
int v4 = 0b_10000_10_11;      // erro, _ depois do b
int v5 = 0xa10_AF_75;           // ok, apesar de ser letra
                               // representa dígito
int v6 = _123_341;              // erro, inicia com _
int v7 = 123_432_;              // erro, termina com _
int v8 = 0x1_0A0_11;           // ok
int v9 = 144__21_12;           // ok
```

A mesma regra se aplica a números de ponto flutuante:

```
double d1 = 345.45_e3;          // erro, _ antes do e
double d2 = 345.45e_3;          // erro, _ depois do e
double d3 = 345.4_5e3;          // ok
double d4 = 34_5.45e3_2;      // ok
double d5 = 3_4_5.4_5e3;      // ok
double d6 = 345._45F;           // erro, _ depois do .
double d7 = 345_.45;           // erro, _ antes do .
double d8 = 345.45_F;           // erro, _ antes do indicador de
                               // float
double d9 = 345.45_d;           // erro, _ antes do indicador de
                               // double
```

Iniciando chars

Os `chars` são iniciados colocando o caractere desejado entre **aspas simples**:

```
char c = 'A';
```

Mas podemos iniciar com números também. Neste caso, o número representa a posição do caractere na tabela unicode:

```
char c = 65;
System.out.println(c); // A
```

Não é necessário decorar a tabela unicode, mas é preciso prestar atenção a pegadinhas, como a seguinte:

```
char sete = 7; // número, pois não está entre aspas simples
System.out.println(sete);  // Não imprime nada!!!!
```

Quando usando programas em outras línguas, às vezes queremos usar caracteres unicode, mas não temos um teclado com tais teclas (árabe, chinês etc.). Neste caso, podemos usar uma representação literal de um caractere unicode em nosso código, iniciando o `char` com `\u` :

```
char c = '\u03A9'; // unicode
System.out.println(c); // imprime a letra grega ômega
```

Identificadores

Quando escrevemos nossos programas, usamos basicamente dois tipos de termos para compor nosso código: identificadores e palavras reservadas.

Chamamos de **identificadores** as palavras definidas pelo programador para nomear variáveis, métodos, construtores, classes, interfaces etc. Já **palavras reservadas**, ou **palavras-chave**, são termos predefinidos da linguagem que podemos usar para definir comandos (`if` , `for` , `class` , entre outras).

São diversas palavras-chave na linguagem Java:

- abstract
- assert
- boolean
- break
- byte

- case
- catch
- char
- class
- const
- continue
- default
- do
- double
- else
- enum
- extends
- final
- finally
- float
- for
- goto
- if
- implements
- import
- instanceof
- int
- interface
- long
- native
- new
- package
- private
- protected
- public

- return
- short
- static
- strictfp
- super
- switch
- synchronized
- this
- throw
- throws
- transient
- try
- void
- volatile
- while

NULL, FALSE E TRUE

Outras três palavras reservadas que não aparecem nessa lista são `true`, `false` e `null`. Mas, segundo a especificação na linguagem Java, esses três termos são considerados *literais* e não palavras-chave (embora também sejam reservadas), totalizando 53 palavras reservadas. Para mais, acesse: http://bit.ly/java-keywords.

Identificadores válidos devem seguir as regras:

- Não podem ser igual a uma palavra-chave;
- Podem usar letras (unicode), números, `$` e `_`;

- O primeiro caractere **não** pode ser um número;
- Podem possuir qualquer número de caracteres.

Os identificadores são *case sensitive*, ou seja, respeitam maiúsculas e minúsculas:

```
int aName; // ok
int aname;    // ok, diferente do anterior
int _num;    // ok
int $_ab_c; // ok
int x_y;    // ok
int false;    // inválido, palavra reservada
int x-y;     // inválido, traço
int 4num;     // inválido, começa com número
int av#f;     // inválido, #
int num.spc; // inválido, ponto no meio
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int
        age
        = 100;
        System.out.println(age);
    }
}
```

a) O código não compila: erros a partir da linha que define uma variável do tipo `int`.

b) O código não compila: a variável `age` não foi inicializada, mas foi usada em `System.out.println`.

c) O código compila e imprime 0.

d) O código compila e imprime 100.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int age;
        if(args.length > 0) {
            age = Integer.parseInt(args[0]);
        } else {
            System.err.println("???");
        }
        System.out.println(age);
    }
}
```

a) Não compila: erro na linha que tenta acessar a variável `age`.

b) Compila e imprime 0, ou a idade que for passada na linha de comando.

c) Compila e imprime a idade que for passada na linha de comando.

d) Compila e imprime "???", ou imprime a idade.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int age;
        if(args.length > 0) {
            age = Integer.parseInt(args[0]);
        } else {
            System.err.println("???");
            return;
        }
        System.out.println(age);
    }
```

```
}
```

a) Não compila: erro na linha que tenta acessar a variável `age`.

b) Compila e imprime 0, ou a idade que for passada na linha de comando.

c) Compila e imprime a idade que for passada na linha de comando.

d) Compila e imprime a mensagem de erro, ou imprime a idade.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        boolean array = new boolean[300];
        System.out.println(array[3]);
    }
}
```

a) Imprime `true`.

b) Imprime `false`.

c) Imprime `0`.

d) Imprime `-1`.

e) Imprime `null`.

f) Não compila.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
```

```
    public static void main(String[] args) {
        boolean[] array = new boolean[300];
        System.out.println(array[3]);
    }
}
```

a) Imprime `true`.

b) Imprime `false`.

c) Imprime `0`.

d) Imprime `-1`.

e) Imprime `null`.

f) Não compila.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        boolean argis;
        if(args.length > 0)
            argis = 1;
        else
            argis = 0;
        System.out.println(argis);
    }
}
```

a) Não compila: o método de impressão não recebe `boolean`.

b) Não compila: atribuição inválida.

c) Não compila: o método `length` de array não é uma propriedade.

d) Não compila: o método `length` de `String[]` não é uma

propriedade.

e) Compila e imprime `0` ou `1`.

f) Compila e imprime `false` ou `true`.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int n = 09;
        int m = 03;
        int x = 1_000;
        System.out.println(x - n + m);
    }
}
```

a) Não compila: erro na linha que declara `n`.

b) Não compila: erro na linha que declara `x`.

c) Não compila: erro na linha que declara `m`.

d) Compila e imprime um número menor que 1000.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        for(char c='a';c <= 'z';c++) {
            System.out.println(c);
        }
    }
}
```

a) Não compila: não podemos somar um em um caractere.

b) Não compila: não podemos comparar caracteres com `<`.

c) Compila e imprime o alfabeto entre a e z, inclusive.

9) Qual das palavras a seguir não é reservada em Java?

a) strictfp

b) native

c) volatile

d) transient

e) instanceOf

10) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        boolean BOOLEAN = false;
        if(BOOLEAN) {
            System.out.println("Yes");
        }
    }
}
```

a) Não compila: não podemos declarar uma variável com o nome de uma palavra reservada.

b) Não compila: não podemos declarar uma variável iniciando com letras maiúsculas.

c) Compila e roda, imprimindo `Yes`.

d) Compila e roda, não imprimindo nada.

2.2 VARIÁVEIS DE REFERÊNCIAS A OBJETOS

E TIPOS PRIMITIVOS

As variáveis de tipos primitivos de fato armazenam os valores (e não ponteiros/referências). Ao se atribuir o valor de uma variável primitiva a uma outra variável, o valor é copiado, e o original não é alterado:

```
int a = 10;
int b = a; // copiando o valor de a para b
b++; // somando 1 em b
System.out.println(a); // continua com 10.
```

Os programas construídos com o modelo orientado a objetos utilizam, evidentemente, objetos. Para acessar um atributo ou invocar um método de qualquer objeto, é necessário que tenhamos armazenada uma **referência** para ele.

Uma variável de referência é um ponteiro para o endereço de memória, no qual o objeto se encontra. Ao atribuirmos uma variável de referência a outra, estamos copiando a referência, ou seja, fazendo com que as duas variáveis apontem para o mesmo objeto, e não criando um novo objeto:

```
class Car {
    int age;
}

class Test{
    public static void main(String[] args){
        Car a = new Car();
        Car b = a; // agora b aponta para o mesmo objeto de a

        a.age = 5;

        System.out.println(b.age); // imprime 5
    }
}
```

Duas referências são consideradas iguais somente se elas estão apontando para o mesmo objeto. Mesmo que os objetos que elas apontem sejam iguais, ainda são referências para objetos diferentes:

```
Car a = new Car();
a.age = 5;

Car b = new Car();
b.age = 5;

Object c = a;

System.out.println(a == b); // false
System.out.println(a == c); // true
```

Veremos bastante sobre comparação de tipos primitivos e de referências mais à frente.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int x = 15;
        int y = x;
        y++;
        x++;
        int z = y;
        z--;
        System.out.println(x + y + z);
    }
}
```

a) Imprime 43.

b) Imprime 44.

c) Imprime 45.

d) Imprime 46.

e) Imprime 47.

f) Imprime 48.

g) Imprime 49.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    int v = 15;
}
class A {
    public static void main(String[] args) {
        B x = new B();
        B y = x;
        y.v++;
        x.v++;
        B z = y;
        z.v--;
        System.out.println(x.v + y.v + z.v);
    }
}
```

a) Imprime 43.

b) Imprime 44.

c) Imprime 45.

d) Imprime 46.

e) Imprime 47.

f) Imprime 48.

g) Imprime 49.

2.3 LEIA OU ESCREVA PARA CAMPOS DE OBJETOS

Ler e escrever propriedades em objetos é uma das tarefas mais comuns em um programa Java. Para acessar um atributo, usamos o operador . (ponto), junto a uma variável de referência para um objeto. Veja a seguinte classe:

```
class Car {
    String model;
    int year;

    public Car() { year = 2014; }

    public String getData() {
        return model + " - " + year;
    }

    public void setModel(String m) {
        this.model = m;
    }
}
```

Vamos escrever um código para usar essa classe:

```
Car a = new Car();
a.model = "Ferrari";    // A. acessando diretamente o atributo
a.setModel("Ferrari");  // B. acessando o atributo por um método

// acessando o método e passando o retorno como argumento para
// o método println
System.out.println(a.getData());

}
```

As linhas A e B têm exatamente o mesmo efeito. Como

iniciamos o valor da propriedade `year` no construtor, ao chamar o método `getData`, o valor `2014` é exibido junto ao nome do model.

Quando estamos dentro da classe, não precisamos de nenhum operador para acessar os atributos de instância da classe. Opcionalmente, podemos usar a palavra-chave `this`, que serve como uma variável de referência para o próprio objeto onde o código está sendo executado:

```
class Car{
    int year;
    String model;

    public Car(){
        model = "???";      // acessando variável de
                                // instancia sem o this
        this.year = 2014;        // acessando com o this.
    }
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    int c;
    void c(int c) {
        c = c;
    }
}
class A {
    public static void main(String[] args) {
        B b = new B();
        b.c = 10;
        System.out.println(b.c);
        b.c(30);
        System.out.println(b.c);
    }
```

```
}
```

a) Não compila: conflito de nome de variável membro e método em `B`.

b) Não compila: conflito de nome de variável membro e variável local em `B`.

c) Compila e roda, imprimindo 10 e 30.

d) Compila e roda, imprimindo outro resultado.

2.4 EXPLIQUE O CICLO DE VIDA DE UM OBJETO

O ciclo de vida dos objetos Java está dividido em três fases distintas. Vamos conhecê-las e entender o que cada uma significa.

Criação de objetos

Toda vez que usamos o operador `new`, estamos criando uma nova instância de um objeto na memória:

```
class Person {
    String name;
}

class Test {
    public static void main(String[] args) {
        Person p = new Person(); // criando um novo objeto do
                                 // tipo Person
    }
}
```

Repare que há uma grande diferença entre criar um objeto e declarar uma variável. A variável é apenas uma referência, um

ponteiro, não contém um objeto de verdade.

```
// Apenas declarando a variável,
// nenhum objeto foi criado aqui
Person p;

// Agora um objeto foi criado e atribuído a variável
p = new Person();
```

Objeto acessível

A partir do momento em que um objeto foi criado e atribuído a uma variável, dizemos que o objeto está **acessível**, ou seja, podemos usá-lo em nosso programa:

```
Person p = new Person(); // criação
p.name = "Mário"; // acessando e usando o objeto
```

Objeto inacessível

Um objeto é acessível enquanto for possível "alcançá-lo" através de alguma referência direta ou indireta. Caso não exista nenhum caminho direto ou indireto para acessar esse objeto, ele se torna **inacessível**.

```
Person p = new Person();
p.name = "Mário";

// atribuímos a p o valor null
// o objeto não está mais acessível
p = null; // A

// criando um objeto sem variável
new Person(); // B
```

Nesse código, criamos um objeto do tipo `Person` e o atribuímos à variável `p`. Na linha **A**, atribuímos `null` a `p`. O que acontece com o objeto anterior? Ele simplesmente não pode

mais ser acessado por nosso programa, pois não temos nenhum ponteiro para ele. O mesmo pode ser dito do objeto criado na linha **B**. Após essa linha, não conseguimos mais acessar esse objeto.

Tome cuidado: se a prova perguntar quantos objetos foram criados no código anterior, temos a criação de duas pessoas e uma `String`, totalizando três objetos.

Outra maneira de ter um objeto inacessível é quando o escopo da variável que aponta para ele termina:

```
int value = 100;
if( value > 50) {
    Person p = new Person();
    p.name = "Guilherme";
} // Após esta linha, o objeto do tipo Person não está mais
  // acessível
```

Garbage Collector

Todo objeto inacessível é considerado elegível para o *garbage collector*. Algumas questões da prova perguntam quantos objetos são elegíveis ao garbage collector ao final de algum trecho de código:

```
public class Bla {
     int b;
     public static void main(String[] args) {
         Bla b;
        for (int i = 0; i < 10; i++) {
             b = new Bla();
             b.b = 10;
         }
           System.out.println("end"); // A
     }
 }
```

Ao chegar na linha `A`, temos **9** objetos elegíveis do tipo `Bla`

para o Garbage Collector.

> **Objetos elegíveis X Objetos coletados**
>
> O *garbage collector* roda em segundo plano juntamente com sua aplicação Java. Não é possível prever quando ele será executado, portanto, não se pode dizer com certeza quantos objetos foram efetivamente coletados em um certo ponto da aplicação. O que podemos determinar é quantos objetos são elegíveis para a coleta. A prova pode tentar se aproveitar do descuido do desenvolvedor aqui: nunca temos certeza de quantos objetos passaram pelo garbage collector, logo, somente indique quantos estão passíveis de serem coletados.

Por fim, é importante ver um exemplo de referência indireta, no qual nenhum objeto pode ser "garbage coletado":

```
import java.util.*;
class Car {

}
class Cars {
    List<Car> all = new ArrayList<Car>();
}
class Test {
    public static void main(String args[]) {
        Cars cars = new Cars();
        for(int i = 0; i < 100; i++)
            cars.all.add(new Car());
        // até essa linha todos ainda podem ser alcançados
    }
}
```

Nesse código, por mais que tenhamos criados 100 carros e um

objeto do tipo `Cars`, nenhum deles pode ser garbage coletado, pois todos podem ser alcançados direta ou indiretamente através de nossa *thread* principal.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{

}
class A {
    public static void main(String[] args) {
        B b;
        for(int i = 0;i < 10;i++)
            b = new B();
        System.out.println("end!");
    }
}
```

a) Não compila.

b) Compila e garbage coleta 10 objetos do tipo `B` na linha do `System.out`.

c) Compila e não podemos falar quantos objetos do tipo `B` foram garbage coletados na linha do `System.out`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{

}
class A {
    public static void main(String[] args) {
        B b = new B();
        for(int i = 0;i < 10;i++)
```

```
            b = new B();
        System.out.println("end!");
    }
}
```

a) Não compila.

b) Compila e 10 objetos do tipo `B` podem ser garbage coletados ao chegar na linha do `System.out`.

c) Compila e 11 objetos do tipo `B` podem ser garbage coletados ao chegar na linha do `System.out`.

d) Compila e garbage coleta 11 objetos do tipo `B` na linha do `System.out`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{

}
class A {
    public static void main(String[] args) {
        B[] bs = new B[100];
        System.out.println("end!");
    }
}
```

a) Compila e 100 objetos do tipo `B` são criados, mas não podemos falar nada sobre o garbage collector ter jogado os objetos fora na linha do `System.out`.

b) Compila e nenhum objeto do tipo `B` é criado.

c) Compila, cria 100 e joga fora todos os objetos do tipo `B` ao chegar no `System.out`.

2.5 CHAME MÉTODOS EM OBJETOS

Além de acessar atributos, também podemos invocar métodos em um objeto. Para isso usamos o operador `.` (ponto), junto a uma variável de referência para um objeto. Deve-se prestar atenção ao número e tipo de parâmetros do método, além do seu retorno. Métodos declarados como `void` não possuem retorno, logo, não podem ser atribuídos a nenhuma variável ou passado para outro método como parâmetro:

```
class Person{

    String name;

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }
}

class Test{
    public static void main(String[] args){
        Person p = new Person();

        //chamando método na variável de ref.
        p.setName("Mario");

        //Atribuindo o retorno do método a variável.
        String name = p.getName();

        // erro, método é void
        String a = p.setName("X");
    }
}
```

Quando um método está sendo invocado em um objeto,

podemos chamar outro método no mesmo objeto através da invocação direta ao nome do método:

```
class A {
    void method1() {
        method2();
    }
    void method2() {
    }
}
```

Argumentos variáveis: varargs

A partir do Java 5, **varargs** possibilitam um método que receba um número variável (não fixo) de parâmetros. É a maneira de receber um array de objetos e possibilitar uma chamada mais fácil do método.

Podemos chamá-lo com qualquer número de argumentos:

```
class Calculator{
    public int sum(int... nums){
        int total = 0;
        for (int a : nums){
            total+= a;
        }
        return total;
    }

}
```

`nums` realmente é um array aqui. Você pode fazer um `for` usando o `length`, ou mesmo usar o `enhanced for`. A invocação desse método pode ser feita de várias maneiras, todas os exemplos a seguir são válidos e compilam:

```
    public static void main (String[] args){
        Calculator c = new Calculator();
```

```
        System.out.println(c.sum()); // 0
        System.out.println(c.sum(1)); // 1
        System.out.println(c.sum(1,2)); // 3
        System.out.println(c.sum(1,2,3,4,5,6,7,8,9)); // 45
    }
```

Em todos os casos, um array será criado, e `null` nunca será passado. Um parâmetro `varargs` deve ser sempre o último da assinatura do método para evitar ambiguidade. Isso implica que apenas um dos parâmetros de um método seja `varargs`. E repare que os argumentos variáveis têm de ser do mesmo tipo.

Será dada a prioridade para o método que já podia existir antes no Java 1.4:

```
void method(int ... x) { }
void method(int x) {}

method(5);
```

Isso vai invocar o segundo método. Podemos também passar um array de `ints` para um método que recebe um `varargs`:

```
void method(int ... x) { }

method(new int[] {1,2,3,4});
```

Mas nunca podemos chamar um método que recebe array como se ele fosse `varargs`:

```
void method(int[] x) { }

method(1,2,3); // compile error
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    void x() {
        System.out.println("empty");
    }
    void x(String... args) {
        System.out.println(args.length);
    }
}
class C {
    void x(String... args) {
        System.out.println(args.length);
    }
    void x() {
        System.out.println("empty");
    }
}
class A {
    public static void main(String[] args) {
        new B().x();
        new C().x();
    }
}
```

a) Não compila: conflito entre método com `varargs` e sem argumentos.

b) Compila e imprime empty/empty.

c) Compila e imprime empty/0.

d) Compila e imprime 0/empty.

e) Compila e imprime 0/0.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    void x(int... x) {
        System.out.println(x.length);
    }
}
```

```
class A {
    public static void main(String[] args) {
        new B().x(23789,673482);
    }
}
```

a) Não compila: `varargs` tem método e não atributo `length`.

b) Compila e, ao rodar, imprime os dois números.

c) Compila e, ao rodar, imprime 2.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    void x(int... x) {
        System.out.println(x.length);
    }
}
class A {
    public static void main(String[] args) {
        new B().x(new int[]{23789,673482});
    }
}
```

a) Não compila: `varargs` tem método e não atributo `length`.

b) Não compila: não podemos passar um array para um `varargs`.

c) Compila e, ao rodar, imprime os dois números.

d) Compila e, ao rodar, imprime 1.

e) Compila e, ao rodar, imprime 2.

4) Escolha a opção adequada ao tentar compilar e rodar o

arquivo a seguir:

```
class B{
    void x(Object... x) {
        System.out.println(x.length);
    }
}
class A {
    public static void main(String[] args) {
        new B().x(new Object[]{23789,673482});
    }
}
```

a) Não compila: `varargs` tem método e não atributo `length`.

b) Não compila: não podemos passar um array para um `varargs`.

c) Compila e, ao rodar, imprime os dois números.

d) Compila e, ao rodar, imprime 1.

e) Compila e, ao rodar, imprime 2.

2.6 MANIPULE DADOS USANDO A CLASSE STRINGBUILDER E SEUS MÉTODOS

Para suportar Strings mutáveis, o Java possui as classes `StringBuffer` e `StringBuilder`. A operação mais básica é o `append` que permite concatenar ao mesmo objeto:

```
StringBuffer sb = new StringBuffer();
sb.append("Caelum");
sb.append(" - ");
sb.append("Alura - Casa do Código");

System.out.println(sb);  // Caelum - Alura - Casa do Código
```

Repara que o `append` não devolve novos objetos como em `String`, mas altera o próprio `StringBuffer`, que é mutável.

Podemos criar um objeto desse tipo de diversas maneiras diferentes:

```
// vazio
StringBuilder sb1 = new StringBuilder();
// conteudo inicial
StringBuilder sb2 = new StringBuilder("java");
// tamanho inicial do array para colocar a string
StringBuilder sb3 = new StringBuilder(50);
// baseado em outro objeto do mesmo tipo
StringBuilder sb4 = new StringBuilder(sb2);
```

Tenha cuidado: ao definir o tamanho do array, não estamos criando uma `String` de tamanho definido, somente um array desse tamanho que será utilizado pelo `StringBuilder`, portanto:

```
StringBuilder sb3 = new StringBuilder(50);
System.out.println(sb3); // linha em branco
System.out.println(sb3.length()); // 0
```

As classes `StringBuffer` e `StringBuilder` têm exatamente a mesma interface (mesmos métodos), sendo que a primeira é *thread-safe* e a última não (e foi adicionada no Java 5). Quando não há compartilhamento entre threads, use sempre que possível a `StringBuilder`, que é mais rápida por não precisar se preocupar com *locks*.

Inclusive, em Java, quando fazemos concatenação de Strings usando o `+`, por baixo dos panos, é usado um `StringBuilder`. Não existe a operação `+` na classe `String`. O compilador troca todas as chamadas de concatenação por `StringBuilder`s (podemos ver isso no bytecode compilado).

Principais métodos de StringBuffer e StringBuilder

Há a família de métodos `append` com overloads para receber cada um dos primitivos, Strings, arrays de chars, outros `StringBuffer` etc. Todos eles devolvem o próprio `StringBuffer` / `Builder` o que permite chamadas encadeadas:

```
StringBuffer sb = new StringBuffer();
sb.append("Caelum").append(" - ").append("Ensino e Inovação");
System.out.println(sb);  // Caelum - Ensino e Inovação
```

O método `append` possui uma versão que recebe `Object` e chama o método `toString` de seu objeto.

Há ainda os métodos `insert` para inserir coisas no meio. Há versões que recebem primitivos, Strings, arrays de char etc. Mas todos têm o primeiro argumento recebendo o índice onde queremos inserir:

```
StringBuffer sb = new StringBuffer();
sb.append("Caelum - Inovação");
sb.insert(9, "Ensino e ");

System.out.println(sb); // Caelum - Ensino e Inovação
```

Outro método que modifica é o `delete`, que recebe os índices inicial e final:

```
StringBuffer sb = new StringBuffer();
sb.append("Caelum - Ensino e Inovação");
sb.delete(6, 15);

System.out.println(sb); // Caelum e Inovação
```

Para converter um `StringBuffer` / `Builder` em `String`, basta chamar o `toString` mesmo. O método `reverse` inverte seu conteúdo:

```
System.out.println(new StringBuffer("guilherme").reverse());
                                                    // emrehliug
```

Fora esses, também há o `trim`, `charAt`, `length()`, `equals`, `indexOf`, `lastIndexOf` e `substring`.

Cuidado, pois o método `substring` não altera o valor do seu `StringBuilder` ou `StringBuffer`, mas retorna a `String` que você deseja. Existe também o método `subSequence` que recebe o início e o fim e funciona da mesma maneira que o `substring` com dois argumentos.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder();
        sb.append("guilherme").delete(2,3);
        System.out.println(sb);
    }
}
```

a) O código não compila: erro na linha que tenta imprimir o `StringBuilder`.

b) O código compila e imprime `glherme`.

c) O código compila e imprime `guherme`.

d) O código compila e imprime `gilherme`.

e) O código compila e imprime `gulherme`.

2) Escolha a opção adequada ao tentar compilar e rodar o

arquivo a seguir:

```
class A {
    public static void main(String[] args) {
      StringBuilder sb = new StringBuilder("guilherme");
      System.out.println(sb.indexOf("e") + sb.lastIndexOf("e"));
      System.out.println(sb.indexOf("k") + sb.lastIndexOf("k"));
    }
}
```

a) O código imprime 13 e -2.

b) O código imprime 13 e 0.

c) O código imprime 13 e -1.

d) O código imprime 13 e 8.

e) O código imprime 13 e 10.

f) O código imprime 15 e -2.

g) O código imprime 15 e 0.

h) O código imprime 15 e -1.

i) O código imprime 15 e 8.

j) O código imprime 15 e 10.

2.7 CRIANDO E MANIPULANDO STRINGS

Existem duas maneiras tradicionais de criar uma `String`, uma implícita e outra explícita:

```
String implicit = "Java";
String explicit = new String("Java");
```

A comparação entre esses dois tipos de criação de `Strings` é feita na seção *Test equality between strings and other objects using == and equals()*.

Existem outras maneiras não tão comuns, como através de uma array:

```
char[] name = new char[]{'J', 'a', 'v', 'a'};
String fromArray = new String(name);
```

Ou ainda podemos criar uma `String` baseada em um `StringBuilder` ou `StringBuffer`:

```
StringBuilder sb1 = new StringBuilder("Java");
String nameBuilder = new String(sb1);

StringBuffer sb2 = new StringBuffer("Java");
String nameBuffer = new String(sb2);
```

Como uma `String` não é um tipo primitivo, ela pode ter valor `null`, lembre-se disso:

```
String name = null; // explicit null
```

Podemos concatenar `Strings` com o `+`:

```
String name = "Java" + " " + "Exam";
```

Caso tente concatenar `null` com uma `String`, temos a conversão de `null` para `String`:

```
String nulled = null;
System.out.println("value: " + nulled); // value: null
```

E o contrário também tem o mesmo resultado:

```
String nulled = null;
System.out.println(nulled + " value"); // null value
```

O Java faz a conversão de tipos primitivos para `Strings`

automaticamente:

```
String name = "Java" + ' ' + "Certification" + ' ' + 1500;
System.out.println(name); // Java Certification 1500

String name2 = "Certification";
name2 += ' ' + "Java" + ' ' + 1500;
System.out.println(name2); // Certification Java 1500
```

Lembre-se da precedência de operadores. O exemplo a seguir mostra o código sendo interpretado da esquerda pra direita (primeiro a soma):

```
String value = 15 + 00 + " certification";
System.out.println(value); // 15 certification
```

Strings são imutáveis

O principal ponto sobre Strings é que elas são imutáveis:

```
String s = "caelum";
s.toUpperCase();
System.out.println(s);
```

Esse código imprime `caelum` em minúscula. Isso porque o método `toUpperCase` não altera a `String` original. Na verdade, se olharmos o javadoc da classe `String`, vamos perceber que **todos os métodos que parecem modificar uma String na verdade devolvem uma nova**.

```
String s = "caelum";
String s2 = s.toUpperCase();
System.out.println(s2);
```

Agora sim imprimirá `CAELUM`, uma nova String. Ou, usando a mesma *referência*:

```
String s = "caelum";
s = s.toUpperCase();
```

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

Para tratarmos de "strings mutáveis", usamos as classes `StringBuffer` e `StringBuilder`.

Lembre-se de que a `String` possui um array por trás e, seguindo o padrão do Java, suas posições começam em 0:

```
// 0=g, devolve 'g'
char caracter0 = "guilherme".charAt(0);

// 0=g 1=u, devolve 'u'
char caracter1 = "guilherme".charAt(1);

// 0=g 1=u 2=i, devolve 'i'
char character2 = "guilherme".charAt(2);
```

Cuidado ao acessar uma posição indevida, você pode levar um `StringIndexOutOfBoundsException` (atenção ao nome da `Exception`, não é `ArrayIndexOutofBoundsException`):

```
char character20 = "guilherme".charAt(20); // exception
char characterNegative = "guilherme".charAt(-1); // exception
```

Principais métodos de String

O método `length` imprime o tamanho da `String`:

```
String s = "Java";
System.out.println(s.length()); // 4
System.out.println(s.length); // não compila: não é atributo
System.out.println(s.size()); // não compila: não existe size
                              // em String Java
```

Já o método `isEmpty` diz se a `String` tem tamanho zero:

```
System.out.println("".isEmpty()); // true
System.out.println("java".isEmpty()); // false
System.out.println(" ".isEmpty()); // false
```

Devolvem uma nova String:

- `String toUpperCase()` — tudo em maiúscula;
- `String toLowerCase()` — tudo em minúsculo;
- `String trim()` — retira espaços em branco no começo e no fim;
- `String substring(int beginIndex, int endIndex)` — devolve a substring a partir dos índices de começo e fim;
- `String substring(int beginIndex)` — semelhante ao anterior, mas toma a substring a partir do índice passado até o final da String;
- `String concat(String)` — concatena o parâmetro ao fim da String atual e devolve o resultado;
- `String replace(char oldChar, char newChar)` — substitui todas as ocorrências de determinado `char` por outro;
- `String replace(CharSequence target, CharSequence replacement)` — substitui todas as ocorrências de determinada `CharSequence` (como String) por outra.

O método `trim` limpa caracteres em branco nas duas pontas da `String`:

```
System.out.println("     ".trim()); // imprime só a quebra de
                                    // linha do println
System.out.println(" ".trim().isEmpty()); // true
System.out.println(" guilherme ".trim()); // imprime 'guilherme'
System.out.println(". .".trim()); // imprime '. .'
```

O método `replace` substituirá todas as ocorrências de um texto por outro:

```
System.out.println("java".replace("j", "J")); // Java
System.out.println("guilherme".replace("e", "i")); // guilhirmi
```

Podemos sempre fazer o *chaining* e criar uma sequência de "transformações" que retornam uma nova `String`:

```
String parsed = "  My Java
                    Certification!  ".toUpperCase().trim();

System.out.println(parsed); // "MY JAVA CERTIFICATION!"
```

Para extrair pedaços de uma `String`, usamos o método `substring`. Cuidado ao usar o método `substring` com valores inválidos, pois eles jogam uma `Exception`. O segredo do método `susbtring` é que ele não inclui o caractere da posição final, mas inclui o caractere da posição inicial:

```
String text = "Java";

// ava
System.out.println(text.substring(1));

// StringIndexOutOfBoundsException
System.out.println(text.substring(-1));

// StringIndexOutOfBoundsException
System.out.println(text.substring(5));

// Java
System.out.println(text.substring(0, 4));

// ava
System.out.println(text.substring(1, 4));

// Jav
System.out.println(text.substring(0, 3));

// StringIndexOutOfBoundsException
System.out.println(text.substring(0, 5));

// StringIndexOutOfBoundsException
System.out.println(text.substring(-1, 4));
```

Comparação:

- `boolean equals(Object)` — compara igualdade caractere a caractere (herdado de `Object`);
- `boolean equalsIgnoreCase(String)` — compara caractere a caractere, ignorando maiúsculas/minúsculas;
- `int compareTo(String)` — compara as 2 Strings por ordem lexicográfica (vem de `Comparable`);
- `int compareToIgnoreCase(String)` — compara as 2 Strings por ordem lexicográfica, ignorando maiúsculas/minúsculas.

E aqui, todas as variações desses métodos. Não precisa saber o número exato que o `compareTo` retorna. Basta saber que: será negativo caso a `String` na qual o método for invocado vier antes; zero se for igual; positivo se vier depois do parâmetro passado.

```
String text = "Certification";
System.out.println(text.equals("Certification")); // true
System.out.println(text.equals("certification")); // false
System.out.println(text.equalsIgnoreCase("CerTifIcatIon"));//true

System.out.println(text.compareTo("Aim")); // 2
System.out.println(text.compareTo("Certification")); // 0
System.out.println(text.compareTo("Guilherme")); // -4

System.out.println(text.compareTo("certification")); // -32

System.out.println(text.compareToIgnoreCase("certification"));//0
```

Buscas simples:

- `boolean contains(CharSequence)` — devolve `true` se a String contém a sequência de `chars`;
- `boolean startsWith(String)` — devolve `true` se começa com a String do parâmetro;
- `boolean endsWith(String)` — devolve `true` se termina com a String do parâmetro;

- `int indexOf(char)` e `int indexOf(String)` — devolvem o índice da primeira ocorrência do parâmetro;
- `int lastIndexOf(char)` e `int lastIndexOf(String)` — devolvem o índice da última ocorrência do parâmetro.

O código a seguir exemplifica todos os casos desses métodos:

```
String text = "Pretendo fazer a prova de certificação de Java";

System.out.println(text.indexOf("Pretendo")); // imprime 0
System.out.println(text.indexOf("Pretendia")); // imprime -1
System.out.println(text.indexOf("tendo")); // imprime 3

System.out.println(text.indexOf("a")); // imprime 10
System.out.println(text.lastIndexOf("a")); // imprime 45
System.out.println(text.lastIndexOf("Pretendia")); //imprime -1

System.out.println(text.startsWith("Pretendo")); // true
System.out.println(text.startsWith("Pretendia")); // false

System.out.println(text.endsWith("Java")); // true
System.out.println(text.endsWith("Oracle")); // false
```

Exercícios

1) Considere o seguinte código dentro de um `main` :

```
class A{
    public static void main(String [] args){
        String s = "aba";
        for(int i = 0; i < 9; i++) {
            s = s +"aba";
        }
        System.out.println(s.length);
    }
}
```

a) Não compila.

b) Compila e imprime 3.

c) Compila e imprime 30.

d) Compila e imprime 33.

e) Compila e imprime 36.

2) Dada a seguinte classe:

```
class B {
    String msg;

    void imprime() {
        if (!msg.isEmpty())
            System.out.println(msg);
        else
            System.out.println("empty");
    }
}
```

O que acontece se chamarmos `new B().imprime()`?

a) Não compila.

b) Compila, mas dá exceção na hora de rodar.

c) Compila, roda e não imprime nada.

d) Compila, roda e imprime "empty".

3) Dada a seguinte classe:

```
class B {

    void imprime() {
        String msg;
        if (!msg.isEmpty())
            System.out.println(msg);
        else
            System.out.println("empty");
    }
}
```

O que acontece se chamarmos `new B().imprime()` ?

a) Não compila.

b) Compila, mas dá exceção na hora de rodar.

c) Compila, roda e não imprime nada.

d) Compila, roda e imprime "empty".

4) Qual é a saída nos dois casos, respectivamente?

```
String s = "Caelum";
s.concat(" - Ensino e Inovação");
System.out.println(s);
```

```
StringBuffer s = new StringBuffer("Caelum");
s.append(" - Ensino e Inovação");
System.out.println(s);
```

a) `Caelum` e `Caelum - Ensino e Inovação`.

b) `Caelum - Ensino e Inovação` e `Caelum - Ensino e Inovação`.

c) `Caelum` e `Caelum`.

d) `Caelum - Ensino e Inovação` e `Caelum`.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String empty = null;
        String full = "Welcome " + empty;
        System.out.println(full);
    }
}
```

a) Não compila, pois empty é nulo.

b) Não compila por outro motivo.

c) Compila e imprime `"Welcome "` .

d) Compila e imprime `"Welcome empty"` .

e) Compila e imprime outro resultado que não foi mencionado nessas alternativas.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String empty;
        String full = "Welcome " + empty;
        System.out.println(full);
    }
}
```

a) Não compila, pois empty é nulo.

b) Não compila por outro motivo.

c) Compila e imprime `"Welcome "` .

d) Compila e imprime `"Welcome vazio"` .

e) Compila e imprime outro resultado que não foi mencionado nessas alternativas.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    String empty;
    public static void main(String[] args) {
```

```
        String full = "Welcome " + empty;
        System.out.println(full);
    }
}
```

a) Não compila, pois empty é nulo.

b) Não compila por outro motivo.

c) Compila e imprime `"Welcome "` .

d) Compila e imprime `"Welcome empty"` .

e) Compila e imprime outro resultado que não foi mencionado nessas alternativas.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    static String empty;
    public static void main(String[] args) {
        String full = "Welcome " + empty;
        System.out.println(full);
    }
}
```

a) Não compila, pois empty é nulo.

b) Não compila por outro motivo.

c) Compila e imprime `"Welcome "` .

d) Compila e imprime `"Welcome empty"` .

e) Compila e imprime outro resultado que não foi mencionado nessas alternativas.

9) Escolha a opção adequada ao tentar compilar e rodar o

arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String s = null;
        String s2 = new String(s);
        System.out.println(s2);
    }
}
```

a) Não compila ao tentar invocar o construtor.

b) Compila e não imprime nada.

c) Compila e imprime `null`.

d) Compila e dá erro de execução ao tentar criar a segunda `String`.

10) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String s = "estudando para a certificação";
        System.out.println(s.substring(3, 6));
    }
}
```

a) Não compila: caractere com acento e cedilha dentro de uma String.

b) Não compila: `substring` é `subString`.

c) Compila e imprime `"uda"`.

d) Compila e imprime `"tuda"`.

e) Compila e imprime `"tud"`.

11) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String s2 = new String(null);
        System.out.println(s2);
    }
}
```

a) Não compila ao tentar invocar o construtor.

b) Compila e não imprime nada.

c) Compila e imprime `null`.

d) Compila e dá erro de execução ao tentar criar a `String`.

12) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int val = 10;
        int div = 4;
        double res = val / div;
        System.out.println(val + div +
                           "...");
        System.out.println(res + " = result");
    }
}
```

a) Imprime os números 10, 4 e 2.5.

b) Imprime os números 14 e 2.5.

c) Nenhuma das outras alternativas.

13) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String s = "estudando para a certificação";
        s.replace("e", 'a');
        System.out.println(s);
    }
}
```

a) Não compila.

b) Compila e imprime `"estudando para a certificação"`.

c) Compila e imprime `"astudando para a cartificação"`.

d) Compila e imprime `"studando para a crtificação"`.

14) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String s = "guilherme";
        s.substring(0,2) = "gua";
        System.out.println(s);
    }
}
```

a) Erro de compilação.

b) Compila e imprime `"guilherme"`.

c) Compila e imprime `"gualherme"`.

Capítulo 3

USANDO OPERADORES E CONSTRUÇÕES DE DECISÃO

3.1 USE OPERADORES JAVA

Para manipular os valores armazenados das variáveis, tanto as primitivas quanto as não primitivas, a linguagem de programação deve oferecer operadores. Um dos operadores mais importantes é o que permite guardar um valor em uma variável. Esse operador é denominado **operador de atribuição**.

No Java, o símbolo = representa o operador de atribuição. Para atribuir um valor, precisamos de uma variável à qual será atribuído o valor, e do valor:

```
long age = ; // não compila, onde está o valor?
long = 15; // não compila, onde está o nome da variável?

long age = 15; // compila
age = 15;
// compila desde que a variável tenha sido declarada
// correta anteriormente
```

Para um valor ser atribuído a uma variável, ambos devem ser compatíveis. Um valor é compatível com uma variável se ele for do

mesmo tipo dela ou de um tipo menos abrangente.

O código a seguir inicializa uma variável com o operador de atribuição `=`:

```
int age = 30;
```

E agora utilizamos um valor do tipo `int`, atribuindo ele a variável do tipo `long`. Como `int` é menos abrangente que `long`, essa atribuição está correta.

```
long age = 30;
```

Procure sempre se lembrar dos tamanhos dos primitivos quando estiver fazendo a prova. `int` é um número médio, será que ele "cabe" em uma variável do tipo `long` (número grande)? Sim, logo o código compila. Mais exemplos:

```
int a = 10;       // tipos iguais
long b = 20;      // int cabe em um long
float c = 10f;  // tipos iguais
double d = 20.0f; // float cabe em um double
double e = 30.0; // tipos iguais
float f = 40.0; // erro, double não cabe em um float.
int g = 10l; // erro, long não cabe em int
float h = 10l; // inteiros cabem em decimais
double i = 20; // inteiros cabem em decimais
long j = 20f; // decimais não cabem em inteiros
```

A exceção a essa regra ocorre quando trabalhamos com tipos inteiros menos abrangentes que `int` (`byte`, `short` e `char`). Nesses casos, o compilador permite que atribuamos um valor inteiro, desde que compatível com o tipo:

```
byte b1 = 10;
byte b2 = 200; // não compila, estoura byte

char c1 = 10;
char c2 = -3; // não compila, char não pode ser negativo
```

Atribuição e referência

Quando trabalhamos com referências, temos de lembrar do polimorfismo:

```
List<String> names = new ArrayList<String>();
```

E no caso do Java 7, quando atribuímos com *generics*, podemos usar o operador diamante:

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

Podemos usar o operador diamante com polimorfismo:

```
List<String> lastnames = new ArrayList<>();
```

Lembre de que as atribuições em Java são por cópia de valor, sempre. No tipo primitivo, copiamos o valor, e em referências a objetos, copiamos o valor da referência (não duplicamos o objeto):

```
List<String> names = new ArrayList<>();

// copia o valor da referência, o objeto é o mesmo
List<String> names2 = names;
names2.add("Guilherme");

// true
System.out.println(names.size() == names2.size());

int age = 15;

int age2 = age; // copia o valor
age2 = 20;

System.out.println(age == age2); // false
```

Operadores aritméticos

Os cálculos sobre os valores das variáveis primitivas numéricas são feitos com os operadores aritméticos. A linguagem Java define

operadores para as principais operações aritméticas (soma, subtração, multiplicação e divisão).

```
int two = 2;
int ten = 10;

// Fazendo uma soma com o operador "+".
int twelve = two + ten;

// Fazendo uma subtração com o operador "-".
int eight = ten - two;

// Fazendo uma multiplicação com o operador "*".
int twenty = two * ten;

// Fazendo uma divisão com o operador "/".
int five = ten / two;
```

Além desses, há um operador para a operação aritmética "resto da divisão". Esse operador só faz sentido para variáveis primitivas numéricas inteiras.

```
int three = 3;
int dez = 10;

// Calculando o resto da divisão de 10 por 3.
int one = ten % three;
```

O resultado de uma operação aritmética é um valor. A dúvida que surge é qual será o tipo dele. Para descobrir o tipo do valor resultante de uma operação aritmética, devem-se considerar os tipos das variáveis envolvidas.

A regra é a seguinte: o resultado é do tipo mais abrangente entre os das variáveis envolvidas ou, no mínimo, o `int`.

```
int age = 15;
long years = 5;

// ok, o maior tipo era long
```

```
long afterThoseYears = age + years;

// não compila, o maior tipo era long, devolve long
int afterThoseYears2 = age + years;
```

Mas devemos lembrar da exceção: o mínimo é um `int`.

```
byte b = 1;
short s = 2;

// devolve no mínimo int, compila
int i = b + s;

// não compila, ele devolve no mínimo int
byte b2 = i + s;

// compila forçando o casting, correndo risco de perder
// informação
byte b2 = (byte) (i + s);
```

Divisão por zero

Dividir (ou usar `mod`) um inteiro por zero lança uma `ArithmeticException`. Se o operando for um `float` ou `double`, isso gera infinito positivo ou negativo (depende do sinal do operador). As classes `Float` e `Double` possuem constantes para esses valores.

```
int i = 200;
int v = 0;

// compila, mas exception
System.out.println(i / v);

// compila e roda, infinito positivo
System.out.println(i / 0.0);
```

Ainda existe o valor `NaN` (*Not a Number*), gerado pela radiciação de um número negativo e por algumas contas com números infinitos.

```
double positiveInfinity = 100 / 0.0;
double negativeInfinity = -100 / 0.0;

// número não definido (NaN)
System.out.println(positiveInfinity + negativeInfinity);
```

Operadores de comparação

A comparação entre os valores de duas variáveis é feita através dos operadores de comparação. O mais comum é comparar a igualdade e a desigualdade dos valores. Existem operadores para essas duas formas de comparação.

- `==` — igual
- `!=` — diferente

Além disso, os valores numéricos ainda podem ser comparados em relação à ordem.

- `>` — maior
- `<` — menor
- `>=` — maior ou igual
- `<=` — menor ou igual

Uma comparação pode devolver dois valores possíveis: verdadeiro ou falso. No Java, uma comparação sempre devolve um valor `boolean`.

```
System.out.println(1 == 1);        // true.
System.out.println(1 != 1);        // false.
System.out.println(2 < 1);        // false.
System.out.println(2 > 1);        // true.
System.out.println(1 >= 1);        // true.
System.out.println(2 <= 1);        // false.
```

Toda comparação envolvendo valores numéricos não

considera o tipo do valor. Confira somente se eles têm o mesmo valor ou não, independente de seu tipo:

```
// true.
System.out.println(1 == 1.0);

// true.
System.out.println(1 == 1);

// true. 1.0 float é 1.0 double
System.out.println(1.0f == 1.0d);

// true. 1.0 float é 1 long
System.out.println(1.0f == 1l);
```

Os valores não primitivos (referências) e os valores *boolean* devem ser comparados somente com dois comparadores, o de igualdade (`==`) e o de desigualdade (`!=`).

```
// não compila, tipo não primitivo só aceita != e ==
System.out.println("Mario" > "Guilherme");

// não compila, boolean só aceita != e ==
System.out.println(true < false);
```

Não podemos comparar tipos incomparáveis, como um `boolean` com um valor numérico. Mas podemos comparar `chars` com numéricos.

```
// não compila, boolean é boolean
System.out.println(true == 1);

// compila, 'a' tem valor numérico também
System.out.println('a' > 1);
```

Cuidado, é muito fácil comparar atribuição com comparação e uma pegadinha aqui pode passar despercebida, como no exemplo a seguir:

```
int a = 5;
```

```
System.out.println(a = 5); // não imprime true, imprime 5
```

Precisão

Ao fazer comparações entre números de ponto flutuante, devemos tomar cuidado com possíveis problemas de precisão. Qualquer conta com estes números pode causar um estouro de precisão, fazendo com que ele fique ligeiramente diferente do esperado. Por exemplo, `1 == (100.0 / 100)` pode não ser verdadeiro caso a divisão tenha uma precisão não exata.

Operadores lógicos

Muitas vezes precisamos combinar os valores `boolean`s obtidos, por exemplo, com comparações ou diretamente de uma variável. Isso é feito utilizando os operadores lógicos.

Em lógica, as operações mais importantes são: `e`, `ou`, `ou exclusivo` e `negação`.

```
System.out.println(1 == 1 & 1 > 2);    // false.
System.out.println(1 == 1 | 2 > 1);    // true.
System.out.println(1 == 1 ^ 2 > 1);    // false.
System.out.println(!(1 == 1));        // false.
```

Antes de terminar a avaliação de uma expressão, eventualmente, o resultado já pode ser descoberto. Por exemplo, quando aplicamos a operação lógica `e`, ao achar o primeiro termo falso, não precisamos avaliar o restante da expressão. Quando usamos esses operadores, sempre os dois lados da expressão são avaliados, mesmo nesses casos em que não precisariam.

Para melhorar isso, existem os operadores de curto circuito `&&` e `||`. Quando já for possível determinar a resposta final olhando apenas para a primeira parte da expressão, a segunda não é avaliada:

```
System.out.println(1 != 1 && 1 > 2);
// false, o segundo termo não é avaliado.

System.out.println(1 == 1 || 2 > 1);
// true, o segundo termo não é avaliado.
```

A maior dificuldade com operadores de curto circuito é se a segunda parte causa efeitos colaterais (um incremento, uma chamada de método). Avaliar ou não (independente da resposta) pode influenciar no resultado final do programa.

```
public static boolean method(String msg) {
    System.out.println(msg);
    return true;
}

public static void main(String[] args) {
    System.out.println(1 == 2 & method("hi"));
    // imprime hi, depois false
    System.out.println(1 == 2 && method("bye"));
    // não imprime bye, imprime false

    int i = 10;
    System.out.println(i == 2 & i++ == 0);
    // imprime false, soma mesmo assim
    System.out.println(i);
    // imprime 11

    int j = 10;
    System.out.println(j == 2 && j++ == 0);
    // imprime false, não soma
    System.out.println(j);
    // imprime 10
}
```

Operadores de incremento e decremento

Para facilitar a codificação, ainda podemos ter operadores que fazem cálculos (aritméticos) e atribuição em uma única operação. Para somar ou subtrair um valor em 1, podemos usar os operadores de incremento/decremento:

```
int i = 5;

// 5 - pós-incremento, i agora vale 6
System.out.println(i++);

// 6 - pós-decremento, i agora vale 5
System.out.println(i--);

// 5
System.out.println(i);
```

E incrementos e decrementos antecipados:

```
int i = 5;

System.out.println(++i);    // 6 - pré-incremento
System.out.println(--i);    // 5 - pré-decremento
System.out.println(i);        // 5
```

Cuidado com os incrementos e decrementos em relação a **pré** e **pós**. Quando usamos pós-incremento, essa é a última coisa a ser executada. E quando usamos o pré-incremento, é sempre a primeira.

```
int i = 10;

// 10, primeiro imprime, depois incrementa
System.out.println(i++);

// 11, valor já incrementado.
System.out.println(i);

// 12, incrementa primeiro, depois imprime
System.out.println(++i);
```

```
// 12, valor incrementado.
System.out.println(i);
```

Existem ainda operadores para realizar operações e atribuições de uma só vez:

```
int a = 10;

// para somar 2 em a
a = a + 2;

//podemos obter o mesmo resultado com:
a += 2;

//exemplos de operadores:
int i = 5;

i += 10; //soma e atribui
System.out.println(i);        // 15

i -= 10; //subtrai e atribui
System.out.println(i);        // 5

i *= 3; // multiplica e atribui
System.out.println(i);        // 15

i /= 3; // divide a atribui
System.out.println(i);        // 5

i %= 2; // divide por 2, e atribui o resto
System.out.println(i);        // 1

System.out.println(i+=3); // soma 3 e retorna o resultado: 4
```

Nesses casos, o compilador ainda dá um desconto para operações com tipos teoricamente incompatíveis. Veja:

```
byte b1 = 3; // compila, dá um desconto
b1 = b1 + 4; // não compila, conta com int devolve int
```

```
byte b2 = 3; // compila, dá um desconto
b2 += 4; // compila, dá um desconto
System.out.println(b2); // imprime 7

b2 += 4003245; // compila também, mas estoura o byte
System.out.println(b2); // imprime -76
```

Esse último caso compila inclusive se passar valores absurdamente altos: `b2+=400` é diferente de `b2 = b2 + 400`. Ele faz o *casting* e roda normalmente.

Cuidado também com o caso de atribuição com o próprio autoincremento:

```
int a = 10;
a += ++a + a + ++a;
```

Como a execução é do primeiro para o último elemento das somas, temos as reduções:

```
a += ++a + a + ++a;
a = a + ++a + a + ++a;
a = 10 +11 + a + ++a;
a = 10 + 11 + 11 + ++a;
a = 10 + 11 + 11 + 12;
a = 44; // a passa a valer 44
```

Um outro exemplo de operador pós-incremento, cujo resultado é 1 e 2:

```
int j = 0;
int i = (j++ * j + j++);
System.out.println(i);
System.out.println(j);
```

Isso porque:

```
i = (0 * j + j++); // j vale 1
i = (0 * 1 + j++); // j vale 1
i = (0 * 1 + 1); // j vale 2
i = 1; // j vale 2
```

Podemos fazer diversas atribuições em sequência, que serão executadas da direita para a esquerda. O resultado de uma atribuição é sempre o valor da atribuição:

```
int a = 15, b = 20, c = 30;
a = b = c; // b = 30, portanto a = 30
```

Outro exemplo mais complexo:

```
int a = 15, b = 20, c = 30;
a = (b = c + 5) + 5; // c = 30, portanto b = 35, portanto a = 40
```

Operador ternário - Condicional

Há também um operador para controle de fluxo do programa, como um `if`. É chamado de **operador ternário**. Se determinada condição acontecer, ele vai por um caminho; caso contrário, vai por outro.

A estrutura do operador ternário é a seguinte: `variável = teste_booleano ? valor_se_verdadeiro : valor_se_falso;`.

```
int i = 5;
System.out.println(i == 5 ? "match": "oops"); // match
System.out.println(i != 5 ? 1: 2);                    // 2

String message = i % 2 == 0 ? "even" : "odd"; // odd
```

O operador condicional sempre tem de retornar valores que podemos usar para atribuir, imprimir etc.

Operador de referência

Para acessar os atributos ou métodos de um objeto, precisamos aplicar o operador `.` (ponto) em uma referência. Você pode imaginar que esse operador navega na referência até chegar no

objeto.

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

// Utilizando o operador "." para acessar um
// objeto String e invocar um método.
int length = s.length();
```

Concatenação de Strings

Quando usamos Strings, podemos usar o `+` para denotar concatenação. É a única classe que aceita algum operador fora o ponto.

Em Java, não há sobrecarga de operadores como em outras linguagens. Portanto, não podemos escrever nossas próprias classes com operadores diversos.

> **StringBuilder**
>
> A concatenação de Strings é um *syntax sugar* que o próprio compilador resolve. No código compilado, na verdade, é usado um `StringBuilder`.

Precedência

Não é necessário decorar a precedência de todos operadores do Java, basta saber o básico, que primeiro são executados pré-incrementos/ decrementos, depois multiplicação/ divisão/ mod, passando para soma/ subtração, depois os *shifts* (`<<` , `>>` , `>>>`) e, por último, os pós-incrementos/ decrementos.

As questões da certificação não entram em mais detalhes que isto.

Pontos importantes

- Na atribuição de um valor para uma variável primitiva, o valor deve ser do mesmo tipo da variável ou de um menos abrangente.

> **Exceção à regra**
>
> Para os tipos `byte`, `short` e `char`, em atribuições com literais do tipo `int`, o compilador verifica se o valor a ser atribuído está no range do tipo da variável.

- Toda variável não primitiva está preparada somente para armazenar referências para objetos que sejam do mesmo tipo dela.
- Toda comparação e toda operação lógica devolvem `boolean`.
- O resultado de toda operação aritmética é no mínimo `int`, ou do tipo da variável mais abrangente que participou da operação.
- A comparação de valores numéricos não considera os tipos dos valores.
- As referências e os valores `boolean` só podem ser

comparados com `==` ou `!=`.

- Toda atribuição é por cópia de valor.

> **Observação:** o recurso do *autoboxing* permite fazer algumas operações diferentes envolvendo variáveis não primitivas. Discutiremos sobre autoboxing adiante.

Casting de tipos primitivos

Não podemos atribuir a uma variável de um tipo um valor que não é compatível com ela:

```
double d = 3.14;
int i = d;
```

Só podemos fazer essas atribuições se os valores forem *compatíveis*. Compatível é quando um tipo cabe em outros, e ele só cabe se o *range* (alcance) dele for mais amplo que o do outro.

- **byte -> short -> int -> long -> float -> double**
- **char -> int**

Se estivermos convertendo de um tipo que vai da esquerda para a direita nessa tabelinha, não precisamos de casting. A **autopromoção** fará o serviço por nós. Já se estamos indo da direita para a esquerda, precisamos do *casting* e não importam os valores que estão dentro. Veja um exemplo:

```
double d = 0;
float  f = d;
```

Esse código não compila sem um casting! O casting é a maneira que usamos para moldar uma variável de um tipo em outro. Nós estamos avisando o compilador que sabemos da possibilidade de perda de precisão ou truncamento, mas nós realmente queremos fazer isso:

```
float  f = (float) d;
```

Podemos fazer casting entre ponto flutuante e inteiro. O resultado será o número truncado, sem as casas decimais:

```
double d = 3.1415;
int i = (int) d; // 3
```

Dica

Não é preciso decorar a sequência **int->long->float** etc. Basta lembrar dos alcances das variáveis. Por exemplo, o `char` tem dois bytes e guarda um número positivo. Será então que posso atribuir um `char` a um `short`? Não, pois um `short` tem 2 bytes, e usa meio a meio entre os números positivos e negativos.

Exercícios

1) Qual código a seguir compila?

a)

```
short s = 10;
char c = s;
```

b)

```
char c = 10;
long l = c;
```

c)

```
char c = 10;
short s = c;
```

2) Faça contas com diferentes operandos:

```
int i1 = 3/2;
double i2 = 3/2;
double i3 = 3/2.0;

long x = 0; double d = 0;
double zero = x + d;
System.out.println(i1 + i2 + i3 + x + d + zero);
```

Qual o resultado?

a) 3

b) 3.5

c) 4

d) 4.5

3) O código a seguir pode lançar um `NullPointerException`. Como evitar isso mantendo a mesma lógica?

```
void method(Car c) {
    if(c != null & c.getPreco() > 100000) {
        System.out.println("expensive");
    }
}
```

a) Trocando `!=` por `==`.

b) Trocando `>` por `<`.

c) Trocando `&` por `|`.

d) Trocando `&` por `&&`.

4) Alguns testes interessantes com tipos primitivos:

```
int i = (byte) 5;
long l = 3.0;
float f = 0.0;
char c = 3;
char c2 = -2;
```

Quais compilam?

a) `i`, `f` e `c`

b) `i`, `f`, `c` e `c2`

c) `i`, `f` e `c2`

d) `i` e `c`

e) `f` e `c`

f) `f` e `c2`

g) `i` e `l`

h) `l`, `f` e `c`

i) `i`, `c` e `c2`

5) A expressão a seguir pode ser reduzida, como podemos fazer?

```
if ((train && !car) || (!train && car)) {
    // ....
}
```

a) Trocando para usar um operador `&` e um `|`.

b) Trocando para usar dois operadores `&` e um `|`.

c) Trocando para usar um operador `!` e um `^`.

d) Trocando para usar um operador `^`.

e) Removendo os parênteses.

f) Removendo o `||` do meio.

g) Removendo os `!`.

6) Imprima a divisão por 0 de números inteiros e de números com ponto flutuante:

```
System.out.println(3 / 0);
System.out.println(3 / 0.0);
System.out.println(3.0 / 0);
System.out.println(-3.0 / 0);
```

Quais os resultados?

7) Qual o resultado desse código?

```
class Xyz {
  public static void main(String[] args) {
     int y;
     for(int x = 0; x<10; ++x) {
       y = x % 5 + 2;
     }
     System.out.println(y);
  }
}
```

a) Erro de compilação na linha 3

b) Erro de compilação na linha 7

c) 1

d) 2

e) 3

f) 4

g) 5

h) 6

8)

```
class Test {
    public static void main(String[] args){
        byte b = 1;
        int i = 1;
        long l = 1;
        float f = 1.0;
    }
}
```

O código:

a) Não compila a linha 3, pois `1` é `int` e não pode ser colocado em um `byte`.

b) Não compila a linha 4, pois `1` é `long` e não pode ser colocado em um `int`.

c) Não compila a linha 5, pois `1` é `int` e não pode ser colocado em um `long`.

d) Não compila a linha 6, pois `1.0` é `double` e não pode ser colocado em um `float`.

e) Todas as linhas compilam.

9)

```
class $_o0o_$ {
    public static void main(String[] args) {
        int $$ = 5;
        int __ = $$++;
        if (__ < ++$$ || __-- > $$)
            System.out.print("A");

        System.out.print($$);
        System.out.print(__);
    }
}
```

O estranho código:

a) Não compila por causa do nome da classe.

b) Não compila por causa dos nomes das variáveis.

c) Compila, mas dá erro na execução.

d) Compila, roda e imprime A76.

e) Compila, roda e imprime A75.

f) Compila, roda e imprime A74.

g) Compila, roda e imprime 76.

10) O que acontece com o seguinte código? Compila? Roda?

```
public class Test{
    public static void main(String[] args) {
        byte b1 = 5;
        byte b2 = 3;
        byte b3 = b1 + b2;
    }
}
```

11) O que acontece com seguinte código?

```
public class Test{
    public static void main(String[] args) {
        byte b1 = 127;
        byte b2 = -128;
        byte b3 = b1 + b2;
        System.out.println(b3);
    }
}
```

a) Não compila por um erro na linha 3.

b) Não compila por um erro na linha 4.

c) Não compila por um erro na linha 5.

d) Compila e imprime -1.

12) Qual é o resultado do código a seguir?

```
public class Test {
    public static void main(String[] args) {
        int i;
        for (i = 0; i < 5; i++) {
            if (++i % 3 == 0) {
                break;
            }
        }
        System.out.println(i);
    }
}
```

a) Imprime 1.

b) Imprime 2.

c) Imprime 3.

d) Imprime 4.

13) Considerando o mesmo código da questão anterior, e se trocarmos para pós-incremento dentro do `if` ?

```
public class Test {
    public static void main(String[] args) {
        int i;
        for (i = 0; i < 5; i++) {
            if (i++ % 3 == 0) {
                break;
            }
        }
        System.out.println(i);
    }
}
```

Qual é o resultado?

a) Imprime 1.

b) Imprime 2.

c) Imprime 3.

d) Imprime 4.

14) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        byte b1 = 100;
        byte b2 = 131;
        System.out.println(b1);
    }
}
```

a) Compila e imprime um número positivo.

b) Compila e imprime um número negativo.

c) Compila e dá uma exception de estouro de número.

d) Compila e imprime um número que não sabemos dizer ao certo.

e) Compila e imprime `"Not A Number"`.

f) Não compila.

15) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        char c = 65;
        char c2 = -3;
        System.out.println(c + c2);
    }
}
```

a) Não compila nas duas declarações de `char`.

b) Não compila nas três linhas dentro do método `main`.

c) Não compila somente na declaração de `c2`.

d) Não compila somente na soma de caracteres.

e) Compila e roda, imprimindo 62.

f) Compila e roda, imprimindo um outro valor.

16) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        char c = 65;
        char c2 = 68 - 65;
        System.out.println(c + c2);
    }
}
```

a) Não compila nas duas declarações de `char`.

b) Não compila nas três linhas dentro do método `main`.

c) Não compila somente na declaração de `c2`.

d) Não compila somente na soma de caracteres.

e) Compila e roda, imprimindo 62.

f) Compila e roda, imprimindo um outro valor.

17) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        double result = 15 / 0;
        System.out.println(result);
    }
}
```

a) Não compila.

b) Compila e dá exception.

c) Compila e imprime positivo infinito.

d) Compila e imprime 0.

18) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String result = "results: " + 15 / 0.0;
        System.out.println(result);
    }
}
```

a) Não compila.

b) Compila e dá exception.

c) Compila e imprime positivo infinito.

d) Compila e imprime 0.

19) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        System.out.println(1==true);
    }
}
```

a) Não compila.

b) Compila e imprime verdadeiro.

c) Compila e imprime falso.

3.2 USE PARÊNTESES PARA SOBRESCREVER A PRECEDÊNCIA DE OPERADORES

Às vezes, desejamos alterar a ordem de precedência de uma linha, e nesses instantes usamos os parênteses:

```
int a = 15 * 4 + 1; // 15 * 4 = 60, 60 + 1 = 61
int b = 15 * (4 + 1); // 4 + 1 = 5, 15 * 5 = 75
```

Devemos tomar muito cuidado na concatenação de `String` e precedência:

```
System.out.println(15 + 0 + " != 150");
// 15 != 150
System.out.println(15 + (0 + " == 150"));
// 150 == 150
```

```
System.out.println(("guilherme" + " silveira").length());
// 18

System.out.println("guilherme" + " silveira".length());
// guilherme9
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String result = ("division: " + 15) / 0.0;
        System.out.println(result);
    }
}
```

a) Não compila.

b) Compila e dá exception.

c) Compila e imprime positivo infinito.

d) Compila e imprime 0.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        System.out.println(((!(true==false))==true ? 1 : 0)==0);
    }
}
```

a) Imprime `true`.

b) Imprime `false`.

c) Não compila.

d) Imprime 1.

e) Imprime 0.

3.3 TESTE A IGUALDADE ENTRE STRINGS E OUTROS OBJETOS

Observe o seguinte código que cria duas Strings:

```
String name1 = new String("Mario");
String name2 = new String("Mario");
```

Como já estudamos anteriormente, o operador `==` é usado para comparação. Neste caso, como se tratam de objetos, vai comparar as duas referências e ver se apontam para o mesmo objeto:

```
String name1 = new String("Mario");
String name2 = new String("Mario");

System.out.println(name1 == name2); // false
```

Até aqui tudo bem. Mas vamos alterar um pouco nosso código, mudando a maneira de criar nossas Strings, e rodar novamente:

```
String name1 = "Mario";
String name2 = "Mario";

System.out.println(name1 == name2); // o que imprime?
```

Ao executar o código, vemos que ele imprime `true`. O que aconteceu?

Pool de Strings

O Java mantém um *pool* de objetos do tipo `String`. Antes de

criar uma nova String, primeiro o Java verifica neste pool se uma String com o mesmo conteúdo já existe; caso sim, ele a reutiliza, evitando criar dois objetos exatamente iguais na memória. Como as duas referências estão apontando para o mesmo objeto do pool, o `==` retorna `true`.

Mas por que isso não aconteceu antes, com nosso primeiro exemplo? O Java só coloca no pool as Strings criadas usando **literais**. Strings criadas com o operador `new` não são colocadas no pool automaticamente.

```
String name1 = "Mario"; //será colocada no pool
String name2 = new String("Mario");
/*
"Mario" é colocado, mas name2 é outra
referência, não colocada no pool
*/
```

Sabendo disso, temos de ter cuidado redobrado quando comparando Strings usando o operador `==`:

```
String s1 = "string";
String s2 = "string";
String s3 = new String("string");

System.out.println(s1 == s2); // true, mesma referencia
System.out.println(s1 == s3); // false, referências diferentes
System.out.println(s1.equals(s3)); // true, mesmo conteúdo
```

Repare que, mesmo sendo instâncias diferentes, quando comparadas usando o método `equals`, o retorno é `true`, caso o conteúdo das Strings seja o mesmo. Quando concatenamos literais, a String resultante também será colocada no pool.

```
String ab = "a" + "b";
System.out.println("ab" == ab); // true
```

Mas isso é verdade **apenas usando literais em ambos os lados**

da concatenação. Se algum dos objetos não for um literal, o resultado será um novo objeto, que não estará no pool:

```
String a = "a";
String ab = a + "b"; //usando uma referência e um literal
System.out.println("ab" == ab); // false
```

Sabemos que Strings são imutáveis, e que cada método chamado em uma String retorna uma nova String, sem alterar o conteúdo do objeto original. Esses objetos resultantes de retornos de métodos não são buscados no pool, são novos objetos:

```
String str = "12 text 345678";
String txt1 = "text";
String txt2 = str.substring(3, 7); // new string
System.out.println(txt1 == txt2); // false
System.out.println(txt1.equals(x.substring(3, 7)); // true
```

OS MÉTODOS DE STRING SEMPRE CRIAM NOVOS OBJETOS?

Nem sempre. Se o retorno do método for exatamente o conteúdo atual do objeto, nenhum objeto novo é criado:

```
String str = "HELLO WORLD";
String upper = str.toUpperCase();    // já está maiúscula
String subs = str.substring(0,11);   // string completa
System.out.println(str == upper);          // true
System.out.println(str == subs);           // true
System.out.println(str == str.toString()); // true
```

Contando Strings

Uma questão recorrente na prova é contar quantos objetos do tipo `String` são criados em um certo trecho de código. Veja o código a seguir e tente descobrir quantos objetos String são

criados:

```
String h = new String ("hello ");
String h1 = "hello ";
String w = "world";

System.out.println("hello ");
System.out.println(h1 + "world");
System.out.println("Hello " == h1);
```

E então? Vamos ver passo a passo:

```
//Cria 2 objetos, um literal (que vai para o pool) e o outro
//com o new
String h = new String ("hello ");

//nenhum objeto criado, usa o mesmo do pool
String h1 = "hello ";
//novo objeto criado e inserido no pool
String w = "world";

//nenhum  objeto criado, usa do pool
System.out.println("hello ");

//criado um novo objeto resultante da concatenação,
// mas este não vai para o pool
System.out.println(h1 + "world");

//Novo objeto criado e colocado no pool (Hello com H maiúsculo).
System.out.println("Hello " == h1);
```

Logo temos 5 Strings criadas.

Cuidado com String já colocadas no pool

Para descobrir se uma String foi criada e colocada no pool, é necessário prestar muita atenção ao contexto do código e ao enunciado da questão. A String só é colocada no pool na primeira execução do trecho de código. Cuidado com questões que criam Strings dentro de métodos, ou que dizem em seu enunciado que o método já foi executado pelo menos uma vez:

```
public class Tests {
    public static void main(String[] args) {
        for(int i = 0; i< 10; i++)
            System.out.println(method());
    }

    private static String method() {
        String x = "x"; // A
        return x.toString();
    }
}
```

Ao executar essa classe, apenas **um** objeto String será criado. O único lugar onde a String é criada é na linha A.

O método equals

Para comparar duas referências, podemos sempre usar o operador `==` . Dada a classe `Client` :

```
class Client {
    private String name;
    Client(String name) {
        this.name = name;
    }
```

```
}

Client c1 = new Client("guilherme");
Client c2 = new Client("mario");
System.out.println(c1==c2); // false
System.out.println(c1==c1); // true

Client c3 = new Client("guilherme");
System.out.println(c1==c3);
// false, pois não é a mesma
// referência: são objetos diferentes na memória
```

Para comparar os objetos de uma outra maneira, que não através da referência, podemos utilizar o método `equals`, cujo comportamento padrão é fazer a simples comparação com o `==`:

```
Client c1 = new Client("guilherme");
Client c2 = new Client("mario");
System.out.println(c1.equals(c2)); // false
System.out.println(c1.equals(c1)); // true

Client c3 = new Client("guilherme");
System.out.println(c1.equals(c3));
// false, pois não é a mesma
// referência: são objetos diferentes na memória
```

Isso é, existe um método em `Object` que você pode reescrever para definir um **critério de comparação de igualdade**. Classes como `String`, `Integer` e muitas outras possuem esse método reescrito, assim `new Integer(10) == new Integer(10)` dá `false`, mas `new Integer(10).equals(new Integer(10))` dá `true`.

É interessante reescrever esse método quando você julgar necessário um critério de igualdade diferente que o `==` retorna. Imagine o caso de nosso `Client`:

```
class Client {
    private String name;
    Client(String name) {
```

```
        this.name = name;
    }

    public boolean equals(Object o) {
        if (! (o instanceof Client)) {
            return false;
        }
        Client second = (Client) o;
        return this.name.equals(second.name);
    }
}
```

O método `equals` não consegue tirar proveito do `generics`, então precisamos receber `Object` e ainda verificar se o tipo do objeto passado como argumento é realmente uma `Client` (o contrato do método diz que você deve retornar `false`, e não deixar lançar exception em um caso desses). Agora sim, podemos usar o método `equals` como esperamos:

```
Client c1 = new Client("guilherme");
Client c2 = new Client("mario");
System.out.println(c1.equals(c2)); // false
System.out.println(c1.equals(c1)); // true

Client c3 = new Client("guilherme");
System.out.println(c1.equals(c3)); // true
```

Cuidado ao sobrescrever o método `equals`: ele deve ser público, e deve receber `Object`. Caso você receba uma referência a um objeto do tipo `Client`, seu método não está sobrescrevendo aquele método padrão da classe `Object`, mas sim criando um novo método (overload). Por polimorfismo, o compilador fará funcionar neste caso, já que pois o compilador fará a conexão ao método mais específico, entre `Object` e `Client`, ele escolherá o método que recebe `Client`:

```
class Client {
    private String name;
    Client(String name) {
```

```
        this.name = name;
    }

    public boolean equals(Client second) {
        return this.name.equals(second.name);
    }
}

Client c1 = new Client("guilherme");
Client c2 = new Client("mario");
System.out.println(c1.equals(c2)); // false
System.out.println(c1.equals(c1)); // true

Client c3 = new Client("guilherme");
System.out.println(c1.equals(c3)); // true
System.out.println(c1.equals((Object) c3));
// false, o compilador não sabe que Object é cliente,
// invoca o equals tradicional, e azar do desenvolvedor
```

Mas caso você use alguma biblioteca (como a API de coleções e de `ArrayList` do Java), o resultado não será o esperado.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String s1 = "s1";
        String s2 = "s" + "1";
        System.out.println(s1==s2);
        System.out.println(s1==("" + s2));
    }
}
```

a) Não compila.

b) Compila e imprime `true`, `false`.

c) Compila e imprime `true`, `true`.

d) Compila e imprime `false` , `false` .

e) Compila e imprime `false` , `true` .

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String s1 = "s1";
        String s2 = s1.substring(0, 1) + s1.substring(1,1);
        System.out.println(s1==s2);
        System.out.println(s1.equals(s2));
    }
}
```

a) Não compila.

b) Compila e imprime `true` , `false` .

c) Compila e imprime `true` , `true` .

d) Compila e imprime `false` , `false` .

e) Compila e imprime `false` , `true` .

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
            String s1 = "s1";
            String s2 = s1.substring(0, 2);
            System.out.println(s1==s2);
            System.out.println(s1.equals(s2));
    }
}
```

a) Não compila.

b) Compila e imprime `true` , `false` .

c) Compila e imprime `true` , `true` .

d) Compila e imprime `false` , `false` .

e) Compila e imprime `false` , `true` .

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B extends C{}
class C {
    int x;
    public boolean equals(C c) {
        return c.x==x;
    }
}
class A {
    public static void main(String[] args) {
        C a = new C();
        C b = new B();
        a.x = 1;
        b.x = 1;
        System.out.println(a==b);
        System.out.println(a.equals(b));
    }
}
```

a) Não compila.

b) Compila e imprime `true` , `false` .

c) Compila e imprime `true` , `true` .

d) Compila e imprime `false` , `false` .

e) Compila e imprime `false` , `true` .

5) Escolha a opção adequada ao tentar compilar e rodar o

arquivo a seguir:

```
class B extends C{}
class D {
    int x;
}
class C {
    int x;
    public boolean equals(Object c) {
        return c.x==x;
    }
}
class A {
    public static void main(String[] args) {
        C a = new C();
        C b = new D();
        a.x = 1;
        b.x = 1;
        System.out.println(a==b);
        System.out.println(a.equals(b));
    }
}
```

a) Não compila.

b) Compila e imprime `true` , `false` .

c) Compila e imprime `true` , `true` .

d) Compila e imprime `false` , `false` .

e) Compila e imprime `false` , `true` .

3.4 UTILIZE O IF E IF/ELSE

Imagine um programa que aceita comandos do usuário, ou seja, um sistema interativo. De acordo com os dados que o usuário passar, o programa se comporta de maneiras diferentes e, consequentemente, pode dar respostas diferentes.

O programador, ao escrever esse programa, deve ter recursos para definir o comportamento para cada possível comando do usuário, em outras palavras, para cada situação. Com isso, o programa será capaz de tomar decisões durante a execução, com o intuito de mudar o fluxo de execução.

As linguagens de programação devem oferecer aos programadores maneiras para *controlar o fluxo de execução dos programas*. Dessa forma, os programas podem tomar decisões que afetam a sequência de comandos que serão executados.

if / else

A maneira mais simples de controlar o fluxo de execução é definir que um determinado trecho de código deve ser executado quando uma condição for verdadeira.

Por exemplo, suponha um sistema de login. Ele deve verificar a autenticidade do usuário para permitir ou não o acesso. Isso pode ser implementado com um `if/else` do Java.

```
boolean authenticated = true;
if (authenticated) {
    System.out.println("Valid");
} else {
    System.out.println("Invalid");
}
```

A sintaxe do `if` é a seguinte, na qual falamos a condição (`condition`) e o que será executado:

```
if (CONDITION) {
    // CODIGO 1
} else {
    // CODIGO 2
}
```

A condição de um `if` **sempre** tem de ser um valor booleano:

```
if(1 - 2) { } // erro, numero inteiro

if(1 < 2) {} //ok, resulta em true

boolean value = true;
if (value == false) {} // ok, mas resulta em false

if (value) {} // ok, value é boolean
```

Atenção dobrada ao código a seguir:

```
int a = 0, b = 1;

if(a = b) {
    System.out.println("same");
}
```

Esta é uma pegadinha bem comum. Repare que não estamos fazendo uma **comparação** aqui, e sim, uma **atribuição** (um único `=`). O resultado de uma atribuição é sempre o valor atribuído, no caso, um inteiro. Logo, este código não compila, pois passamos um inteiro para a condição do `if`.

A única situação em que um código assim poderia funcionar é caso a variável atribuída seja do tipo `boolean`, pois o resultado da atribuição será `boolean`:

```
boolean a = true;

if(a = false) {
    System.out.println("false!");
}
```

Neste caso, o código compila, mas não imprime nada. Após a atribuição, o valor da variável `a` é `false`, e o `if` não é executado.

Caso só tenhamos um comando dentro do `if` ou `else`, as chaves são opcionais:

```
if(!resultado)
    System.out.println("false!");
else
    System.out.println("true!");
```

Caso não tenhamos nada para ser executado em caso de condição `false`, não precisamos declarar o `else`:

```
boolean authenticated = true;
if (authenticated)
    System.out.println("Accepted");
```

Mas sempre temos de ter algum código dentro do `if`; se não, o código não compila:

```
boolean authenticated = true;
if (authenticated)
else // erro
    System.out.println("Declined");
```

Na linguagem Java, **não** existe o comando `elseif`. Para conseguir esse efeito, os `if`s são colocados dentro dos `else`.

```
if (CONDITION1) {
    // CODIGO 1
} else if (CONDITION2) {
    // CODIGO 2
} else {
    // CODIGO 3
}
```

Grande parte das perguntas sobre estruturas de `if/else` são pegadinhas, usando a indentação como forma de distração:

```
boolean authenticated = true;
if (authenticated)
    System.out.println("Accepted");
else
```

```
    System.out.println("Declined");
    System.out.println("Try again");
```

A mensagem `"Tente novamente"` sempre é impressa, independente do valor da variável `autentico` .

Esse foi um exemplo bem simples, vamos tentar algo mais complicado. Tente determinar o que é impresso:

```
int valor = 100;
if (valor > 200)
if (valor <400)
if (valor > 300)
    System.out.println("a");
else
    System.out.println("b");
else
    System.out.println("c");
```

E então? `"c"` ? Vamos reindentar o código para ver se fica mais fácil:

```
int valor = 100;
if (valor > 200)
    if (valor <400)
        if (valor > 300)
            System.out.println("a");
        else
            System.out.println("b");
    else
        System.out.println("c");
```

É sempre complicado analisar código não indentado ou mal indentado, e esse recurso é usado extensivamente em várias questões durante a prova. Fique esperto!

Unreachable Code e Missing return

Um código Java não compila se o compilador perceber que

aquele código não será executado sob hipótese alguma:

```
class Test {
    public int method() {
        return 5;
        System.out.println("Will it run?");
    }
}
```

```
Test.java:10: unreachable statement
        System.out.println("Will it run?");
                ^
```

O código após o `return` não será nunca executado. Esse código não compila. Vamos ver alguns outros exemplos:

```
class Test {
    public int method(int x) {
        if(x > 200) {
            return 5;
        }
    }
}
```

Este também não compila. O que será retornado se `x` for `<= 200`?

```
Test.java:12: missing return statement
    }
        ^
1 error
```

Vamos modificar o código para que ele compile:

```
class Test {
    public int method(int x) {
        if(x > 200) {
            return 5;
        }
        throw new RuntimeException();
    }
}
```

Apesar de não estarmos retornando nada caso o `if` seja falso, o Java percebe que nesse caso uma exceção será disparada. A regra é: todos os caminhos possíveis devem retornar o tipo indicado pelo método, ou lançar exceção.

Em um `if`, essa expressão compila normalmente:

```
if(false) {.... } //compila, apesar de ser unreachable code
```

São pequenos detalhes, tome cuidado para não cair nessas pegadinhas.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        if(args.length > 0)
            System.out.println("1 or more");
        else
            System.out.println("0");
    }
}
```

a) Não compila: `length` é método.

b) Não compila: faltou chaves no `if e else`.

c) Se invocarmos sem argumentos, imprime `0`.

d) Nunca imprimirá `0`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
```

```
    final boolean valor = false;
}
class A {
    public static void main(String[] args) {
        B b = new B();
        if(b.valor = true) {
            System.out.println("uhu true");
        }
    }
}
```

a) Não compila.

b) Compila e imprime `uhu true`.

c) Compila e não imprime nada.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int qt = 15;
        if(qt=15) {
            System.out.println("yes");
        } else {
            System.out.println("no");
        }
    }
}
```

a) Não compila.

b) Imprime `sim`.

c) Imprime `não`.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
```

```
    public static void main(String[] args) {
        if(args.length==1)
            System.out.println("one");
        elseif(args.length==2)
            System.out.println("two");
        elseif(args.length==3)
            System.out.println("three");
        else
            System.out.println("four");
    }
}
```

a) Não compila.

b) Roda e imprime `"one"` quando passamos um argumento.

c) Roda e imprime `"three"` quando passamos 4 argumentos.

d) Roda e não imprime nada quando passamos nenhum argumento.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String name = args[0];
        if(name.equals("guilherme"))
            System.out.println(name);
            System.out.println("good");
        else
            System.out.println("better");
            System.out.println(name);
    }
}
```

a) Erro de compilação no `if`.

b) Erro de compilação no `else`.

c) Compila e imprime o nome e `"good"`, caso o primeiro

argumento seja `guilherme`.

d) Compila e dá erro de execução caso não passe nenhum argumento na linha de comando.

3.5 UTILIZE O SWITCH

Suponha que um programa tenha de reagir diferentemente para três casos possíveis. Por exemplo, suponha que o usuário possa passar três valores possíveis: `1`, `2` e `3`. Se for `1`, o programa deve imprimir `"PRIMEIRA OPCAO"`; se for `2`, `"SEGUNDA OPCAO"`; e se for `3`, `"TERCEIRA OPCAO"`.

Isso pode ser implementado com `if/else`. Mas, há uma outra possibilidade. O Java, assim como outras linguagens de programação, oferece o comando `switch`. Ele permite testar vários casos de uma maneira diferente do `if/else`.

```
int option = 1;
switch (option) {
    case 1:
        System.out.println("number 1");
    case 2:
        System.out.println("number 2");
    case 3:
        System.out.println("number 3");
}
```

O `switch` tem uma sintaxe cheia de detalhes e uma semântica pouco intuitiva. Vamos analisar cada um desses detalhes separadamente para ficar mais simples.

O argumento do `switch` dever ser uma variável compatível com o tipo primitivo `int`, um *wrapper* de um tipo menor que `Integer`, uma **String** ou um **enum**. Enums **não** são cobrados

nessa prova, então vamos focar apenas nos outros dois casos (números e Strings).

O valor de cada `case` deve ser compatível com o tipo do argumento do `switch`; caso contrário, será gerado um erro de compilação na linha do `case` inválido.

```
//argumento do switch int, e cases int
int value = 20;
switch (value){
    case 10 : System.out.println(10);
    case 20 : System.out.println(20);
}

//Argumento String, e cases String
String s = "Oi";
switch (s) {
    case "Oi": System.out.println("Olá");
    case "Hi": System.out.println("Hello");
}

//Argumento Byte, e cases byte
Byte b = 10;
switch (b) {
    case 10: System.out.println("TEN");
}

//argumento do switch int, e cases string, não compila
int mix = 20;
switch (mix){
    case "10" : System.out.println(10);
    case "20" : System.out.println(20);
}
```

Cuidado, pois `switch` de `double` não faz sentido conforme a lista de argumentos que citamos compatíveis com o `switch`!

```
double mix = 20;
switch (mix){  // compile error
    case 10.0 : System.out.println(10);
    case 20.0 : System.out.println(20);
}
```

Você pode usar qualquer tipo primitivo menor que um `int` como argumento do `switch`, desde que os tipos dos cases sejam compatíveis:

```
//argumento do switch byte
byte value = 20;

switch (value){
    // Apesar de ser inteiro, 10 cabe em um byte, o compilador
    // fará o cast automaticamente
    case 10 :
        System.out.println(10);
}
```

O exemplo a seguir mostra o caso de incompatibilidade. Uma vez que o número é muito grande, o compilador não fará o cast e teremos um erro de compilação por tipos incompatíveis:

```
byte value = 20;

switch (value){
    case 32768 :  // compile error
        System.out.println(10);
}
```

Em cada `case`, só podemos usar como valor um literal, uma variável `final` atribuída com valor literal, ou expressões envolvendo os dois. Nem mesmo `null` é permitido:

```
int value = 20;
final int FIVE = 5;
int thirty = 30;

switch (value) {
    case FIVE: // constante
        System.out.println(5);
    case 10: // literal
        System.out.println(10);
    case FIVE * 4: // operação com constante e literal
        System.out.println(20);
    case thirty: // compile error, variável
```

```
        System.out.println(30);
    case thirty + FIVE: // compile error, operação envolvendo
                        // variável
        System.out.println(35);
    case null: // compile error, explicit null
        System.out.println("null");
}
```

Constantes em cases

Para ser considerada uma constante em um `case`, a variável, além de ser final, também deve ter sido inicializada durante a sua declaração. Inicializar a variável em outra linha faz com que ela não possa ser usada como valor em um `case`:

```
int v = 10;
final int TEN = 10;
final int TWENTY; // final, mas não inicializada
TWENTY = 20; // inicializada

switch (v) {
case TEN:
    System.out.println("10!");
    break;
case TWENTY: // compile error
    System.out.println("20!");
    break;
}
```

O `switch` também aceita a definição de um caso padrão, usando a palavra `default`. O caso padrão é aquele que deve ser executado se nenhum `case` "bater".

```
int option = 4;
switch (option) {
    case 1:
        System.out.println("1");
```

```
    case 2:
        System.out.println("2");
    case 3:
        System.out.println("3");
    default:
        System.out.println("DEFAULT");
}
```

Um detalhe sobre a sintaxe do `default` é que ele pode aparecer antes de um ou de diversos `case` s. Desta forma:

```
int option = 4;
switch(option) {
    case 1:
        System.out.println("1");
    case 2:
        System.out.println("2");
    default:
        System.out.println("DEFAULT");
    case 3:
        System.out.println("3");
}
```

Um comportamento contraintuitivo do `switch` é que, quando executado, se algum `case` "bater", tudo que vem abaixo é executado também, todos os `case` s e o `default` , se ele estiver abaixo. Esse comportamento também vale se cair no `default` . Por exemplo, o código anterior imprime:

```
DEFAULT
3
```

Com esse comportamento, podemos inclusive criar `cases` sem nenhum bloco de código dentro:

```
int v = 1;
switch(v){
    case 1:
    case 2:
    case 3:
        System.out.println("1,2,3 => Hi!");
```

```
}
```

Para mudar esse comportamento e não executar o que vem abaixo de um `case` que bater ou do `default`, é necessário usar o comando `break` em cada `case`.

```
int v = 4;
switch(v) {
    case 1:
        System.out.println("1");
        break;
    case 2:
        System.out.println("2");
        break;
    default:
        System.out.println("DEFAULT");
        break;
    case 3:
        System.out.println("3");
        break;
}
```

Neste caso, só será impresso `"DEFAULT"`.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int t = args.length;
        switch(t) {
            case 1:
                System.out.println("1");
            case 2:
                System.out.println("2");
            default:
                System.out.println("+++");
        }
    }
```

```
}
```

a) Não compila.

b) Ao rodar sem argumentos joga uma exception.

c) Ao rodar com dois argumentos, imprime somente `"2"`.

d) Ao rodar com 5 argumentos, imprime `"+++"`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int t2 = 1;
        int t = args.length;
        switch(t) {
            case t2:
                System.out.println("1");
                break;
            default:
                System.out.println("arg???");
        }
    }
}
```

a) Não compila.

b) Ao rodar sem argumentos joga uma exception.

c) Ao rodar com um argumento, imprime somente `"1"`.

d) Ao rodar com 5 argumentos, imprime `"arg???"`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        switch("Guilherme") {
```

```
            case "Guilherme":
                System.out.println("Guilherme");
                break;
            case "42":
                System.out.println("42");
            default:
                System.out.println("Mario");
        }
    }
}
```

a) Não compila, pois um número não pode ser comparado com `String`.

b) Compila e imprime `Guilherme`.

c) Não compila, pois o código do `case 42` e `default` nunca serão executados.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int count = args.length;
        switch(count) {
            case 0 {
                System.out.println("---");
                break;
            } case 1 {
            } case 2 {
                System.out.println("ok");
            } default {
                System.out.println("default");
            }
        }
    }
}
```

a) Erro de compilação.

b) Se rodar com 1 argumento, imprime `ok` e mais uma mensagem.

c) Se rodar com 1 argumento, não imprime nada.

d) Se rodar com 5 argumentos, imprime `default`.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        switch(10) {
            case < 10:
                System.out.println("<");
            default:
                System.out.println("=");
            case > 10:
                System.out.println(">");
        }
    }
}
```

a) Erro de compilação.

b) Compila e imprime `"="`.

c) Compila e imprime `"="` e `">"`.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        switch(10) {
            case 10:
                System.out.println("a");
                break;
                System.out.println("b");
            default:
                System.out.println("c");
```

```
            case 11:
                System.out.println("d");
        }
    }
}
```

a) Não compila.

b) Imprime `a e b e c e d`.

c) Imprime `a`.

CAPÍTULO 4

CRIANDO E USANDO ARRAYS

4.1 DECLARE, INSTANCIE, INICIALIZE E USE UM ARRAY UNIDIMENSIONAL

As linguagens de programação, normalmente, fornecem algum recurso para o armazenamento de variáveis em memória sequencial. No Java, os arrays permitem esse tipo de armazenamento.

Um array é um objeto que armazena sequencialmente "uma porção" de variáveis de um determinado tipo. É importante reforçar que os arrays são objetos. Uma referência para um objeto array deve ser armazenada em uma variável do tipo array.

A prova de certificação verifica se o candidato está apto a manipular tanto arrays de tipos primitivos quanto de tipos não primitivos. Os quatro pontos importantes sobre arrays são:

- Declarar
- Inicializar
- Acessar
- Percorrer

Arrays de tipos primitivos

Declaração

Para declarar um array, é utilizado `[]` logo após o tipo da variável:

```
// Declaração de um array para guardar variáveis do tipo int.
int[] age;

// Declaração de um array para guardar variáveis do tipo long.
long[]size;
```

Podemos declarar a array com o `[]` , logo após ao nome da variável:

```
// Declaração de um array para guardar variáveis do tipo double.
double weight[];

// Declaração de um array para guardar variáveis do tipo long.
long []size;
```

Inicialização

Como um array é um objeto, a inicialização envolve a criação de um objeto. O `new` , operador que cria objetos, é utilizado para construir um array.

Se você não executa o `new` , qual o valor padrão? Para atributos, é `null` , e para variáveis locais, não há valor, como qualquer outra variável de referência:

```
public class Clients {

    int[] ages;

    public static void main(String[] args) {
        Clients c = new Clients();
        System.out.println(c.ages); // null
    }
```

```
}
```

O caso a seguir mostra um erro de compilação. Uma vez que a variável é local, ela não tem valor padrão `null` e o compilador reclama que ela não foi inicializada.

```
public class Products {

    public static void main(String[] args) {
        int[] prices;
        System.out.println(prices); // compile error
    }
}
```

E como instancio um array?

```
int[] ages;
double lengths[];

ages = new int[10];
lengths = new double[50];
```

Na inicialização, é definida a capacidade do array, ou seja, a quantidade de variáveis que ele terá. Quando falarmos em tamanho de um array, estaremos nos referindo à sua capacidade.

Cada variável guardada em um array é iniciada implicitamente no momento em que o array é criado. Os valores atribuídos às variáveis são os valores `default`. O exemplo a seguir imprime `0`, pois esse é o valor `default` para `int`.

```
int[] ages;
ages = new int[10];
System.out.println(ages[0]);
```

E temos alguns casos extremos, como criar um array de tamanho zero, compila e roda:

```
    int[] numbers = new int[0];
```

Já criar uma array com tamanho negativo compila, mas joga uma `NegativeArraySizeException` durante sua execução.

```
int[] numbers;
numbers = new int[-1];
```

Durante a declaração de uma referência para um array, temos a oportunidade de criá-lo de uma maneira mais fácil se já sabemos o que queremos colocar dentro. Por ser a maneira mais simples de inicializar uma array de tamanho e valores conhecidos, esse código pode aparecer bastante. Não passamos o tamanho e fazemos a declaração dos elementos entre chaves e separados por vírgula. O array a seguir tem tamanho 5:

```
int[] numbers;
numbers = new int[]{1,2,5,7,5};
```

Em arrays de referências, podemos ter inclusive valores nulos. O array a seguir tem tamanho 3:

```
Car[] cars = new Car[]{new Car(), null, new Car()};
```

E se a declaração e a inicialização estiverem na **mesma linha**, podemos simplificar ainda mais:

```
int[] numbers = {1,2,5,7,5};
```

Mas temos de tomar um pouco de cuidado com esse modo mais simples de declarar o array. Só podemos fazer como no exemplo anterior, quando **declaramos e inicializamos** o array na mesma linha. Se fizermos a declaração e a inicialização em linhas separadas, o código não compila:

```
int[] numbers = {1,2,5,7,5}; // ok
int[] numbers2;
numbers2 = {1,2,5,7,5}; // compile error
```

Se desejamos inicializar posteriormente, devemos adicionar o operador `new` para poder iniciar o array em outra linha:

```
int[] numbers2;
numbers2 = new int[]{1,2,5,7,5}; // ok
```

Acesso

As posições de um array são indexadas (numeradas) de `0` até a capacidade do array menos um. Para acessar uma das variáveis do array, é necessário informar sua posição.

```
// Coloca o valor 10 na primeira variável do array ages.
int ages[] = new int[10];
ages[0] = 10;

// Coloca o valor 73.14 na última variável do array weights.
double weights[] = new double[50];
weights[49] = 73.14;
```

O que acontece se alguém tentar acessar uma posição que não existe? Durante a execução, teremos uma `ArrayIndexOutOfBoundsException`, como mostra o exemplo a seguir:

```
weights[50] = 88.4; // ArrayIndexOutOfBoundsException
```

A *exception* lançada pelo Java é `ArrayIndexOutOfBoundsException`. Cuidado para não confundir com a exception que a `String` joga ao tentar acessar uma posição inválida dela.

Percorrendo

Supondo que a capacidade de um array qualquer seja `100`, os índices desse array variam de `0` até `99`, ou seja, de `0` até a

capacidade menos um. O tamanho de um array é definido na inicialização e fica guardado no próprio array, podendo ser recuperado posteriormente.

Para recuperar o tamanho ou a capacidade de um array, é utilizado um atributo chamado `length` presente em todos os arrays.

```
int[] ages = {33, 30, 13};

for (int i = 0; i < ages.length; i++) {
    ages[i] = i;
}
```

No `for` tradicional, as posições de um array são acessadas através dos índices. Dessa forma, é possível, inclusive, modificar os valores que estão armazenados no array.

Porém, em determinadas situações, é necessário apenas ler os valores de um array sem precisar modificá-los. Nesse caso, pode ser usado o `for` introduzido na versão 5 do Java.

```
int[] ages = {33, 30, 13};

for(int age : ages){
    System.out.println(age);
}
```

Não há índices no `for` do Java 5. Ele simplesmente percorre os valores, assim ele não permite modificar o array facilmente.

Array de referências

Em cada posição de um array de tipos não primitivos, é guardada uma variável não primitiva. Esse é um fato fundamental. O código a seguir declara e inicializa um array de `Exam`:

```
Exam[] exams = new Exam[10];
```

Lembrando de que o `new` inicia as variáveis implicitamente e que o valor padrão para variáveis não primitivas é `null`, todas as dez posições do array desse código estão `null` imediatamente após o `new`.

```
Exam[] exams = new Exam[10];

// Erro de execução ao tentar aplicar o operador "."
// em  uma referência com valor null.
// NullPointerException
exams[0].timeLimit = 10;
```

Para percorrer um array de tipos não primitivos, podemos usar um laço:

```
Exam[] exams = new Exam[10];

for (int i = 0; i < exams.length; i++){
    exams[i] = new Exam();
    exams[i].timeLimit = 210;
}

for (Exam exam : exams){
    System.out.println(exam.timeLimit);
}
```

Caso a classe `Exam` seja abstrata, devido ao polimorfismo, é possível adicionar filhas de `Exam` nesse array: o polimorfismo funciona normalmente, portanto, funciona igualmente para interfaces.

```
abstract class Exam {
}
class PracticalExam extends Exam {
}

class TheoreticalExam extends Exam {
}
class Test {
```

```
    public static void main(String[] args) {
        Exam[] exams = new Exam[2];
        exams[0] = new TheoreticalExam();
        exams[1] = new PracticalExam();
    }
}
```

Uma vez que o array de objetos é sempre baseado em referências, lembre-se de que um objeto não será copiado, mas somente sua referência passada:

```
Client guilherme = new Client();
guilherme.setName("Guilherme");

Client[] clients = new Clients[10];
clients[0] = guilherme;

System.out.println(guilherme.getName()); // Guilherme
System.out.println(clients[0].getName()); // Guilherme

guilherme.setName("Silveira");

System.out.println(guilherme.getName()); // Silveira
System.out.println(clients[0].getName()); // Silveira
```

Casting de arrays

Não há *casting* de arrays de tipo primitivo, portanto, não adianta tentar:

```
int[] values = new int[10];
long[] vals = values; // compile error
```

Já no caso de referências, é possível fazer a atribuição sem casting de um array para outro tipo de array, por causa do polimorfismo.

```
String[] values = new String[2];
values[0] = "Certification";
values[1] = "Java";
```

```
Object[] vals = values;
for(Object val : vals) {
    System.out.println(val); // Certification, Java
}
```

E o casting compila normalmente. Mas, ao executarmos, um array de `Object` não é um array de String e levamos uma `ClassCastException`:

```
Object[] values = new Object[2];
values[0] = "Certification";
values[1] = "Java";

String[] vals = (String[]) values;
for(Object val : vals) {
    System.out.println(val);
}
```

Isso ocorre pois a classe dos dois é distinta e a classe pai de `array de string` não é um array de objeto, e sim, um `Object` (lembre-se: todo array herda de `Object`):

```
Object[ ] objects = new Object[ 2 ];
String[ ] strings = new String[ 2 ];
System.out.println(objects.getClass().getName());
// [ Ljava.lang.Object;
System.out.println(strings.getClass().getName());
// [ Ljava.lang.String;

System.out.println(strings.getClass().getSuperclass());
// java.lang.Object
```

Exercícios

1) Escolha a opção que não compila:

a) `int[] x;`

b) `int x[];`

c) `int[]x ;`

d) `int [] x ;`

e) `[]int x ;`

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int x[] = new int[30];        // A
        int y[] = new int[3] {0,3,5}; // B
    }
}
```

a) A linha **A** não compila.

b) A linha **B** não compila.

c) O código compila e roda.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, em relação às linhas dentro do método `main` :

```
class A {
    public static void main(String[] args) {
        int x[] = new int[0];
        int x[] = new int[] {0,3,5};
        int x[] = {0,3,5};
    }
}
```

a) A primeira e segunda linhas não compilam.

b) A segunda e terceira linhas não compilam.

c) Somente a terceira linha não compila.

d) O programa compila e roda, dando uma exception.

e) O programa compila e roda, imprimindo nada.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int x[] = new int[3];
        for(int i=x.length;i>=0;i--) x[i]=i*2;
        System.out.println("end!");
    }
}
```

a) O programa não compila

b) O programa imprime `end!`.

c) O programa compila e dá erro em execução.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int x[] = new int[3];
        for(x[1]=x.length-1;x[0]==0;x[1]--) {
            x[x[1]]=-5;
            System.out.println(x[1]);
        }
    }
}
```

a) Não compila.

b) Compila, imprime alguns números e dá uma `Exception`.

c) Compila e não imprime nada.

d) Compila e imprime `2`.

e) Compila e imprime `-5`.

f) Compila e imprime `2, -5`.

g) Compila e imprime `2, -5, -5`.

h) Compila e imprime `2, 1, -5`.

i) Compila e imprime `-5, -5`.

j) Dá exception.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int x[] = new int[3];
        for(x[1]=x.length-1;x[1]>=0;x[1]--) {
            x[x[1]]=-5;
            System.out.println(x[1]);
        }
    }
}
```

a) Não compila.

b) Compila, imprime alguns números e dá uma `Exception`.

c) Compila e não imprime nada.

d) Compila e imprime `2`.

e) Compila e imprime `-5`.

f) Compila e imprime `2, -5`.

g) Compila e imprime `2, -5, -5`.

h) Compila e imprime `2 , 1 , -5 .`

i) Compila e imprime `-5 , -5 .`

j) Dá exception.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String[] valores = new String[2];
        valores[0] = "Certification";
        valores[1] = "Java";
        Object[] vals = (Object[]) valores;
        vals[1] = "Daniela";
        System.out.println(vals[1].equals(valores[1]));
    }
}
```

a) O código não compila.

b) O código compila e dá erro em execução.

c) O código compila e imprime `false .`

d) O código compila e imprime `true .`

8) Quais das maneiras adiante são declarações e inicializações válidas para um array?

a) `int[] array = new int[10];`

b) `int array[] = new int[10];`

c) `int[] array = new int[];`

d) `int array[] = new int[];`

e) `int[] array = new int[2]{1, 2};`

f) `int[] array = new int[]{1, 2};`

g) `int[] array = int[10];`

h) `int[] array = new int[1, 2, 3];`

i) `int array[] = new int[1, 2, 3];`

j) `int array[] = {1, 2, 3};`

4.2 DECLARE, INSTANCIE, INICIALIZE E USE UM ARRAY MULTIDIMENSIONAL

Podemos generalizar a ideia de array para construir arrays de duas dimensões, em outras palavras, **array de arrays**. Analogamente, podemos definir arrays de quantas dimensões quisermos.

Declaração

```
// Um array de duas dimensões.
int[][] table;

// Um array de três dimensões.
int[][] cube[];

// Um array de quatro dimensões.
int[] [][]hipercube[];
```

Perceba que as dimensões podem ser definidas do lado esquerdo ou direito da variável.

Inicialização

Podemos inicializar as arrays com dimensões diferentes, como

no caso a seguir em que inicializamos a primeira dimensão com 10 e a segunda com 15:

```
int[][] table = new int[10][15];
```

Podemos também inicializar somente a primeira dimensão, deixando as outras para depois:

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

Podemos inicializar diretamente com valores que conhecemos, e nesse caso colocamos todas as dimensões:

```
int[][] test = new int[][]{{1,2,3},{3,2,1},{1,1,1}};
```

Acesso

O acesso tradicional é feito através de uma posição que desejamos, como no caso a seguir onde acessamos a primeira "linha", segunda "coluna":

```
System.out.println(table[0][1]);
```

Podemos criar um array que não precisa ser "quadrado", ele pode ter tamanhos estranhos:

```
int[][] weird = new int[2][];
weird[0] = new int[20];
weird[1] = new int[10];
for(int i = 0; i < weird.length;i++) {
    System.out.println(weird[i].length); // 20, 10
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
```

```
    public static void main(String[] args) {
        int zyx[][]=new int[3];
        int[]x=new int[20];
        int[]y=new int[10];
        int[]z=new int[30];
        zyx[0]=x;
        zyx[1]=y;
        zyx[2]=z;
        System.out.println(zyx[2].length);
    }
}
```

a) Não compila, erro ao declarar `zyx`.

b) Compila e dá erro ao tentar atribuir o segundo array a `zyx`.

c) Compila e dá erro ao tentar imprimir o tamanho do array.

d) Compila e imprime 10.

e) Compila e imprime 20.

f) Compila e imprime 30.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int zyx[][]=new int[3][];
        int[]x=new int[20];
        int[]y=new int[10];
        int[]z=new int[30];
        zyx[0]=x;
        zyx[1]=y;
        zyx[2]=z;
        System.out.println(zyx[2].length);
    }
}
```

a) Não compila, erro ao declarar `zyx`.

b) Compila e dá erro ao tentar atribuir o segundo array a `zyx`.

c) Compila e dá erro ao tentar imprimir o tamanho do array.

d) Compila e imprime 10.

e) Compila e imprime 20.

f) Compila e imprime 30.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int zyx[][]=new int[3][10];
        int[]x=new int[20];
        int[]y=new int[10];
        int[]z=new int[30];
        zyx[0]=x;
        zyx[1]=y;
        zyx[2]=z;
        System.out.println(zyx[2].length);
    }
}
```

a) Não compila, erro ao declarar `zyx`.

b) Não compila, erro ao atribuir arrays de tamanho diferente de `10` em `zyx`.

c) Compila e dá erro ao tentar atribuir o segundo array a `zyx`.

d) Compila e dá erro ao tentar imprimir o tamanho do array.

e) Compila e imprime 10.

f) Compila e imprime 20.

g) Compila e imprime 30.

4) Compila? Roda?

```
class Test {
    public static void main(String[] args){
        int[] id = new int[10];
        id[0] = 1.0;

        int[10][10] tb = new int[10][10];

        int[][][] cb = new int[][][];
    }
}
```

a) O código não compila.

b) O código compila e dá erro em execução.

c) O código compila e roda.

4.3 DECLARE E USE UMA ARRAYLIST

Nesta prova, dentre as coleções veremos somente a `ArrayList`, uma lista que usa internamente um array. Rápida no método `get`, pois sua estrutura interna permite acesso aleatório (*random access*) em tempo constante.

Jamais se esqueça de importar a `ArrayList`:

```
import java.util.ArrayList;
```

O primeiro passo é criar uma `ArrayList` vazia de `String`s:

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

A `ArrayList` herda diversos métodos abstratos e concretos. Veremos vários deles aqui, dentre esses, os principais para a

certificação, vindos da interface `Collection`. Por exemplo, para adicionar itens, fazemos:

```
ArrayList<String> names = new ArrayList<String>();
names.add("certification");
names.add("java");
```

Para remover e verificar a sua existência na lista:

```
ArrayList<String> names = new ArrayList<String>();
names.add("certification");
names.add("java");

System.out.println(names.contains("java")); // true
System.out.println(names.contains("c#")); // false

// true, encontrado e removido
boolean removed = names.remove("java");

System.out.println(names.contains("java")); // false
System.out.println(names.contains("c#")); // false
```

Note que o `remove` remove somente a primeira ocorrência daquele objeto.

Podemos também verificar o tamanho de nossa `ArrayList`:

```
ArrayList<String> names = new ArrayList<String>();
names.add("certification");
names.add("java");
System.out.println(names.size()); // 2
```

E convertê-la para um array:

```
ArrayList<String> names = new ArrayList<String>();
names.add("certification");
names.add("java");

Object[] objectArray = names.toArray();
```

Caso desejarmos um array de String, devemos indicar isso ao método `toArray` de duas formas diferentes:

```
ArrayList<String> names = new ArrayList<String>();
names.add("certification");
names.add("java");

String[] names2 = names.toArray(new String[0]);
String[] names3 = names.toArray(new String[names.size()]);
```

Ambas passam um array de String: o primeiro menor e o segundo com o tamanho suficiente para os elementos. Se ele possui o tamanho suficiente, ele mesmo será usado; enquanto que, se o tamanho não é suficiente, o `toArray` cria um novo array do mesmo tipo.

Além disso, podemos adicionar uma coleção inteira em outra:

```
ArrayList<String> names = new ArrayList<String>();
names.add("certification");
names.add("java");

ArrayList<String> countries = new ArrayList<String>();
countries.add("korea");
countries.add("brazil");

ArrayList<String> everything = new ArrayList<String>();
everything.addAll(names);
everything.addAll(countries);
System.out.println(everything.size()); // 4
```

Outros métodos são específicos da interface `List` e recebem uma posição específica onde você quer colocar ou remover algo do array usado na `ArrayList` . O método `get` devolve o elemento na posição desejada, lembrando de que começamos sempre com 0:

```
ArrayList<String> names = new ArrayList<String>();
names.add("certification");
System.out.println(names.get(0)); // certification
```

Já o método `add` foi sobrecarregado para receber a posição de inclusão:

```
ArrayList<String> names = new ArrayList<String>();
names.add("certification");
System.out.println(names.get(0)); // certification

names.add(0, "java");
System.out.println(names.get(0)); // java
System.out.println(names.get(1)); // certification
```

O mesmo acontece para o método `remove`:

```
ArrayList<String> names = new ArrayList<String>();
names.add("java");
names.add("certification");

String removed = names.remove(0); // retorna java
System.out.println(names.get(0)); // certification
```

E o método `set`, que serve para alterar o elemento em determinada posição:

```
ArrayList<String> names = new ArrayList<String>();
names.add("java");
names.set(0, "certification");

System.out.println(names.get(0)); // certification
System.out.println(names.size()); // 1
```

Os métodos `indexOf` e `lastIndexOf` retornam a primeira ou a última posição que possui o elemento desejado, respectivamente. Caso esse elemento não esteja na lista, ele retorna -1:

```
ArrayList<String> names = new ArrayList<String>();
names.add("guilherme");
names.add("mario");
names.add("paulo");
names.add("mauricio");
names.add("adriano");
names.add("alberto");
names.add("mario");

System.out.println(names.indexOf("guilherme")); // 0
```

```
System.out.println(names.indexOf("mario")); // 1
System.out.println(names.indexOf("john")); // -1
System.out.println(names.lastIndexOf("mario")); // 6
System.out.println(names.lastIndexOf("john")); // -1
```

Iterator e o enhanced for

A interface `Iterator` define uma maneira de percorrer coleções. Isso é necessário porque, em coleções diferentes de `List`, não possuímos métodos para pegar o enésimo elemento. Como, então, percorrer todos os elementos de uma coleção?

- `hasNext`: retorna um booleano indicando se ainda há elementos a serem percorridos por esse iterador.
- `next`: pula para o próximo elemento, devolvendo-o.
- `remove`: remove o elemento atual da coleção.

O código que costuma aparecer para percorrer uma coleção é o seguinte:

```
Collection<String> strings = new ArrayList<String>();
Iterator<String> iterator = strings.iterator();
while (iterator.hasNext()) {
    String current = iterator.next();
    System.out.println(current);
}
```

O `enhanced-for` também pode ser usado nesse caso:

```
Collection<String> strings = new ArrayList<String>();
for (String current : strings) {
    System.out.println(current);
}
```

O método equals em coleções

A maioria absoluta das coleções usa o método `equals` na hora de buscar por elementos, como nos métodos `contains` e `remove`. Se você deseja ser capaz de remover ou buscar elementos, terá de provavelmente sobrescrever o método `equals` para refletir o conceito de igualdade em que está interessado, e não somente a igualdade de referência (implementação padrão do método).

Cuidado ao tentar sobrescrever o método `equals`. Se você escrevê-lo recebendo um tipo específico em vez de `Object`, não o estará sobrescrevendo, e o `ArrayList` continuará invocando o código antigo, a implementação padrão de `equals`!

ArrayList e referências

Vale lembrar de que Java sempre trabalha com referências para objetos, e não cria cópias de objetos cada vez que os atribuímos a uma variável ou referência:

```
Client guilherme = new Client();
guilherme.setName("Guilherme");

ArrayList<Client> clients = new ArrayList<Client>();
clients.add(guilherme);

System.out.println(guilherme.getName()); // Guilherme
System.out.println(clients.get(0).getName()); // Guilherme

guilherme.setName("Silveira");

System.out.println(guilherme.getName()); // Silveira
System.out.println(clients.get(0).getName()); // Silveira
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o

arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        ArrayList<String> c = new ArrayList<String>();
        c.add("a");
        c.add("c");
        System.out.println(c.remove("a"));
    }
}
```

a) Não compila: erro ao declarar a `ArrayList`.

b) Não compila: erro ao invocar `remove`.

c) Compila e, ao rodar, imprime `a`.

d) Compila e, ao rodar, imprime `true`.

e) Compila e, ao rodar, imprime `false`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
    public static void main(String[] args) {
        ArrayList<String> c = new ArrayList<String>();
        c.add("a");
        c.add("c");
        System.out.println(c.remove("a"));
    }
}
```

a) Não compila: erro ao declarar a `ArrayList`.

b) Não compila: erro ao invocar `remove`.

c) Compila e, ao rodar, imprime `a`.

d) Compila e, ao rodar, imprime `true`.

e) Compila e, ao rodar, imprime `false`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
    public static void main(String[] args) {
        ArrayList<String> c = new ArrayList<String>();
        c.add("a");
        c.add("a");
        System.out.println(c.remove("a"));
        System.out.println(c.size());
    }
}
```

a) Não compila: erro ao declarar a `ArrayList`.

b) Não compila: erro ao invocar `remove`.

c) Compila e, ao rodar, imprime `a` e 0.

d) Compila e, ao rodar, imprime `true` e 0.

e) Compila e, ao rodar, imprime `a` e 1.

f) Compila e, ao rodar, imprime `true` e 1.

g) Compila e, ao rodar, imprime `a` e 2.

h) Compila e, ao rodar, imprime `true` e 2.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("a");list.add("b");
```

```
        list.add("a");list.add("c");
        list.add("a");list.add("b");
        list.add("a");
        System.out.println(list.lastIndexOf("b"));
    }
}
```

a) Não compila.

b) Compila e imprime -1.

c) Compila e imprime 0.

d) Compila e imprime 1.

e) Compila e imprime 2.

f) Compila e imprime 3.

g) Compila e imprime 4.

h) Compila e imprime 5.

i) Compila e imprime 6.

j) Compila e imprime 7.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
    public static void main(String[] args) {
        ArrayList<String> l = new ArrayList<String>();
        l.add("a");
        l.add("b");
        l.add(1,"amor");
        l.add(3,"love");
        System.out.println(l);
        String[] array = l.toArray();
        System.out.println(array[2]);
```

```
    }
}
```

a) Não compila.

b) Compila e imprime `"amor"`.

c) Compila e imprime `"b"`.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        ArrayList<String> b = new ArrayList<String>();
        ArrayList<String> c = new ArrayList<String>();
        b.add("a");c.add("c");
        b.add("b");c.add("d");
        a.addAll(b);
        a.addAll(c);
        System.out.println(a.get(0));
        System.out.println(a.get(3));
    }
}
```

a) Não compila.

b) Compila e imprime `a` e `d`.

c) Compila e imprime `c` e `b`.

d) Compila e não sabemos a ordem em que os elementos serão impressos.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
```

```
    public static void main(String[] args) {
        ArrayList<String> a = new ArrayList<String>();
        a.add("a", 0);
        a.add("b", 0);
        a.add("c", 0);
        a.add("d", 0);
        System.out.println(a.get(0));
        System.out.println(a.get(1));
        System.out.println(a.get(2));
        System.out.println(a.get(3));
    }
}
```

a) Não compila.

b) Compila e imprime `abcd`.

c) Compila e imprime `dcba`.

d) Compila e imprime `adcb`.

e) Compila e imprime `bcda`.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.*;
class A {
  public static void main(String[] args) {
    ArrayList<String> a = new ArrayList<String>();
    a.add(0,"b");
    a.add(0,"a");
    for(Iterator<String> i=a.iterator();i.hasNext();i.next()) {
        String element = i.next();
        System.out.println(element);
    }
}
}
```

a) Não compila.

b) Compila e imprime `a`.

c) Compila e imprime `a` e `b`.

d) Compila e imprime `b` e `a`.

e) Compila e imprime `b`.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.util.ArrayList;
class A {
    public static void main(String[] args) {
        ArrayList<String> ss = new ArrayList<String>();
        ss.add("a");
        ss.add("b");
        ss.add("c");
        ss.add("d");

        for(String s:ss){
            if(s.equals("c")) s = "b";
            else if(s.equals("b")) s= "c";
        }
        for(String s:ss) System.out.println(s);
    }
}
```

a) Não compila, `s` é `final` por padrão.

b) Compila e imprime `a`, `c`, `b`, `d`.

c) Compila e imprime `a`, `b`, `c`, `d`.

d) Compila e imprime `a`, `c`, `c`, `d`.

e) Compila e imprime `a`, `c`, `d`, `b`.

Capítulo 5

USANDO LAÇOS

5.1 CRIE E USE LAÇOS DO TIPO WHILE

Outra maneira de controlar o fluxo de execução de um programa é definir que um determinado trecho de código deve executar várias vezes, como uma repetição ou um laço. Uma linguagem como o Java oferece alguns tipos de laços para o programador escolher. O comando `while` é um deles.

```
int i = 1;
while (i < 10) {
    System.out.println(i);
    i++;
}
```

A sintaxe do `while` é a seguinte:

```
while (CONDITION) {
    // CODIGO
}
```

Assim como no `if` , a condição de um bloco `while` deve ser um booleano. Da mesma maneira, se o bloco de código tiver apenas uma linha, podemos omitir as chaves:

```
int i = 0;
while( i < 10)
    System.out.println(i++);
```

O corpo do `while` é executado repetidamente até que a condição se torne falsa. Em outras palavras, enquanto a condição for verdadeira.

É necessário tomar cuidado para não escrever um `while` infinito, ou seja, um laço que não terminaria se fosse executado.

```
int i = 1;
//Quando fica false?
while(i < 10){
    System.out.println(i);
}
```

Em casos em que é explícito que o loop será infinito, o compilador é esperto e não deixa compilar caso tenha algum código executável após o laço:

```
class A {
    int a() {
        while(true) { // true, true, true ...
            System.out.println("do something");
        }
        return 1; // compile error
    }
}
```

Mesmo que a condição use uma variável, pode ocorrer um erro de compilação, caso a variável seja final:

```
class A {
    int a() {
        final boolean RUNNING = true;
        while(RUNNING) { // true, true, true ...
            System.out.println("do something");
        }
        return 1; // compile error
    }
}
```

Agora, caso a variável não seja final, o compilador não tem

como saber se o valor vai mudar ou não, por mais explícito que possa parecer, e o código compila normalmente:

```
class A {
    int a() {
        boolean rodando = true; // não final
        while(rodando) { // true? false?
            System.out.println("do something");
        }
        return 1; // ok
    }
}
```

Caso o compilador detecte código dentro de um laço que nunca será executado, também teremos um erro de compilação:

```
    //unreachable statement, compile error.
    while(false) { /* code */ }

    //unreachable statement, compile error.
    while(1 > 2) { /* code */ }
```

Lembre-se de que o compilador só consegue analisar operações com literais ou com constantes. No caso a seguir, o código compila, mesmo nunca sendo executado:

```
int a = 1;
int b = 2;
while(a > b){ // ok
    System.out.println("OI");
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int a = 10;
```

```
        while(a>100) a++;
        System.out.println(a);
    }
}
```

a) Não compila, pois nunca entra no loop.

b) Compila e imprime 99.

c) Compila e imprime 100.

d) Compila e imprime 101.

e) Compila e imprime outro valor.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        boolean run = true;
        while(run) {
            System.out.println(run);
        }
        System.out.println("finish");
    }
}
```

a) Transformar a variável em `final` faz o código compilar.

b) Colocar uma linha dentro do laço que faz `run = false` faz o código compilar.

c) O código compila e roda em loop infinito.

d) O código compila e roda, após algumas passagens pelo laço ele imprime uma exception e para.

5.2 CRIE E USE LAÇOS DO TIPO FOR, INCLUINDO O ENHANCED FOR

Observando um pouco os códigos que utilizam `while`, dá para perceber que eles são formados por quatro partes: inicialização, condição, comandos e atualização.

```
int i = 1; // Inicialização
while (i < 10) { // Condição
    System.out.println(i); // Comandos
    i++; // Atualização
}
```

A inicialização é importante para que o laço execute adequadamente. Mesmo com essa importância, a inicialização fica separada do `while`.

A atualização é fundamental para que não aconteça um "loop infinito". Porém, a sintaxe do `while` não a coloca em evidência.

Há um outro laço que coloca em destaque a inicialização, a condição e a atualização. Esse laço é o `for`.

```
for (int i = 1; i < 10; i++) {
    System.out.println(i);
}
```

O `for` tem três argumentos separados por `;` (ponto e vírgula). O primeiro é a inicialização; o segundo, a condição; e o terceiro, a atualização.

A inicialização é executada somente uma vez no começo do `for`. A condição é verificada no começo de cada rodada (iteração). A atualização é executada no fim de cada iteração.

Todos os três argumentos do `for` são opcionais. Desta forma,

você poderia escrever o seguinte código:

```
for(;;){
    // CODE
}
```

O que acontece com esse laço? Para responder essa pergunta, é necessário saber quais são os "valores default" colocados nos argumentos do `for`, quando não é colocado nada pelo programador. A inicialização e a atualização ficam realmente vazias. Agora, a condição recebe por padrão o valor `true`. Então, o código anterior depois de compilado fica assim:

```
for (;true;){ // true, true, true ...
    // CODE
}
```

Nos exemplos anteriores, basicamente o que fizemos na inicialização foi declarar e inicializar apenas uma variável qualquer. Porém, é permitido declarar diversas variáveis de um mesmo tipo ou inicializar diversas variáveis.

Na inicialização, não é permitido declarar variáveis de tipos diferentes. Mas é possível inicializar variáveis de tipos diferentes. Veja os exemplos:

```
// Declarando três variáveis do tipo int e inicializando as três.
// Repare que o "," separa as declarações e inicializações.
for (int i = 1, j = 2, k = 3;;){
    // CODIGO
}

// Declarando três variáveis de tipos diferentes
int a;
double b;
boolean c;

// Inicializando as três variáveis já declaradas
for (a = 1, b = 2.0, c = true;;){
```

```
    // CODE
}
```

Na atualização, é possível fazer diversas atribuições separadas por `,` (vírgula).

```
//a cada volta do laço, incrementamos o i e decrementamos o j
for (int i=1,j=2;; i++,j--){
    // code
}
```

Como já citamos anteriormente, não é possível inicializar variáveis de tipos diferentes:

```
for (int i=1, long j=0; i< 10; i++){ // compile error
    // code
}
```

No campo de condição, podemos passar qualquer expressão que resulte em um `boolean`. São exatamente as mesmas regras do `if` e `while`.

No campo de atualização, não podemos só usar os operadores de incremento, podemos executar qualquer trecho de código:

```
    for (int i = 0; i < 10; i += 3) { // somatório
        // code
    }

    for (int i = 0; i < 10; System.out.println(i++)) { // bizarro
        // code
    }
```

Enhanced for

Quando vamos percorrer uma coleção de objetos (*collection*) ou um array, podemos usar uma versão simplificada do `for` para percorrer essa coleção de maneira simplificada. Essa forma simplificada é chamada de *enhanced for*, ou *foreach*:

```
int[] numbers = {1,2,3,4,5,6};
for (int num : numbers) { // enhanced for
    System.out.println(num);
}
```

A sintaxe é mais simples, temos agora duas partes dentro da declaração do `for` :

```
for(VARIABLE : COLLECTION){
    CODE
}
```

Nesse caso, declaramos uma variável que receberá cada um dos membros da coleção ou array que estamos percorrendo. O próprio `for` vai a cada iteração do laço atribuir o próximo elemento da lista à variável. Seria o equivalente a fazer o seguinte:

```
int[] numbers = {1,2,3,4,5,6};

for( int i=0; i < numbers.length; i++){
    int num = numbers[i]; //declaração da variável e atribuição
    System.out.println(num);
}
```

Se fosse uma coleção, o código fica mais simples ainda se comparado com o `for` original:

```
ArrayList<String> names = //lista com vários nomes

//percorrendo a lista com o for simples
for(Iterator<String> iterator = names.iterator();
    iterator.hasNext();){
    String name = iterator.next();
    System.out.println(name);
}

//percorrendo com o enhanced for
for (String name : nomes) {
    System.out.println(name);
}
```

Existem, porém, algumas limitações no *enhanced for*. Não podemos, por exemplo, modificar o conteúdo da coleção que estamos percorrendo usando a variável que declaramos:

```
ArrayList<String> names = //lista com vários nomes

//tentando remover nomes da lista
for (String name : names) {
    name = null;
}

//o que imprime abaixo?
for (String name : names) {
    System.out.println(name);
}
```

Ao executar esse código, você perceberá que a coleção não foi modificada, nenhum elemento mudou de valor para `null`.

Outra limitação é que não há uma maneira natural de saber em qual iteração estamos, já que não existe nenhum contador. Para saber em qual linha estamos, precisaríamos de um contador externo. Também não é possível percorrer duas coleções ao mesmo tempo, já que não há um contador centralizado. Para todos esses casos, é recomendado usar o `for` simples.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        for(;;) {
            System.out.println("a");
        }
        System.out.println("b");
    }
```

```
}
```

a) Não compila.

b) Compila e imprime `a` infinitamente.

c) Compila e imprime `b`.

d) Compila e imprime `a`, depois `b`, depois para.

e) Compila, imprime `a` diversas vezes e depois joga um `StackOverflowError`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        for(;false;) {
            System.out.println("a");
            break;
        }
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `b`.

c) Compila, imprime `a` e `b`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
  public static void main(String[] args) {
    for(int i=0, int j=1; i<10; i++, j++) System.out.println(i);
}
}
```

a) Não compila.

b) Compila e imprime 0 até 9.

c) Compila e imprime 0 até 10.

d) Compila e imprime 1 até 10.

e) Compila e imprime 1 até 11.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        for(int i=0, j=1; i<10 ;i++, j++) System.out.println(i);
    }
}
```

a) Não compila.

b) Compila e imprime 0 até 9.

c) Compila e imprime 0 até 10.

d) Compila e imprime 1 até 10.

e) Compila e imprime 1 até 11.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        for(int i=0; i<10, false; i++) {
            System.out.println('a');
        }
        System.out.println('b');
    }
}
```

a) Não compila.

b) Compila e imprime `'a' e 'b'`.

c) Compila e imprime `'b'`.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        for(int i=0; i<2; i++, System.out.println(i)) {
            System.out.println(i);
        }
    }
}
```

a) Não compila.

b) Compila e imprime `0 1 2`.

c) Compila e imprime `0 0 1 1 2 2`.

d) Compila e imprime `0 1 1 2 2`.

e) Compila e imprime `0 1 1 2`.

5.3 CRIE E USO LAÇOS DO TIPO DO/WHILE

Uma outra opção de laço `while` seria o `do .. while`, que é bem parecido com o `while`. A grande diferença é que a condição é testada após o corpo do *loop* ser executado pelo menos uma vez:

```
int i = 1;
do { //executa ao menos 1 vez
    System.out.println(i);
    i++;
} while (i < 10); // se der true, volta e executa novamente.
```

A condição do `do .. while` só é verificada no final de cada iteração e não no começo, como no `while`. Repare que, ao final do bloco `do .. while`, existe um ponto e vírgula. Esse é um detalhe que passa desapercebido muitas vezes, mas que resulta em erro de compilação se omitido:

```
int i = 1;
do {
    System.out.println(i);
    i++;
} while (i < 10) // não compila, faltou o ;
```

Assim como no `while`, caso tenhamos apenas uma linha, as chaves podem ser omitidas. Caso exista mais de uma linha dentro do `do .. while` e não existam chaves, teremos um erro de compilação:

```
int i = 0;
//compila normal
do
    System.out.println(i++);
while(i<10);

//erro, mais de uma linha dentro do do .. while
do
    System.out.print("value: "); //erro
    System.out.println(i++);
while(i<10);
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        boolean i = false;
        do {
            System.out.println(i);
```

```
        } while(i);
    }
}
```

a) Não compila.

b) Compila e imprime `false`.

c) Compila e não imprime nada.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        if(args.length < 10) {
            do {
                if(args.length>2) return;
            } while(true);
        }
        System.out.println("finished");
    }
}
```

a) Não compila.

b) Compila e entra em loop infinito caso seja passado zero, um ou dois argumentos. Não imprime nada caso 3 a 9 argumentos. Imprime `'finished'` caso 10 ou mais argumentos.

c) Compila e entra em loop infinito caso seja passado zero, um ou dois argumentos. Caso contrário, imprime `'finished'`.

d) Compila e sempre entra em loop infinito.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

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

```
        int i = 0;
        do System.out.println(i); while(i++>10);
    }
}
```

a) Não compila.

b) Compila e não imprime nada.

c) Compila e imprime `0`.

d) Compila e imprime de `0` até `9`.

e) Compila e imprime de `0` até `10`.

f) Compila e não imprime nada.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int i = 0;
        do System.out.println(i) while(i++<10);
    }
}
```

a) Não compila.

b) Compila e imprime de `0` até `9`.

c) Compila e imprime de `0` até `10`.

d) Compila e não imprime nada.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

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

```
        int i = 0;
        do; while(i++<10);
    }
}
```

a) Não compila.

b) Compila e entra em loop infinito.

c) Compila e sai.

5.4 COMPARE OS TIPOS DE LAÇOS

Embora o `for`, `while` e `do .. while` sejam todos estruturas que permitam executar *loops*, existem similaridades e diferenças entre essas construções que serão cobradas na prova.

Comparando while e do .. while

No caso do `while` e `do while`, ambos são muito similares, sendo a principal diferença o fato do `do .. while` ter a condição testada somente após executar o código de dentro do *loop*, pelo menos uma vez.

```
int i = 20;

//imprime 20, já que só faz o teste após a execução do código
do {
    System.out.println(i);
    i++;
} while(i < 10);

int j = 20;

//não imprime nada, já que testa antes de executar o bloco
while(j < 10){
    System.out.println(j);
    j++;
```

```
}
```

Comparando for e enhanced for

Apesar de ser estruturalmente mais complexo, o `for` simples é mais poderoso que o `enhanced for`. Com o `enhanced for`, não podemos:

- Percorrer mais de uma coleção ao mesmo tempo;
- Remover os elementos da coleção;
- Inicializar um array.

Caso desejemos fazer uma iteração de **leitura, por todos os elementos da coleção**, aí sim o `enhanced for` é a melhor opção.

Comparando while e for

Ambas estruturas de laço permitem executar as mesmas operações em nosso código, e são bem similares. Mas, apesar disso, existem situações em que o código ficará mais simples caso optemos por uma delas.

Geralmente, optamos por usar `for` quando sabemos a quantidade de vezes que queremos que o laço seja executado. Pode ser percorrer todos os elementos de uma coleção (onde sabemos a quantidade de vezes que o loop será executado por saber o tamanho da coleção), ou simplesmente executar o laço uma quantidade fixa de vezes.

Usamos o `while` ou `do .. while` quando não sabemos a quantidade de vezes em que o laço será executado. Mas sabemos uma condição que, enquanto for verdadeira, fará com que o laço seja repetido.

O exemplo a seguir mostra um código no qual conhecemos a condição de parada. Porém, não faz sentido nenhum ter uma variável inicializada ou uma condição de incremento, então escolhemos um `while`:

```
while(account.getAmount() > 0) {
    account.withdraw(1000);
}
```

Note que, caso queira contar quantas vezes foi sacado, faria sentido usar um `for`:

```
int total;
for(total = 0; account.getAmount() > 0; total++) {
    account.withdraw(1000);
}
System.out.println("Withdraw " + total + " times");
```

Exercícios

1) Qual o laço mais simples de ser usado quando desejamos iterar por duas coleções ao mesmo tempo?

a) for

b) while

c) enhanced for

d) do... while

2) Qual o melhor laço a ser usado para, dependendo do valor de um elemento, removê-lo de nossa lista?

a) for

b) enhanced for

3) Para todos os números entre `i` e `100`, devo imprimir algo, sendo que, mesmo que `i` seja maior que `100`, devo imprimir algo pelo menos uma vez. Qual laço devo usar?

a) enhanced for

b) do... while

c) while

4) Qual o laço a ser usado caso queira executar um código eternamente?

a) enhanced for

b) for

c) for ou while

d) for ou while ou do...while

e) while ou do...while

5) Qual laço deve ser usado para inicializar os valores de um array?

a) enhanced for

b) for

5.5 USE BREAK E CONTINUE

Em qualquer estrutura de laço, podemos aplicar os controladores `break` e `continue`. O `break` serve para parar o laço totalmente. Já o `continue` interrompe apenas a iteração

atual. Vamos ver alguns exemplos:

```
int i = 1;
while (i < 10) {
    i++;
    if (i == 5)
        break; // sai do while com i valendo 5
    System.out.println(i);
}
System.out.println("End");
```

Ao executar o `break`, a execução do `while` para completamente. Temos a seguinte saída:

```
2
3
4
End
```

Vamos comparar com o `continue`:

```
int i = 1;
while (i < 10) {
    i++;
    if (i == 5)
        continue; // vai para a condição com o i valendo 5
    System.out.println(i);
}
```

Neste caso, pararemos a execução da iteração apenas quando o valor da variável for igual a 5. Ao encontrar um `continue`, o código volta ao início da iteração, ao ponto do loop. Nossa saída agora é a seguinte:

```
2
3
4
6
7
8
9
10
```

```
End
```

Isto é, o `break` quebra o laço atual, enquanto o `continue` vai para a próxima iteração do laço.

Tome cuidado, pois um laço que tenha um `while` infinito do tipo `true` e que contenha um `break` é compilável, já que o compilador não sabe se o código poderá parar, possivelmente sim:

```
while(true) {
    if(1==2) break; // lies, lies...
    System.out.println("infinite loop");
}
```

Os controladores de laços, `break` e `continue`, podem ser aplicados no `for`. O `break` se comporta da mesma maneira que no `while` e no `do .. while`, parar o laço por completo. Já o `continue` faz com que a iteração atual seja abortada, executando em seguida a parte de *atualização* do `for`, e em seguida a de *condição*. Vamos ver o exemplo a seguir:

```
for (int i = 1; i < 10; i++) {
    if (i == 8) {
        break; // sai do for sem executar mais nada do laço.
    }
    if (i == 5) {
        // pula para a atualização sem executar o resto do corpo.
        continue;
    }
    System.out.println(i);
}
```

A saída desse código é:

```
1
2
3
4
6
7
```

Rótulos em laços (labeled loops)

Às vezes, encontramos a necessidade de "encaixar" um laço dentro de outro. Por exemplo, um `for` dentro de um `while` ou de outro `for`. Nesses casos, pode ser preciso manipular melhor a execução dos laços encaixados com os controladores de laços, `break` e `continue`.

```
for (int i = 1; i < 10; i++) { //laço externo
    for (int j = 1; j < 10; j++) { // laço interno
        if (i * j == 25)    {
            break; // qual for será quebrado?
        }
    }
}
```

Quando utilizamos o `break` ou o `continue` em laços encaixados, eles são aplicados no laço mais próximo. Por exemplo, nesse código, o `break` vai "quebrar" o `for` mais interno. Se fosse preciso "quebrar" o `for` mais externo, como faríamos?

Labeled statements

Podemos adicionar *labels* (rótulos) a algumas estruturas de código, e usá-los posteriormente para referenciarmos essas estruturas. Para declarar um label, usamos um nome qualquer (mesma regra de nomes de variáveis etc.) seguido de dois pontos (`:`). Por exemplo, podemos dar um label para um `for` como o que segue:

```
externo: //label
for(int i=0; i<10;i++){
    // code
}
```

Podemos usar esses *labels* para referenciar para qual loop queremos que o `break` ou o `continue` seja executado:

```
external: for (int i = 1; i < 10; i++) {
  internal: for (int j = 1; j < 10; j++) {
    if (i * j == 25)    {
        break external; // quebrando o for externo
    }
    if (i * j == 16) {
        continue internal; // pulando um iteração do for interno
    }
  }
}
```

LABEL HTTP

O código a seguir imprime os valores de 1 a 10. Mas como ele compila sendo que temos uma URI logo antes do laço `for` ?

```
http://www.casadocodigo.com.br
http://www.codecrushing.com
    for (int i = 1; i <= 10; i++) {
        System.out.println(i);
    }
}
```

Um rótulo ou label pode estar presente antes de um *statement* qualquer, mas só podemos utilizar um *statement* de `break` ou `continue` caso o rótulo esteja referenciando um `for` , `while` ou `switch` :

```
void labelAnywhere() {
    myLabel: System.out.println("hi");
}

void continueSomeRandomLabelDoesNotCompile() {
    myLabel: System.out.println("hi");
```

```
    if(1<10) continue myLabel; // erro de compilação
}
```

Cuidado, mesmo dentro de um `for` ou similar, o `continue` e o `break` só funcionarão se forem relativos a um label dentro do qual estão, e do tipo `for`, `do...while`, `switch` ou `while`. Vale lembrar de que `switch` só aceita `break`.

```
void labelAnywhereIsOkBreakAnywhereIsNotOk() {
    myLabel: System.out.println("hi");
    for(int i=0;i < 10;i++) {
        break myLabel; // compile error
    }
}
void anotherLoopLabel() {
    myLabel:
    for(int i=0;i < 10;i++) {
        System.out.println("hi");
    }
    for(int i=0;i < 10;i++) {
        break myLabel; // compile error
    }
}
```

Rótulos podem ser repetidos desde que não exista conflito de escopo:

```
void repeatedLabel() {
    myLabel: for (int i = 0; i < 10; i++) {
        break myLabel;
    }
    myLabel: for (int i = 0; i < 10; i++) {
        break myLabel;
    }
}
void sameNameNestedLabelsDoesNotCompile() {
    myLabel: for (int i = 0; i < 10; i++) {
        // compile error
        myLabel: for (int j = 0; j < 10; j++) {
            break myLabel;
        }
    }
```

```
    }
```

Não há conflito de nome entre rótulos e variáveis, pois seu uso é bem distinto. O compilador sabe se você está referenciando um rótulo ou uma variável:

```
class A {
    int myLabel = 15;
    void labelNameAndVariableNameIsOk() {
        myLabel: for (int i = 0; i < 10; i++) {
            int myLabel = 10;
            break myLabel;
        }
    }
}
```

Um mesmo statement pode ter dois labels:

```
    void twoLabelsInTheSameStatement() {
        first: second: for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
```

Tome bastante cuidado com `breaks` e `continues` que são de `switch`, mas parecem ser de `fors`:

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

        for(int i = 0; i < 4; i++) {
            System.out.println("Before switch");
            mario:
            guilherme: switch(i) {
                case 0:
                case 1:
                    System.out.println("Case " + i);
                    for(int j = 0; j < 3; j++) {
                        System.out.println(j);
                        if(j==1) break mario;
                    }
                case 2:
                    System.out.println("At i = " + i);
```

```
                    continue;
                case 3:
                    System.out.println("At 3");
                    break;
                default:
                    System.out.println("Weird...");
                    break;
            }
            System.out.println("After switch");
        }
    }
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
         outside: for(int a=0;a<30;a++)
         for(int b=0;b<1;b++)
         if(a+b==25) continue outside;
         else if(a+b==20) break outside;
         if(a==0) break outside;
         else System.out.println(a);
    }
}
```

a) Não compila.

b) Compila e imprime 1 até 29.

c) Compila e não imprime nada.

d) Compila e imprime 1 até 19, 21 até 24, 26 até 29.

e) Compila e imprime 1 até 24, 26 até 29.

f) Compila e imprime 1 até 19.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
         outside: for(int a=0;a<30;a++)
         for(int b=0;b<1;b++)
         if(a+b==25) continue outside;
         else if(a+b==20) break;
         else System.out.println(a);
    }
}
```

a) Não compila.

b) Compila e imprime 0 até 29.

c) Compila e não imprime nada.

d) Compila e imprime 0 até 19, 21 até 24, 26 até 29.

e) Compila e imprime 0 até 24, 26 até 29.

f) Compila e imprime 0 até 19.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int a = args.length;
        int i = 0;
        switch(a) {
            case 0:
            case 1:
            for(i=0;i<15;i++, System.out.println(i))
            if(i==5) continue;
            if(i==15) break;
            case 2:
            System.out.println("2");
        }
```

```
        System.out.println("end");
    }
}
```

a) Não compila.

b) Compila e ao rodar com 0 argumentos imprime 0 até 14, 2, end.

c) Compila e ao rodar com 0 argumentos imprime 0 até 15, 2, end.

d) Compila e ao rodar com 0 argumentos imprime 0 até 4, 6 até 14, 2, end.

e) Compila e ao rodar com 0 argumentos imprime 1 até 4, 6 até 15, end.

f) Compila e ao rodar com 0 argumentos imprime 0 até 4, 6 até 9, 2, end.

g) Compila e ao rodar com 0 argumentos imprime 1 até 4, 6 até 9, end.

h) Compila e ao rodar com 0 argumentos imprime 1 até 4, 6 até 15, 2, end.

i) Compila e ao rodar com 0 argumentos imprime 1 até 15, 2, end.

j) Compila e ao rodar com 0 argumentos imprime 1 até 15, end.

Capítulo 6

TRABALHANDO COM MÉTODOS E ENCAPSULAMENTO

6.1 CRIE MÉTODOS COM ARGUMENTOS E VALORES DE RETORNO

Classes, *enums* e interfaces podem ter métodos definidos em seus corpos. Todo método tem uma *assinatura* (também chamada de *interface*), enquanto todos os métodos não abstratos devem possuir um *corpo*.

A assinatura de um método é composta pelas seguintes partes:

- Um modificador de visibilidade (nem que seja implícito, *package-private*);
- Um tipo de retorno;
- Um nome seguindo as regras de identificadores;
- Um conjunto de parâmetros (pode ser vazio), cada um com seu nome e seu tipo.

Outros modificadores da assinatura são opcionais e somente alguns são cobrados na prova:

- `final` — Em caso de herança, o método não pode ser sobrescrito nas classes filhas.
- `abstract` — Obriga as classes filhas a implementarem o método. O método abstrato **não** pode ter corpo definido.
- `static` — O método deixa de ser um membro da instância e passa a ser acessado diretamente através da classe.
- `synchronized` — Não cai nesta prova. *Lock* da instância.
- `native` — Não cai nesta prova. Permite a implementação do método em código nativo (*JNI*);
- `strictfp` — Não cai nesta prova. Ativa o modo de portabilidade matemática para contas de ponto flutuante.
- `throws <EXCEPTIONS>` — Após a lista de parâmetros, podemos indicar quais exceptions podem ser jogadas pelo método.

A ordem dos elementos na assinatura dos métodos é sempre a seguinte, sendo que os modificadores podem aparecer em qualquer ordem dentro da seção `MODIFICADORES : <MODIFICADORES> <TIPO_RETORNO> <NOME> (<PARÂMETROS>) <THROWS_EXCEPTIONS>`

Parâmetros

Em Java, usamos parâmetros em métodos e construtores. Definimos uma lista de parâmetros sempre declarando seus tipos e nomes, e separando por vírgula:

```
class Param {
    void test(int a, int b) {

    }
}
```

Invocamos um método passando os parâmetros separados por vírgula:

```
p.test(1, 2);
```

Enquanto a declaração das variáveis é feita na declaração do método, a inicialização dos valores é feita por quem chama o método. Note que, em Java, não é possível ter valores `default` para parâmetros e todos são obrigatórios, não podemos deixar de passar nenhum, como mostra o exemplo a seguir:

```
class ParamTest {
    void test() {
        print("guilherme"); // compile error
        print(33); // compile error
        print("guilherme", 33); // ok
    }
    void print(String name, int age) {
        System.out.println(name + " " + age);
    }
}
```

O único modificador possível de ser marcado em parâmetros é `final`, para indicar que aquele parâmetro não pode ter seu valor modificado depois da chamada do método:

```
class Param {
    void test (final int a) {
        a = 10; // compile error
    }
}
```

Promoção em parâmetros

Temos de saber que nossos parâmetros também estão sujeitos à promoção de primitivos e ao polimorfismo. Por exemplo, a classe a seguir ilustra as duas situações:

```
class Param {
    void primitive (double d) {

    }

    void reference (Object o) {

    }
}
```

O primeiro método espera um `double`. Mas se chamarmos passando um `int`, um `float` ou qualquer outro tipo compatível mais restrito, este será promovido automaticamente a `double` e a chamada funciona:

```
Param p = new Param();
p.primitive(10);
p.primitive(10L);
p.primitive(10F);
p.primitive((short) 10);
p.primitive((byte) 10);
p.primitive('Z');
```

A mesma coisa ocorre com o método que recebe `Object`: podemos passar qualquer um que **é um** `Object`, ou seja, qualquer objeto:

```
Param p = new Param();
p.reference(new Car());
p.reference(new Moto());
```

Retornando valores

Todo método pode retornar um valor ou ser definido como "retornando" `void`, quando não devolve nada:

```
class A {
    int number() {
        return 5;
    }
    void nothing() {
        return;
    }
}
```

No caso de métodos de tipo de retorno `void` (nada), podemos omitir a última instrução:

```
class A {
    void nothing() {
        // return; // optional
    }
}
```

Um método desse tipo também pode ter um retorno antecipado, um `early return`:

```
class A {
    void nothing(int i) {
        if(i >= 0) return; // early return
        System.out.println("negative");
    }
}
```

Não podemos ter nenhum código que seria executado após um retorno, uma vez que o compilador detecta que o código jamais será executado:

```
class A {
    void nothing(int i) {
        if(i >= 0) {
            return;

            // compile error
            System.out.println(">= 0");
        }
        System.out.println("< 0");
    }
```

```
}
```

Todo método que possui um tipo de retorno definido (isto é, diferente de `void`) deve retornar algo, ou jogar uma `Exception` em cada um dos caminhos de saída possíveis do método; caso contrário, o código não compila. O exemplo a seguir é uma situação na qual ocorre um erro de compilação devido a nem todos os caminhos possuirem um `return` ou `throw`:

```
String method(int a) {
    if(a > 0) {
        return "> 0";
    } else if(a < 0) {
        return "< 0";
    }
    // compile error
}
```

Lembre-se de que isso é feito pelo compilador, então ele não conhece os valores da variável `a`, nem sabe dizer quais serão tais valores em tempo de execução, portanto, não podendo concluir se todos os casos foram cobertos, como mostra o exemplo a seguir:

```
String method(int a) {
    if(a > 0) {
        return "> 0";
    } else if(a <= 0) {
        return "<= 0";
    }
    // compile error
}
```

Podemos jogar uma `exception` ou colocar um `return`:

```
String method(int a) {
    if(a > 0) {
        return "positive";
    } else if(a < 0) {
        return "negative";
    }
```

```
    return "zero";
}

String method2(int a) {
    if(a > 0) {
        return "positive";
    } else if(a < 0) {
        return "negative";
    }
    throw new RuntimeException("zero!");
}
```

Métodos que não retornam nada não podem ter seu resultado atribuído a uma variável:

```
void method() {
    System.out.println("hi");
}
void method2() {
    // compile error
    int i = method();
}
```

Pelo outro lado, mesmo que um método retorne algo, seu retorno pode ser ignorado:

```
int method() {
    System.out.println("hi");
    return 5;
}
void method2() {
    int i = method(); // i = 5, ok
    method(); // ok
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

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

```
        x(args.length);
    }
    static int x(final int l) {
        for(int i=0;i<100;i++) {
            switch(i) {
                case l:
                    System.out.println(l);
                    if(l==i) return;
                case 0:
                    System.out.println(0);
            }
        }
        System.out.println("end");
        return -1;
    }
}
```

a) Não compila.

b) Compila e, ao rodar com cinco parâmetros, imprime 0, 5 e end.

c) Compila e, ao rodar com cinco parâmetros, imprime 0, 5, -1 e end.

d) Compila e, ao rodar com cinco parâmetros, imprime 0 e 5.

e) Compila e, ao rodar com cinco parâmetros, imprime 0, 5 e -1.

f) Compila e, ao rodar com cinco parâmetros, imprime 0, 5, 0 e end.

g) Compila e, ao rodar com cinco parâmetros, imprime 0, 5, 0, -1 e end.

h) Compila e, ao rodar com cinco parâmetros, imprime 0 e 5.

i) Compila e, ao rodar com cinco parâmetros, imprime 0, 5, 0 e

-1.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        x(args.length);
    }
    static int x(final int l) {
        for(int i=0;i<100;i++) {
            switch(i) {
                case l:
                    System.out.println(l);
                    if(l==i) return 3;
                case 0:
                    System.out.println(0);
            }
        }
        System.out.println("End");
        return -1;
    }
}
```

a) Não compila.

b) Compila e, ao rodar com cinco parâmetros, imprime 0, 5 e End.

c) Compila e, ao rodar com cinco parâmetros, imprime 0, 5, -1 e End.

d) Compila e, ao rodar com cinco parâmetros, imprime 0 e 5.

e) Compila e, ao rodar com cinco parâmetros, imprime 0, 5 e -1.

f) Compila e, ao rodar com cinco parâmetros, imprime 0, 5, 0 e End.

g) Compila e, ao rodar com cinco parâmetros, imprime 0, 5, 0, -1 e End.

h) Compila e, ao rodar com cinco parâmetros, imprime 0 e 5.

i) Compila e, ao rodar com cinco parâmetros, imprime 0, 5, 0 e -1.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        System.out.println(a(args.length));
    }
    static int a(int l) {
        if(l<10) return b(l);
        else return c();
    }
    static int b(int l) {
        if(l<10) return b(l);
        else return c();
    }
    static long c() {
        return 3;
    }
}
```

a) Não compila: erro ao invocar o método `b`.

b) Não compila: erro ao invocar o método `c`.

c) Não compila por um motivo não listado.

d) Compila e, ao chamar com 15 argumentos, imprime 3.

e) Compila e, ao chamar com 15 argumentos, entra em loop infinito.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        System.out.println(a(args.length)[0]);
    }
    static int,int a(int l) {
        if(l==0) return {0, l};
        else return {l, 0};
    }
}
```

a) Não compila.

b) Ao invocar com nenhum parâmetro, imprime 0.

c) Ao invocar com 5 parâmetros, imprime 0.

6.2 APLIQUE A PALAVRA CHAVE STATIC A MÉTODOS E CAMPOS

O modificador estático diz que determinado atributo ou método pertence à classe, e não a cada objeto. Com isso, você não precisa de uma instância para acessar o atributo, basta o nome da classe.

```
public class Car {
    public static int totalCars;
}
```

E depois, para acessar:

```
Car.totalCars = 5;
```

Um método estático é um método da classe, podendo ser chamado sem uma instância:

```
public class Car {
    private static int totalCars = 15;

    public static int getTotalCars () [
        return totalCars;
    }
}

public class Test {
    public static void main(String[] args) {
        int total = Car.getTotalCars();
        System.out.println(total); // 15
    }
}
```

O que não podemos fazer é usar um método/atributo de instância de dentro de um método estático:

```
public class Car{
    private static int totalCars;
    private int weight;

    public static int getWeight() {
        return weight; // compile error
    }
}
```

Esse código não compila, pois `weight` é um atributo de instância. Se alguém chamar esse método, que valor ele retornaria, já que não estamos trabalhando com nenhuma instância de carro em específico?

Repare que a variável estática pode acessar um método estático, e esse método acessar algo ainda não definido e ter um resultado inesperado à primeira vista:

```
    static int b = getMethod();
    public static int getMethod() {
        return a;
    }
    static int a = 15;
```

O valor de `b` será 0, e não 15, uma vez que a variável `a` ainda não foi inicializada e possui seu valor padrão da execução do método `getMethod`.

Outro caso interessante é que uma variável estática pode acessar outra estática. O exemplo a seguir mostra uma situação válida:

```
static int first = 10;
static int second = first + 5; // ok
```

Já o exemplo a seguir mostra a tentativa de uso de uma variável estática que só será inicializada posteriormente na invocação do `inicializa`:

```
static int another;
static void inicializa() {
    another = 10;
}
static int one = another + 1; // 0 + 1 = 1
```

Um detalhe importante é que membros estáticos podem ser acessados através de instâncias da classe (além do acesso direto pelo nome da classe).

```
Car c = new Car();
int i = c.getTotalCars();
```

Cuidado com essa sintaxe, que pode levar a acreditar que é um método de instância. É uma sintaxe estranha, mas que compila e acessa o método estático normalmente.

Além disso, esteja atento pois, caso uma classe possua um método estático, ela não pode possuir outro método não estático com assinatura que a sobrescreveria (mesmo que em classe mãe/filha):

```
class A {
    static void a() { // compile error
    }
    void a() { // compile error
    }
}

class B {
    static void a() {
    }
}
class C extends B {
    void a() { // compile error
    }
}
```

Outro ponto importante a tomar nota é que o *binding* do método é feito em compilação, portanto, o método invocado não é detectado em tempo de execução. Leve em consideração:

```
class A {
    static void method() {
        System.out.println("a");
    }
}

class B extends A {
    static void method() {
        System.out.println("b");
    }
}
```

Caso o tipo referenciado de uma variável seja `A` em tempo de compilação, o método será o da classe `A`. Se for referenciado como `B`, será o método da classe `B`:

```
    A a = new A();
    a.method(); // a

    B b = new B();
    b.method(); // b
```

```
    A a2 = b;
    a2.method(); // a
```

A inicialização de uma variável estática pode invocar também métodos estáticas:

```
class A {
    static int idade = grabAge();
    static int grabAge() {
        return 18;
    }
}
```

A palavra-chave `static` pode ser aplicada a classes aninhadas, mas este tópico não é cobrado nesta primeira certificação.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        x();
    }
    static x() {
        System.out.println("x");
        y();
    }
    static y() {
        System.out.println("y");
    }
}
```

a) Não compila.

b) Imprime `x`, `y`.

c) Imprime `y`, `x`.

d) Imprime `x`.

e) Imprime `y`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        x();
    }
    static void x() {
        System.out.println("x");
        y();
    }
    static void y() {
        System.out.println("y");
    }
}
```

a) Não compila.

b) Imprime `x`, `y`.

c) Imprime `y`, `x`.

d) Imprime `x`.

e) Imprime `y`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    void y() {
        this.z();
    }
    static void z() {
        System.out.println("z");
    }
}
```

```
class A {
    public static void main(String[] args) {
        new A().x();
    }
    static void x() {
        new B().y();
    }
}
```

a) Não compila ao tentar invocar `y`.

b) Não compila ao tentar invocar `z`.

c) Não compila ao tentar invocar `x`.

d) Compila e imprime `z`.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    static void x() {
        System.out.println("x");
    }
    static void y() {
        System.out.println("y");
    }
}
class A extends B {
    public static void main(String[] args) {
        this.x();
        A.y();
    }
}
```

a) Não compila, erro ao invocar `x`.

b) Imprime `x`, `y`.

c) Não compila, erro ao invocar `y`.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    static void x() {
        System.out.println("x");
    }
    static void y() {
        System.out.println("y");
    }
}
class A extends B {
    public static void main(String[] args) {
        x();
        A.y();
    }
}
```

a) Não compila, erro ao invocar `x`.

b) Imprime `x`, `y`.

c) Não compila, erro ao invocar `y`.

6.3 CRIE MÉTODOS SOBRECARREGADOS

Um método pode ter o mesmo nome que outro, desde que a sua invocação não fique ambígua: os **argumentos** que são recebidos têm de ser obrigatoriamente diferentes, seja em quantidade ou em tipos, como mostra o código a seguir.

```
class Test {
    public void method(int i) {
        System.out.println("int");
    }

    protected void method(double x) {
        System.out.println("double");
    }
```

```
    public static void main(String[] args) {
        new Test().method(15); // int
        new Test().method(15.0); // double
    }
}
```

Já o código a seguir não compila:

```
class Test {
    public int method() {}
    protected double method() {} // compile error
}
```

Nesse exemplo, temos ambiguidade porque o tipo de retorno não é suficiente para distinguir os métodos durante a chamada. O Java decide qual das assinaturas de método sobrecarregado (*overloaded*) será usada em **tempo de compilação**.

Métodos sobrecarregados podem ter ou não um retorno diferente e uma visibilidade diferente. Mas eles não podem ter exatamente os mesmos tipos e quantidade de parâmetros. Nesse caso, seria uma sobrescrita de método.

No caso de sobrecarga com tipos que possuem polimorfismo, como em `Object` ou `String`, o compilador sempre invoca o método com o tipo mais específico (o menos genérico):

```
public class Test {
    void method(Object o) {
        System.out.println("object");
    }
    void method(String s) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Test().method("random"); // string
    }
}
```

Se quisermos forçar a invocação ao método mais genérico, devemos fazer o casting forçado:

```
public class Test {
    void method(Object o) {
        System.out.println("object");
    }
    void method(String s) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Test().method((Object) "random"); // object

        String x = "random";
        Object y = x;
        new Test().method(x); // string
        new Test().method(y); // object
    }
}
```

Um exemplo clássico é a troca de ordem, que é vista como sobrecarga, afinal, são dois métodos totalmente distintos:

```
public class Test {
    void method(String i, double x) {
        System.out.println(1);
    }
    void method(double x, String i) {
        System.out.println(2);
    }
    public static void main(String args[]) {
        new Test().method("guilherme", 33.0); // 1
        new Test().method(33.0, "guilherme"); // 2
    }
}
```

Porém, apesar de compiláveis, às vezes o compilador não sabe qual método deverá chamar. No caso a seguir, os números 2 e 3 podem ser considerados tanto `int` quanto `double`. Assim, o compilador fica perdido em qual dos dois métodos invocar, e

decide não compilar:

```
public class Test {
    void method(int i, double x) { // ok
    }
    void method(double x, int i) { // ok
    }

    public static void main(String[] args) {
        new Test().method(2.0, 3); // double, int
        new Test().method(2, 3.0); // int, double

        new Test().method(2, 3); // compile error
    }
}
```

Isso também ocorre com referências, que é diferente do caso com tipo mais específico. Aqui não há tipo mais específico, pois onde um é mais específico, o outro é mais genérico:

```
public class Xpto {
    void method(Object o, String s) {
        System.out.println("object");
    }
    void method(String s, Object o) {
        System.out.println("string");
    }

    public static void main(String[] args) {
        new Xpto().method("string", "string"); // compile error
    }
}
```

Note como ele é totalmente diferente do caso da regra do mais específico:

```
class Xpto2 {
    void method(Object o, Object o2) {
        System.out.println("object");
    }
    void method(String s, String s2) {
        System.out.println("string");
```

```
    }

    public static void main(String[] args) {
        new Xpto2().method("string", "string"); // string
    }
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int x = b(15);
        System.out.println(x);
        System.out.println(15);
        System.out.println(15.0);
    }
    static int b(int i) { return i; }
    static double b(int i) { return i; }
}
```

a) Não compila.

b) Compila e imprime `15`, `15`, `15`.

c) Compila e imprime `15`, `15`, `15.0`.

d) Compila e imprime `15`, `15.0`, `15.0`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int x = b(15);
        System.out.println(x);
        System.out.println(15);
        System.out.println(15.0);
    }
```

```
    static int b(int i) { return i; }
    static double b(double i) { return i; }
}
```

a) Não compila.

b) Compila e imprime `15 , 15 , 15 .`

d) Compila e imprime `15 , 15 , 15.0 .`

e) Compila e imprime `15 , 15.0 , 15.0 .`

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

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

a) Não compila.

b) Compila e imprime `"[]" .`

c) Compila e imprime `"..." .`

d) Compila e dá exception.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{}
class C{}
class D extends B{}
class A {
    int a(D d) { return 1; }
    int a(C c) { return 2; }
```

```
    int a(B b) { return 3; }
    int a(A a) { return 4; }
    public static void main(String[] args) {
        System.out.println(a(new D()));
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e imprime `3`.

e) Compila e imprime `4`.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{}
class C{}
class D extends B{}
class A {
    int a(D d) { return 1; }
    static int a(C c) { return 2; }
    static int a(B b) { return 3; }
    static int a(A a) { return 4; }
    public static void main(String[] args) {
        System.out.println(a(new D()));
    }
}
```

a) Compila e imprime `1`.

b) Compila e imprime `3`.

c) Compila e imprime `2`.

d) Não compila.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{}
class C{}
class D extends B{}
class A {
    static int a(D d) { return 1; }
    static int a(C c) { return 2; }
    static int a(B b) { return 3; }
    static int a(A a) { return 4; }
    public static void main(String[] args) {
        System.out.println(a(new D()));
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e imprime `3`.

e) Compila e imprime `4`.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{}
class C{}
class D extends B{}
class A {
    static int a(D d, B b) { return 1; }
    static int a(C c, C c) { return 2; }
    static int a(B b, B b) { return 3; }
    static int a(A a, A a) { return 4; }
    public static void main(String[] args) {
        System.out.println(a(new D(), new D()));
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e imprime `3`.

e) Compila e imprime `4`.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{}
class C{}
class D extends B{}
class A {
    static int a(D d, B b2) { return 1; }
    static int a(C c, C c2) { return 2; }
    static int a(B b, B b2) { return 3; }
    static int a(A a, A a2) { return 4; }
    public static void main(String[] args) {
        System.out.println(a(new D(), new D()));
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e imprime `3`.

e) Compila e imprime `4`.

6.4 DIFERENCIE O CONSTRUTOR PADRÃO E OS DEFINIDOS PELO USUÁRIO

Quando não escrevemos um construtor na nossa classe, o compilador nos dá um construtor padrão. Esse construtor, chamado de *default*, não recebe argumentos, tem a mesma visibilidade da classe e tem a chamada a `super()`.

Veja a classe a seguir:

```
class A {
}
```

Na verdade, ela acaba sendo:

```
class A {
    A() {
        super();
    }
}
```

Caso você adicione um construtor qualquer, o construtor `default` deixa de existir:

```
class A {}
class B {
    B(String s) {}
}
class Test {
    public static void main(String[] args) {
        new A(); // default constructor, ok
        new B(); // no default constructor, compile error
        new B("CDC"); // string constructor
    }
}
```

Dentro de um construtor, você pode acessar e atribuir valores aos atributos, suas variáveis membro:

```
class Test {
    int i;
    Test() {
        i = 15; // i = 15
        System.out.println(i); // 15
```

```
    }

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

Os valores inicializados com a declaração das variáveis são inicializados **antes** do construtor, justamente por isso o valor inicial de `i` é 0, o valor padrão de uma variável `int` membro.

```
class Test {
    int i;
    Test() {
        System.out.println(i); // default, 0
        i = 15; // i = 15
        System.out.println(i); // 15
    }

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

Vale lembrar de que variáveis membro são inicializadas automaticamente para: numéricas 0, `boolean` `false`, referências `null`.

Cuidado ao acessar métodos cujas variáveis ainda não foram inicializadas no construtor. O exemplo a seguir mostra um caso em que o método de inicialização é invocado antes de setar o valor da variável no construtor, o que causa um `NullPointerException`.

```
class A {

    int i = 15;
    String name;
    int length = getLength();
```

```
    A(String name) {
        this.name = name;
    }

    int getLength() {
        return name.length();
    }

    A() {
    }

}
```

Mesmo que inicializemos a variável fora do construtor, após a chamada do método pode ocorrer um erro, como no caso a seguir, de um outro `NullPointerException` :

```
class A {

    int i = 15;
    String name;
    int length = getLength();
    String lastname = "Silveira";

    A(String name) {
        this.name = name;
    }

    int getLength() {
        return lastname.length();
    }

    A() {
    }

}
```

Mudar a ordem da declaração das variáveis resolve o problema, uma vez que o método é agora invocado após a inicialização da variável `lastname` :

```
class A {
```

```
    int i = 15;
    String name;
    String lastname = "Silveira";
    int length = getLength();

    A(String name) {
        this.name = name;
    }

    int getLength() {
        return lastname.length();
    }

    A() {
    }

}
```

Cuidado ao invocar métodos no construtor e variáveis estarem nulas:

```
class Test {
    String name;
    Test() {
        tryLength(); // NullPointerException
        name = "guilherme";
    }

    private void tryLength() {
        System.out.println(name.length());
    }

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

E mais cuidado ainda caso isso ocorra por causa de sobrescrita de método, em que também poderemos ter essa `Exception`:

```
class Base {
    String name;
```

```
    Base() {
        test();
        name = "guilherme";
    }

    void test() {
        System.out.println("testing");
    }

}
class Test extends Base {
    void test() {
        System.out.println(name.length());//NullPointerException
    }
    public static void main(String[] args) {
        new Test();
    }
}
```

Já se o método `test` for privado, como o *binding* da chamada ao método é feito em compilação, o método invocado pelo construtor é o da classe mãe, sem dar a `Exception` :

```
class Base {
    String name;
    Base() {
        test();
        name = "guilherme";
    }

    private void test() {
        System.out.println("test");
    }

}
class Test extends Base {
    void test() {
        System.out.println(name.length());
    }
    public static void main(String[] args) {
        new Test();
    }
}
```

Você pode entrar em loop infinito, cuidado. Veja o caso a seguir, no qual o compilador não tem como detectar o loop, resultando em um `StackOverflow`:

```
class Test {
    Test() {
        new Test(); // StackOverflow
    }
    public static void main(String[] args) {
        new Test();
    }
}
```

Construtores podem ser de todos os tipos de modificadores de acesso: `private`, `protected`, `default` e `public`.

É comum criar um construtor privado e um método estático para criar seu objeto:

```
class Test {
    private Test() {
    }

    public static Test cria() {
        return new Test();
    }
}
```

Tenha muito cuidado com um método com nome do construtor. Se colocar um `void` na frente, vira um método:

```
class Test {
  void Test() {
    System.out.println("constructor?");
  }

  public static void main(String[] args) {
    new Test(); // default constructor...
  }
}
```

E podemos invocar o método:

```
class Test {
  void Test() {
    System.out.println("constructor?");
  }

  public static void main(String[] args) {
    new Test().Test(); // constructor?
  }
}
```

Existem também blocos de inicialização que não são cobrados na prova.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    final String n;
    A() {
        a();
        n = "learning";
    }
    void a() {
        System.out.println("test");
    }
}
class B extends A {
    void a() {
        System.out.println(n.length());
    }
    public static void main(String[] args) {
        new B();
    }
}
```

a) Não compila.

b) Compila e imprime `"test"` .

c) Compila e imprime `length` .

d) Compila e dá exception.

e) Compila e não imprime nada.

6.5 CRIE E SOBRECARREGUE CONSTRUTORES

Construtores também podem ser sobrecarregados:

```
class Test {
    public Test() {
    }
    public Test(int i) {
    }
}
```

Quando existem dois construtores na mesma classe, um construtor pode invocar o outro através da palavra chave `this` .

```
class Test {
    public Test() {
        System.out.println("simple");
    }
    public Test(int i) {
        this(); // simple
    }
}
```

Note que loops não compilam:

```
class Test {
    public Test(String s) {
        this(s, s);  // compile error, loop
    }
    public Test(String s, String s2) {
        this(s); // compile error, loop
```

```
    }
}
```

Temos de tomar cuidado com sobrecarga da mesma maneira que tomamos cuidado com sobrecarga de métodos: os construtores invocados seguem as mesmas regras que as de métodos.

Quando um método utiliza `varargs`, se ele possui uma variação do método sem nenhum argumento e invocarmos sem argumento, ele chamará o método sem argumentos (para manter compatibilidade com versões anteriores do Java). Por esse motivo, ao invocar o método `method` do código a seguir, será impresso `0 args`.

```
void desativa(Client... clients) {
    System.out.println("varargs");
}
void desativa() {
    System.out.println("0 args");
}
void method() {
    desativa(); // 0 args
}
```

A instrução `this` do construtor, caso presente, deve ser sempre a primeira dentro do construtor:

```
class Test {
    Test() {
        String value = "value...";
        this(value); // compile error
    }

    Test(String s) {
        System.out.println(s);
    }

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

```
    }
}
```

Justo por isso não é possível ter duas chamadas a `this`:

```
class Test {
    Test() {
        this(value);
        this(value); // compile error
    }

    Test(String s) {
        System.out.println(s);
    }

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

A instrução `this` pode envolver instruções:

```
class Test {
    Test() {
        this(value());
    }

    private static String value() {
        return "value...";
    }

    Test(String s) {
        System.out.println(s);
    }

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

A instrução não pode ser um método da própria classe, pois o objeto não foi construído ainda:

```
class Test {
```

```
    Test() {
        this(value()); // instance method, compile error
    }

    private String value() {
        return "value...";
    }

    Test(String s) {
        System.out.println(s);
    }

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

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B { B() { this(1); } B(int i) { this(); } }
class A {
    public static void main(String[] args) {
        new B();
    }
}
```

a) Não compila.

b) Compila e joga exception.

c) Compila e não imprime nada.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B() { B(A a) {} B() {} }
class C() { C(B b) {} C() {} }
class A {
```

```
    public static void main(String[] args) {
        new A(); new B(); new C();
    }
}
```

a) Não compila ao invocar o construtor de `A`.

b) Não compila ao invocar o construtor de `B`.

c) Não compila ao invocar o construtor de `C`.

d) Não compila na definição das classes `B` e `C`.

e) Compila e joga exception.

f) Compila e não imprime nada.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B { B(A a) {} B() {} }
class C { C(B b) {} C() {} }
class A {
    public static void main(String[] args) {
        new A(); new B(); new C();
    }
}
```

a) Não compila ao invocar o construtor de `A`.

b) Não compila ao invocar o construtor de `B`.

c) Não compila ao invocar o construtor de `C`.

d) Não compila na definição das classes `B` e `C`.

e) Compila e joga exception.

f) Compila e não imprime nada.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B { B(A a) {} B() {} }
class C { C(B b) {} C() {} }
class A {
    public static void main(String[] args) {
        new C(new B(new A()));
    }
}
```

a) Não compila ao invocar o construtor de `A`.

b) Não compila ao invocar o construtor de `B`.

c) Não compila ao invocar o construtor de `C`.

d) Não compila na definição das classes `B` e `C`.

e) Compila e joga exception.

f) Compila e não imprime nada.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B { B(A a) {new C(); } B() { new C(this);} }
class C { C(B b) {new B(new A());} C() {new B();} }
class A {
    public static void main(String[] args) {
        new C(new B(new A()));
    }
}
```

a) Não compila ao invocar o construtor de `A`.

b) Não compila ao invocar o construtor de `B`.

c) Não compila ao invocar o construtor de `C`.

d) Não compila na definição das classes `B` e `C`.

e) Compila e joga exception.

f) Compila e não imprime nada.

6.6 APLIQUE MODIFICADORES DE ACESSO

Os modificadores de acesso, ou modificadores de visibilidade, servem para definir quais partes de cada classe (ou se uma classe inteira) estão visíveis para serem utilizadas por outras classes do sistema. Só é permitido usar um único modificador de acesso por vez:

```
private public int x; // compile error
```

O Java possui quatro modificadores de acesso:

- `public`;
- `protected`;
- Nenhum modificador, chamado de `default` ou `private protected`;
- `private`.

Classes e interfaces cobradas na prova só aceitam os modificadores `public` ou `default`.

Membros (construtores, métodos e variáveis) podem receber qualquer um dos quatro modificadores. Variáveis locais (declaradas dentro do corpo de um método ou construtor) e parâmetros não podem receber nenhum modificador de acesso, mas podem receber outros modificadores.

Top Level Classes e Inner Classes

Classes internas (*nested classes* ou *inner classes*) são classes que são declaradas dentro de outras classes. Esse tipo de classe pode ser definida com qualquer um dos quatro modificadores de acesso, já que são consideradas membros da classe onde foram declaradas (*top level class*).

Nesta certificação, não são cobradas classes internas, apenas *top level classes*.

Para entender como os modificadores funcionam, vamos imaginar as seguintes classes, todas definidas nos diretórios correspondentes ao seus pacotes. A primeira classe representa uma forma geométrica (*Shape*) qualquer:

```
package shape;

class Shape{
    double side;
    double getArea(){
        return 0;
    }
}
```

A segunda representa uma forma específica, um quadrado:

```
package shape;

class Square extends Shape{}
```

Por último, o triângulo é representado pela classe `Triangle`:

```
package shape.another;
import shape.*;
```

```
class Triangle extends Shape{}
```

Public

O modificador `public` é o menos restritivo de todos. Classes, interfaces e membros marcados com esse modificador podem ser acessados de qualquer componente, em qualquer pacote. Vamos alterar nossa classe `Shape`, marcando-a e todos seus membros com o modificador `public`:

```
package shape;

public class Shape{
    public double side;
    public double getArea(){
        return 0;
    }
}
```

Agora vamos fazer um teste:

```
package shape.another;
import shape.*;

public class TestAnotherPackage{

    public static void main(String... args){
        Shape f = new Shape(); // public, ok
        f.side = 5.5; // public, ok
        f.getArea(); // public, ok
    }
}
```

Repare que, mesmo nossa classe `TestAnotherPackage` estando em um pacote diferente da classe `Shape`, é possível acessar a classe e todos os membros declarados como `public`.

Desenvolvedores experientes costumam julgar boa prática

fugir do uso de qualquer coisa pública. Assim que algo é definido dessa maneira, não sabemos mais quem, como e quando essas variáveis, classes e métodos estão sendo usados, o que dificulta a manutenção do código a médio e longo prazo. Como regra geral, referências `static public` são consideradas perigosas quando levamos em conta a manutenção do código.

Protected

Membros definidos com o modificador `protected` podem ser acessados por classes e interfaces no mesmo pacote, e por qualquer classe que estenda aquela onde o membro foi definido, independente do pacote.

Vamos modificar nossa classe `Shape` para entendermos melhor:

```
package shape;

public class Shape{
    protected double side; // protected
    public double getArea(){}
}
```

Com o modificador `protected`, nossa classe de testes em outro pacote não compila mais:

```
package shape.another;
import shape.*;

public class TestAnotherPackage{

    public static void main(String... args){
        Shape f = new Shape();
        f.side = 5.5; // protected, compile error
        f.getArea();
    }
}
```

Se criarmos uma nova classe de teste no pacote `shape`, conseguimos acessar novamente o atributo, pois estamos no mesmo pacote tentando acessar um membro `protected`:

```
package shape;

public class Test{

    public static void main(String... args){
        Shape f = new Shape();
        f.side = 5.5; // same package, protected, ok
    }
}
```

Embora esteja em um pacote diferente, a classe `Triangle` consegue acessar o atributo `side`, já que ela estende da classe `Shape`. Classes filhas podem acessar os membros `protected` de sua classe mãe (avó etc).

```
package shape.another;
import shape.*;

class Triangle extends Shape{

    public void printSide(){
        System.out.println("Side=" + side); // ok
    }
}
```

Agora repare que, se efetuarmos o casting do objeto atual para uma `Shape`, não podemos acessar seu lado (`side`):

```
package another;
import shape.*;

class Triangle extends Shape{

    public void printSide(){
        System.out.println("Side=" + side); // ok

        // compile error
```

```
        System.out.println("Side=" + ((Shape) this).side);
    }
}
```

Isso ocorre porque estamos dizendo que queremos acessar a variável membro `side` de um objeto através de uma referência para este objeto, e não diretamente. Diretamente seria o uso puro do `this` ou nada. Nesse caso, após usar o `this`, usamos um casting, o que deixa o compilador perdido.

A reformulação do exemplo anterior mostra novamente o mesmo problema. Dentro da classe `Triangle`, podemos acessar membros `protected` de `Shape` somente se for de nós mesmos (`this` ou diretamente), mas não através de uma outra referência (`myself`).

```
package another;
import shape.*;

class Triangle extends Shape{

    public void printSide(){
        System.out.println("Side=" + this.side); // ok

        Shape myself = ((Shape) this);
        // compile error
        System.out.println("Side=" + myself.side);
    }
}
```

Default

Se não definirmos explicitamente qual o modificador de acesso, podemos dizer que aquele membro está usando o modificador `default`, também chamado de `package private`. Neste caso, os membros da classe só serão visíveis dentro do mesmo pacote, uma restrição mais rígida do que o uso do

modificador `protected` ou `public`:

```
package shape;

public class Shape{
    protected double side;
    public double getArea(){
        return 0;
    }
    double getPerimeter(){ // default access
        return 0;
    }
}
```

O método `getPerimeter()` só será visível para todas as classes do pacote `shape`. Agora a classe `Triangle` — que, apesar de herdar de `Shape`, está em outro pacote — não consegue mais ver o método.

```
package another;
import shape.*;

class Triangle extends Shape{

    public void printPerimeter(){
        // compile error
        System.out.println("Perimeter=" + getPerimeter());
    }
}
```

Palavra-chave default

Lembre-se! A palavra-chave `default` é usada para definir a opção padrão em um bloco `switch`, ou para definir um valor inicial em uma *Annotation*. No Java 8, ela pode ser usada para definir a implementação padrão de um método em uma interface.

Usá-la em uma declaração de classe ou membro é inválido e causa um erro de compilação:

```
default class Ball { // compile error
    default String color; // compile error
}
```

Mas e se declararmos uma classe com o modificador `default`? Isso vai fazer com que aquela classe só seja visível dentro do pacote onde foi declarada. Não importa quais modificadores os membros dessa classe tenham, se a própria classe não é visível fora de seu pacote, nenhum de seus membros é visível também.

Veja a classe `Square`, que está definida com o modificador `default`:

```
package shape;

class Square extends Shape{}
```

Olhe agora o seguinte código, onde definimos a classe `TestAnotherPackage`. Perceba que não é possível usar a classe `Square`, mesmo importando todas as classes do pacote `shape`:

```
package shape.another;
import shape.*;

public class TestAnotherPackage{

    public static void main(String... args){
        Square q = new Square(); // compile error
    }
}
```

Linha com erro de compilação

Eventualmente, na prova, é perguntado em quais linhas ocorreram os erros de compilação. É bem importante prestar atenção nesse detalhe.

Por exemplo, no exemplo a seguir, o erro sempre acontecerá quando tentarmos acessar a classe `Square`, que não é visível fora de seu pacote. O import de todas as classes públicas do pacote `shape` é feita sem problemas.

```
package shape.another;

import shape.*;  // ok

public class TestAnotherPackage{

    public static void main(String... args){
        Square q = new Square();  // compile error
    }
}
```

O mesmo código pode apresentar erro em uma linha diferente, apenas mudando o `import`. Repare que o código a seguir dá erro nas duas linhas, tanto do `import` quanto na tentativa de uso:

```
package shape.another;
```

```
import shape.Square; // compile error

public class TestAnotherPackage{

    public static void main(String... args){
        Square q = new Square(); // compile error
    }
}
```

Private

O `private` é o mais restritivo de todos os modificadores de acesso. Membros definidos como `private` só podem ser acessados de dentro da classe e de nenhum outro lugar, independente de pacote ou herança.

No exemplo a seguir, demonstramos o uso do `private` com a variável membro `color`, que só poderá ser acessada por membros dentro da classe `Shape`. Nem `Square` nem `Triangle` poderão acessar o membro `color` por estarem fora da classe `Shape`: um está até mesmo fora do arquivo, outro fora do pacote.

```
package shape;

public class Shape{
    protected double side;
    public double getArea(){}

    private String color;

}
```

PRIVATE E CLASSES ANINHADAS OU ANÔNIMAS

Classes aninhadas ou anônimas podem acessar membros privados da classe onde estão contidas. Na certificação, tais classes não são cobradas.

Métodos privados e padrão não podem ser sobrescritos. Se uma classe o "sobrescreve", ele simplesmente é um método novo, portanto, não podemos dizer que é sobrescrita. Veremos isso mais a fundo na seção sobre sobrescrita (seção *Implementando herança*).

Resumo das regras de visibilidade

Todos os membros da classe com o modificador de `private` só podem ser acessados de dentro dela mesma. Todos os membros da classe sem nenhum modificador de visibilidade (ou seja, com visibilidade **package-private**) podem ser acessados de dentro da própria classe, ou de dentro de qualquer outra classe, interface ou enum do mesmo pacote.

Todos os membros da classe com o modificador `protected` podem ser acessados:

- De dentro da classe, ou de dentro de qualquer outra classe, interface ou enum do mesmo pacote;
- De dentro de alguma classe que deriva direta ou indiretamente da classe, independente do pacote. O membro `protected` só pode ser chamado através da referência `this` , ou por uma referência que seja dessa

classe filha.

Todos os membros da classe com o modificador `public` podem ser acessados de qualquer lugar da aplicação. E não podemos ter classes/ interfaces/ enums top-level como `private` ou `protected`.

Uma classe é dita *top-level* se ela não foi definida dentro de outra classe, interface ou enum. Analogamente, são definidas as interfaces top-level e os enums top-level.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o `Test`, com os arquivos a seguir nos diretórios adequados:

```
import model.Client;
class Test {
    public static void main(String[] args) {
        new Client("guilherme").print();
    }
}
```

```
package model;

public class Client {
    private String name;
    Client(String name) {
        this.name = name;
    }
    public void print() {
        System.out.println(name);
    }
}
```

a) Não compila: erro na classe `Teste`.

b) Não compila: erro na classe `Client`.

c) Erro de execução: método `main`.

d) Roda e imprime `"Guilherme"`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    private static int a(int b) {
        return b(b)-1;
    }
    private static int b(int b) {
        return b-1;
    }
    public static void main(String[] args) {
        System.out.println(new A().a(5));
    }
}
```

a) Não compila

b) Compila e imprime 3.

c) Compila e dá erro de execução.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    private public int a(int b) {
        return b(b)-1;
    }
    private static int b(int b) {
        return b-1;
    }
    public static void main(String[] args) {
        System.out.println(new A().a(5));
    }
}
```

a) Não compila nas invocações de métodos.

b) Não compila na declaração de variáveis e métodos.

c) Compila e imprime 3.

d) Compila e dá erro.

4) Escolha a opção adequada ao tentar compilar e rodar os arquivos a seguir, cada um em seu diretório adequado:

```
package a;
import b.*;
public class A extends B { protected int a(String s)
                           {return 2;} }
```

```
package b;
import a.*;
public class B { public int a(Object s) {return 1;} }
```

```
import a.*;
import b.*;
class A {
    public static void main(String[] args) {
        System.out.println(new A().a("a"));
    }
}
```

a) Não compila.

b) Imprime 1.

c) Imprime 2.

d) Erro em execução.

5) Escolha a opção adequada ao tentar compilar e rodar os arquivos a seguir, cada um em seu diretório adequado:

```
package a;
import b.*;
public class A extends B { protected int a(String s)
                           {return 2;} }
```

```
package b;
import a.*;
public class B { public int a(Object s) {return 1;} }
```

```
import a.*;
import b.*;
class C {
    public static void main(String[] args) {
        System.out.println(new A().a("a"));
    }
}
```

a) Não compila.

b) Imprime 1.

c) Imprime 2.

d) Erro em execução.

6) Escolha a opção adequada ao tentar compilar e rodar os arquivos a seguir, cada um em seu diretório adequado:

```
package a;
import b.*;
public class A extends B { protected int a(String s)
                           {return 2;} }
```

```
package b;
import a.*;
public class B { default int a(Object s) {return 1;} }
```

```
import a.*;
import b.*;
class C {
    public static void main(String[] args) {
        System.out.println(new A().a("a"));
    }
}
```

a) Não compila.

b) Imprime 1.

c) Imprime 2.

d) Erro em execução.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    static int bs=0;
    final int b = ++bs;
    private B() {}
    static B b() { return new B(); }
}
class A {
    public static void main(String[] args) {
        System.out.println(B.b().b);
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 0.

d) Compila e dá erro de execução.

6.7 APLIQUE PRINCÍPIOS DE ENCAPSULAMENTO A UMA CLASSE

A **assinatura** de um método é o que realmente deve importar para o usuário de alguma classe. Segundo os bons princípios do encapsulamento, a implementação dos métodos deve estar *encapsulada* e não deve fazer diferença para o usuário.

O que é importante em uma classe é **o que ela faz** e não **como ela faz**. *O que ela faz* é definido pelos comportamentos expostos,

ou seja, pelos métodos e suas assinaturas.

O conjunto de assinaturas de métodos visíveis de uma classe é chamado de **interface de uso**. É através dessas operações que os usuários vão se comunicar com os objetos dessa classe.

Mantendo os detalhes de implementação de nossas classes "escondidos", evitamos que mudanças na forma de implementar uma lógica quebre vários pontos de nossa aplicação.

Uma das formas mais simples de começar a encapsular o comportamento de uma classe é escondendo seus atributos. Podemos fazer isso facilmente usando a palavra-chave `private` :

```
public class Person{
    private String name;
}
```

Caso precisemos acessar um desses atributos a partir de outra classes, teremos de criar um método para liberar o acesso de leitura desse atributo. Seguindo a especificação dos *javabeans*, esse método seria um *getter*. Da mesma forma, se precisarmos liberar a escrita de algum atributo, criamos um método *setter*:

```
    public class Person{
        private String name;

        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name = name;
        }
    }
```

Com essa abordagem, poderíamos fazer uma validação em nossos métodos, para evitar que nossos atributos fiquem com

estado inválido. Por exemplo, podemos verificar se o nome possui pelo menos 3 caracteres:

```
public class Person{
    private String name;
    private String lastname;

    public String getName() {
        return name;
    }
    public void setName(String name) {
        if(name != null && name.trim().length() >= 3)
            this.name = name;
        else{
            throw new IllegalArgumentException(
             "At least 3 chars");
        }
    }
}
```

Encapsulamento é muito mais do que atributos privados, *getters* e *setters*. Não é nosso foco aqui discutir boas práticas de programação, e sim o conhecimento necessário para passar na prova. Em questões sobre encapsulamento, sempre fique atento à alternativa que esconde mais detalhes de implementação da classe analisada, ou aquela que evita modificar valores de nosso objeto sem antes pedir permissão a ele.

A prova pode utilizar tanto o termo *encapsulation* como *information hiding* para falar sobre encapsulamento (ou esconder informações).

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
```

```
    private int b;
    public int getB() { return b; }
    public void setB(int b) { this.b= b; }
}
class A {
    public static void main(String[] args) {
        new B().setB(5);
        System.out.println(new B().getB());
    }
}
```

a) Não compila.

b) Compila e imprime 0.

c) Compila e imprime 5.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    private int b;
    public int getB() { return b; }
    public void setB(int b) { this.b= b; }
}
class A {
    public static void main(String[] args) {
        B b = new B();
        b.setB(5);
        System.out.println(b.getB());
    }
}
```

a) Não compila.

b) Compila e imprime 0.

c) Compila e imprime 5.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    private int b;
    public int getB() { return b; }
    public void setB(int b) { b= b; }
}
class A {
    public static void main(String[] args) {
        B b = new B();
        b.setB(5);
        System.out.println(b.getB());
    }
}
```

a) Não compila.

b) Compila e imprime 0.

c) Compila e imprime 5.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B{
    int b;
    public void setB(int b) { b= b; }
}
class A {
    public static void main(String[] args) {
        B b = new B();
        b.setB(5);
        System.out.println(b.b);
    }
}
```

a) Não compila, pois não é possível ter `setter` sem `getter`.

b) Compila e imprime 0.

c) Compila e imprime 5.

5) Escolha a opção adequada ao tentar compilar e rodar o

arquivo a seguir:

```
class B{
    private final int b;
    B(int b) { this.b = b;}
    public int getB() { return b; }
    public void setB(int b) { b= b; }
}
class A {
    public static void main(String[] args) {
        B b = new B(10);
        b.setB(5);
        System.out.println(b.getB());
    }
}
```

a) Não compila.

b) Compila e imprime 0.

c) Compila e imprime 5.

d) Compila e imprime 10.

6.8 EFEITOS EM REFERÊNCIAS A OBJETOS E A TIPOS PRIMITIVOS

As informações que queremos enviar para um método devem ser passadas como parâmetro. O domínio de como funciona a passagem de parâmetro é fundamental para a prova de certificação.

O requisito para entender passagem de parâmetro no Java é saber como funciona a pilha de execução e o *heap* de objetos. A pilha de execução é o "lugar" onde são empilhados os métodos invocados na mesma ordem em que foram chamados.

O *heap* é o "lugar" no qual são guardados os objetos criados

durante a execução. Considere o exemplo a seguir:

```
class Test {
    public static void main(String[] args) {
        int i = 2;
        test(i);
    }

    private static void test(int i) {
        for (int j = 0; j < i; j++) {
            new String("j = " + j);
        }
    }
}
```

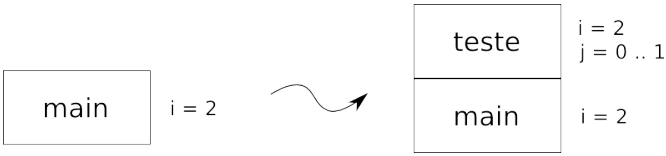


Figura 6.1: Pilha de execução

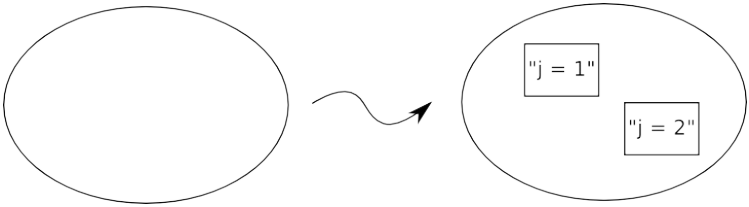


Figura 6.2: Heap

A passagem de parâmetros é feita por cópia de valores. Dessa forma, mudanças nos valores das variáveis definidas na lista de parâmetros de um método não afetam variáveis de outros métodos, mas isso tem algumas implicações importantíssimas com as quais devemos tomar muito cuidado.

Passagem de parâmetros primitivos

Veja o seguinte código:

```
class Test {
    public static void main(String[] args) {
        int i = 2;
        test(i);
        System.out.println(i);
    }

    static void test(int i) {
        i = 3;
    }
}
```

Ao executar a classe `Teste`, será impresso o valor `2`. É necessário perceber que as duas variáveis com o nome `i` estão em métodos diferentes. Há um `i` no `main()` e outro `i` no `teste()`. Alterações em uma das variáveis não afetam o valor da outra.

Passagem de parâmetros de referência

Agora veja esta classe:

```
class Test {
    public static void main(String[] args) {
        Exam exam = new Exam();
        exam.timeLimit = 100;
        test(exam);
        System.out.println(exam.timeLimit);
    }

    static void test(Exam exam) {
        exam.timeLimit = 210;
    }
}

class Exam {
    double timeLimit;
}
```

Esse exemplo é bem interessante e causa muita confusão. O que será impresso na saída, ao executar a classe `Test`, é o valor `210`. Os dois métodos têm variáveis com o mesmo nome (`exam`). Essas variáveis são realmente independentes, ou seja, mudar o valor de uma não afeta o valor da outra.

Por outro lado, como são variáveis não primitivas, elas guardam referências e, neste caso, são referências que apontam para o mesmo objeto. Modificações nesse objeto podem ser executadas através de ambas as referências. Note que só existe um único objeto que foi instanciado, as duas referências possuem um valor que referencia esse mesmo objeto.

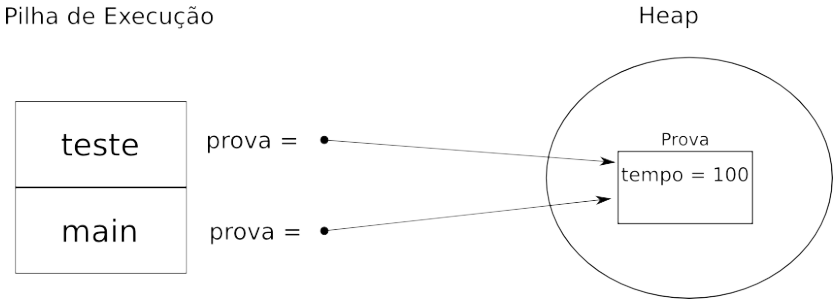


Figura 6.3: Passagem de parâmetros não primitivos

Portanto, se eu trocar a referência, só trocarei a referência da variável local, e não a referência original do método que nos invocou, como no exemplo do `test2`, em que estamos trocando somente a referência local e não a original:

```
class Exam {
    int timeLimit;
}
class TestReferenceAndPrimitive {
    public static void main(String[] args) {
        Exam exam = new Exam();
```

```
        exam.timeLimit = 100;
        test(exam);
        System.out.println(exam.timeLimit);

        test2(exam);
        System.out.println(exam.timeLimit);

        int i = 2;
        i = test(i);
        System.out.println(i);
    }
    static void test2(Exam exam) {
        exam = new Exam();
        exam.timeLimit = 520;
    }

    static void test(Exam exam) {
        exam.timeLimit = 210;
    }

    static int test(int i) {
        i = 5;
        System.out.println(i);
        return i;
    }
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int i = 150;
        i = ++s(i);
        System.out.println(i);
    }
    static int s(int i) {
        return ++i;
    }
}
```

a) Não compila.

b) Compila e imprime 150.

c) Compila e imprime 151.

d) Compila e imprime 152.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        int[] i = {150, 151};
        i = s(i);
        System.out.println(i[1]);
    }
    static int[] s(int[] i) {
        int[] j = {i[0], i[1]};
        i[1]++;
        return j;
    }
}
```

a) Não compila.

b) Compila e imprime 150.

c) Compila e imprime 151.

d) Compila e imprime 152.

e) Compila e imprime 153.

CAPÍTULO 7

TRABALHANDO COM HERANÇA

7.1 IMPLEMENTANDO HERANÇA

Em Java, podemos usar herança simples entre classes com o `extends` . A nomenclatura usada é de **classe mãe** (*parent class*) e **classe filha** (*child class*), ou **superclasse** e **subclasse**.

Herança entre classes permite que um código seja reaproveitado, de maneira que a classe filha reutilize o código da parte mãe. A classe filha especializa a classe mais genérica, e ela é uma especialização de uma superclasse.

Herança em Java pode ser entre classes, reaproveitando membros, ou herança de uma interface, com a qual reaproveitamos interfaces de métodos. O exemplo a seguir ilustra uma hierarquia de tipos que utiliza herança em três classes.

```
class Parent {
}
class Child extends Parent {
}
class Grandchild extends Child {
}
```

Vale lembrar de que toda classe que não define de quem está

herdando herda de `Object`:

```
class Explicit extends Object {
}
class Implicit {
    // extends Object
}
```

Mesmo que uma classe herde explicitamente de outra, indiretamente ela herdará de `Object`:

```
class ImplicitChild extends Implicit {
    // Implicit extends Object
}
```

Mas não podemos herdar de duas classes explicitamente:

```
class Simple1 {}
class Simple2 {}
class Complex extends Simple1, Simple2 { // compile error
}
```

Para podermos herdar de uma classe, a classe mãe precisa ser visível pela classe filha e, pelo menos, um de seus construtores também. O exemplo a seguir não compila, pois existe um construtor padrão (*default constructor*) que chama o construtor sem argumentos da classe pai, que não existe:

```
class Parent {
    Parent(int x) {
    }
}
class Child1 extends Parent{ // compile error
    // implicit Child1() { super(); }
}
```

Já o código a seguir compila:

```
class Parent {
    Parent(int x) {
    }
```

```
}
class Child2 extends Parent{
    Child2() {
        super(15); // ok
    }
}
```

Além disso, a classe mãe não pode ser `final`:

```
final class Parent {
}
class Child extends Parent { // compile error
}
```

Mas uma classe filha pode ser `final`:

```
class Parent {
}
final class Child extends Parent {
    // ok
}
```

Herança de métodos e atributos

Todos os métodos e atributos de uma classe mãe são herdados, independente das visibilidades.

```
class X {
    int a;
    public void b() {
    }
}
class Y extends X {
    // a + b
}
```

Dependendo da visibilidade e das classes envolvidas, a classe filha não consegue enxergar o membro herdado. No exemplo a seguir, herdamos o atributo, mas não o enxergamos diretamente.

```
class X {
```

```
    private int x;

    public void setX(int x) {
        this.x = x;
    }

    public int getX() {
        return x;
    }
}

class Y extends X {
    public void method () {
        this.x = 5; // compile error: "x has private access in X"

        this.setX(10); // ok
    }
}
```

Métodos estáticos e herança

Não existe herança de métodos estáticos. Mas quando herdamos de uma classe com métodos estáticos, podemos chamar o método da classe mãe usando o nome da filha (embora não seja uma boa prática):

```
class W {
    static void method() {
        System.out.println("w");
    }
}

class Z extends W {
}

class Test {
    public static void main(String[] args) {
        W.method(); // w
        Z.method(); // w
    }
}
```

Já o uso do `super` não compila, pois no contexto estático não existe objeto onde o método está sendo chamado:

```
class W {
    public static void method() {
        System.out.println("w");
    }
}
class Z extends W {
    public static void method() {
        super.method();  // compile error
    }
}
```

Por não existir herança, o modificador `abstract` não é aceito em métodos estáticos como ilustra o exemplo a seguir:

```
class StaticRunner {
    abstract static void run(); // compile error
}
```

Podemos até escrever na subclasse um método estático de mesmo nome, mas isso **não é sobrescrita** (alguns chamam de *redefinição de um método*):

```
class W {
    public static void method() {
        System.out.println("w");
    }
}
class Z extends W {
    public static void method() {
        System.out.println("z");
    }
}
public class Test {
    public static void main(String[] args) {
        System.out.println(W.method()); // w
        System.out.println(Z.method()); // z
    }
}
```

Na verdade, você até segue as regras de sobrescrita de método (visibilidade e retorno), mas no polimorfismo ele não funciona como métodos de instância. O compilador define qual método será invocado em tempo de compilação, ignorando completamente qual o tipo do objeto referenciado em tempo de execução.

O código a seguir mostra um exemplo no qual o *binding* do polimorfismo seria feito em tempo de execução se o método fosse de instância. Entretanto, como ele é estático, o *binding* é feito em compilação e o método invocado é o da classe `W`.

```
class W {
    public static void method() {
        System.out.println("w");
    }
}
class Z extends W {
    public static void method() {
        System.out.println("z");
    }
}
public class Test {
    public static void main(String[] args) {
        W w = new W();
        w.method(); // w

        Z z = new Z();
        z.method(); // z

        W zPolimorphedAsW = z;
        zPolimorphedAsW.method(); // w
    }
}
```

Construtores e herança

Não existe herança de construtores. O que existe é a classe filha chamar o construtor da mãe.

Sobrescrita de atributos

Não existe sobrescrita de atributos. Podemos, sim, ter um atributo na classe filha com mesmo nome da mãe, mas não chamamos de sobrescrita. Nesses casos, o objeto vai ter 2 atributos diferentes: um da mãe (acessível com `super`) e um na filha (acessível com `this`).

Object

Em Java, toda classe é obrigada a usar a herança. Quando não escrevemos `extends` explicitamente, estamos herdando de `java.lang.Object` automaticamente. Isso quer dizer que todo objeto em Java **é um** `Object` e, portanto, herda todos os métodos da classe `Object` (por isso esses são muito importantes).

Há vários métodos em `Object`, mas o mais simples talvez seja o `toString`, que podemos sobrescrever em nossas classes para devolver alguma `String` que represente o objeto:

```
class Car {
    String color;
```

```
    public String toString() {
        return "Color = " + this.color;
    }
}
```

Precisamos lembrar de que o `toString` é chamado automaticamente para nós quando usamos o objeto no contexto de `String`:

```
    Car c = new Car();
    c.color = "green";

    System.out.println(c); // toString => Color = green

    String s = "msg: " + c; // toString => msg: Color = green
    System.out.println(s);
```

Exercícios

1) O código compila?

```
class A {
    public void method(long l) {
    }
}
class B extends A{
    protected void method(int i) {
    }
}
```

2) O código compila?

```
    import java.io.*;
    class Vehicle {
        protected void turnon() throws IOException {}
    }
    class Car extends Vehicle {
        public void turnon() throws FileNotFoundException {}
    }
```

3) Escolha a opção adequada ao tentar compilar e rodar o

arquivo a seguir:

```
class B extends C { int m(int a) { return 1; } }
class C extends A { int m(double b) { return 3; } }
class A extends B {
    int m(String c) { return 3; }
    public static void main(String[] args) {
        System.out.println(new C().m(3));
    }
}
```

a) O código não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

e) Compila e imprime 1, 2, 3.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B { int m(int a) { return 1; } }
class C { int m(double b) { return 2; } }
class A extends B, C{
    public static void main(String[] args) {
        System.out.println(new C().m(3));
    }
}
```

a) O código não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 1, 2.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B { private B() {} static B B(String s)
          { return new B(); } }
class A {
    public static void main(String[] args) {
        B b = B.B("t");
    }
}
```

a) Não compila.

b) Compila e imprime `"t"`.

c) Compila e não imprime nada.

d) Compila e joga uma exception.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B { private B() {} static B B(String s)
          { return new B(); } }
class A extends B {
    public static void main(String[] args) {
        B b = B.B("t");
    }
}
```

a) Não compila.

b) Compila e imprime `"t"`.

c) Compila e não imprime nada.

d) Compila e joga uma exception.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    private String s;
    protected B() {}
    static A B(String s) {
        return new A();
    }
}
class A extends B {
    A(String s) {
        this.s = s;
    }
    public static void main(String[] args) {
        B b = A.B("t");
        System.out.println(b.s);
    }
}
```

a) Não compila.

b) Compila e imprime `"t"` .

c) Compila e não imprime nada.

d) Compila e joga uma exception.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    protected String s;
    protected B() {}
    static A B(String s) {
        return new A(s);
    }
}
class A extends B {
    A(String s) {
        this.s = s;
    }
    public static void main(String[] args) {
        A b = A.B("t");
        System.out.println(b.s);
    }
```

```
}
```

a) Não compila.

b) Compila e imprime `"t"`.

c) Compila e não imprime nada.

d) Compila e joga uma exception.

7.2 DESENVOLVA CÓDIGO QUE MOSTRA O USO DE POLIMORFISMO

Reescrita ou sobrescrita é a maneira como uma subclasse pode *redefinir* o comportamento de um método que foi herdado de uma das suas superclasses (direta ou indiretamente). O exemplo a seguir mostra a classe `Car` sobrescrevendo o método `turnon`:

```
class Vehicle {
    public void turnon() {
        System.out.println("Vehicle running!");
    }
}

class Car extends Vehicle {
    public void turnon() {
        System.out.println("Car running!");
    }
}
```

Agora, considere invocar o método `turnon` de um objeto do tipo `Car` de duas maneiras distintas:

```
public class Test{
    public static void main(String [] args){
        Car c = new Car();
        c.turnon(); // Car running!

        Vehicle v = new Car();
```

```
        v.turnon(); // Car running!
    }
}
```

O método chamado aqui será o da classe `Car`, independente de a referência ser do tipo `Vehicle` (o que importa é o objeto).

Será descoberto qual método será executado (*binding*) em **tempo de execução** (em *compile-time*, a *assinatura* é decidida em tempo de compilação!). E isso é a chamada virtual de método (*virtual method invocation*).

Para reescrever um método, é necessário:

- Exatamente o mesmo nome;
- Os parâmetros têm de serem iguais em tipo e ordem (nomes das variáveis podem mudar);
- Retorno do método deve ser igual ou mais específico que o da mãe;
- Visibilidade deve ser igual ou maior que o da mãe;
- Exceptions lançadas devem ser iguais ou menos que na mãe;
- Método na mãe não pode ser `final`.

Se essas regras não forem respeitadas, pode haver um erro de compilação, ou o método declarado não será considerado uma reescrita do método.

A regra sobre visibilidade é: um método reescrito só pode ter visibilidade maior ou igual à do método que está sendo reescrito. Essa não é uma regra mágica! Faz todo o sentido; pense um pouco

sobre o que poderia acontecer se ela não existisse.

O código a seguir não compila, pois `ligar` é público na classe mãe, então só pode ser reescrito com visibilidade pública:

```
class Vehicle {
    public void turnon() {
        System.out.println("Vehicle running!");
    }
}

class Car extends Vehicle {
    protected void turnon() { // compile error
        System.out.println("Car running!");
    }
}
```

Muito cuidado com interfaces, pois a definição de um método é, por padrão, `public`, e o exercício pode apresentar uma pegadinha de compilação:

```
interface A {
    void a();
}
class B implements A {
    void a() { // compile error
    }
}
class C implements A {
    public void a() { // ok
    }
}
```

Estranhamente, um método sobrescrito pode ser abstrato, dizendo para o compilador que quem herdar dessa classe terá de sobrescrever o método original:

```
class A {
    void a() {
    }
}
```

```
abstract class B extends A {
    abstract void a(); // ok
}
class C extends B{
    // compile error
}
class D extends B{
    void a() {
        // ok
    }
}
```

O retorno de um método pode ser **covariante**. O compilador permite que a classe filha tenha um retorno igual ou mais específico polimorficamente (um subtipo).

Cuidado! O retorno covariante não vale para tipos primitivos. Um exemplo de retorno covariante válido é retornar uma `ArrayList` (subtipo) quando a definição na superclasse definia uma `List` (interface implementada por `ArrayList`):

```
class A {
    List<String> metodo () {
    }
}

class B extends A {
    ArrayList<String> method() { // ok
    }
}
```

Outra regra importante sobre reescrita é a assinatura em relação ao *lançamento de exceções* (`throws`). Um método reescrito só pode lançar as mesmas exceções *checked* ou menos que o método que está sendo reescrito (quanto às *unchecked*, não há regras e sempre podemos lançar quantas quisermos).

```
import java.sql.SQLException;
import java.io.IOException;
```

```
class A {
    public void method() throws SQLException, IOException {
    }
}

class B extends A {
    public void method() throws IOException { // ok
    }
}
```

Esse código compila, pois o método na classe `A` lança menos exceções que na classe mãe, respeitando a regra. Já o código a seguir não compila:

```
import java.sql.SQLException;
import java.io.IOException;

class A {
    public void method() throws SQLException {
    }
}

class B extends A {
    public void method() throws IOException { // compile error
    }
}
```

Apesar de ambos os métodos lançarem apenas uma exceção, não é isso que importa, pois elas são diferentes. Outro caso que não compila uma vez que `Exception` é super tipo de `IOException`:

```
import java.io.IOException;

class A {
    public void method() throws IOException {
    }
}

class B extends A {
    public void method() throws Exception { // compile error
    }
```

```
}
```

Repare que, quando dizemos *menos exceções que na super classe*, isso indica não apenas quantidade, mas também devemos considerar o polimorfismo. Se trocarmos o exemplo anterior, compilamos pois `IOException` é mais específico que `Exception` na árvore de herança.

```
import java.io.IOException;

class A {
    public void method() throws Exception {
    }
}

class B extends A {
    public void method() throws IOException { // ok
    }
}
```

`Exceptions` do tipo `RuntimeException` podem ser adicionadas sem esse tipo de restrição, uma vez que qualquer método pode jogar uma `RuntimeException`. O exemplo a seguir compila justamente devido a essa regra:

```
class A {
    public void method() {
    }
}

class B extends A {
    public void method() throws RuntimeException { // ok
    }
}

class C extends A {
    public void method() throws ArrayIndexOutOfBoundsException {
        // ok
    }
}
```

Polimorfismo and method invocation

Imagine as classes:

```
class Vehicle {
    void turnon() {
        System.out.println("Vehicle running");
    }
}
class Car extends Vehicle {
    void turnon() {
        System.out.println("Car running");
    }

    void turnoff() {
    }
}
```

Se desejamos um objeto do tipo `Car` com uma referência do tipo `Car`, ou seja, sem usar polimorfismo, podemos invocar os dois métodos:

```
Car c = new Car();
c.turnon(); // Car running
c.turnoff();
```

E, como estamos trabalhando com sobrescrita, o método `turnon` chamado é o da subclasse, `Car`.

Mas e se usarmos polimorfismo e a referência para `Vehicle`?

```
Vehicle v = new Car();
v.turnon(); // Car running
v.turnoff(); // compile error
```

Vamos linha por linha: primeiro, podemos chamar um `Car` de `Vehicle`, porque ele *é um* `Vehicle` (compila sem problemas). Ao herdarmos de `Vehicle`, fizemos com que todo carro seja um veículo. Podemos também chamar o método

`turnon`, pois ambas as classes o possuem. Mas o método que seria invocado é o da classe filha, o sobrescrito.

Já a chamada ao método `turnoff` não compila, porque ele não está definido na classe `Vehicle`. Como a referência é desse tipo, o método não é visível, mesmo que em tempo de execução ele exista no objeto.

A regra é: para saber se um método de um objeto pode ser chamado durante a compilação, olhamos para o tipo da referência em tempo de compilação. Para realmente chamar um método de instância em tempo de execução, devemos olhar para o objeto ao que demos `new`. Lembre-se das exceções em relação a métodos estáticos.

Essa regra faz sentido quando pensamos em um método polimórfico, como o seguinte:

```
class VehicleTester {
    void method (Vehicle v) {
       v.turnon(); // ok
       v.turnoff(); // compile error
    }
}
```

Se passarmos um objeto `Car` para o método, teoricamente ambas as chamadas funcionariam, já que a classe possui tanto o `turnon` quanto o `turnoff`.

Mas imagine uma classe `Motorcicle` que não tem o método `turnoff`:

```
class Vehicle {
    void turnon() {
        System.out.println("Vehicle running");
    }
}
```

```
class Car extends Vehicle {
    void turnon() {
        System.out.println("Car running");
    }

    void turnoff() {
    }
}
class Motorcicle extends Vehicle {
    void turnon() {
        System.out.println("Motorcicle running");
    }
}
```

Como `Motorcicle` *é um* `Vehicle`, podemos passar como argumento. O que aconteceria se pudéssemos ter chamado o `turnoff` na referência `Vehicle`? Alguma coisa estaria errada.

Portanto, a regra geral é que somente podemos acessar os métodos de acordo com o tipo da referência, pois a verificação da existência do método é feita em compilação. Mas qual o método que será invocado, isso será conferido dinamicamente, em execução.

Um ponto muito importante é que o compilador **nunca** sabe o valor das variáveis depois da linha que as cria. Ou seja, ele não sabe se estamos passando um `Car` ou uma `Motorcycle`. O que ele sabe é apenas o tipo da variável; no caso, `Vehicle`. E como `Vehicle` não tem o método `turnoff`, o código não pode compilar.

this, super e sobrescrita de métodos

Na ocasião em que um método foi sobrescrito, podemos utilizar as palavras-chave `super` e `this` para deixar explícito qual método desejamos invocar:

```
class A {
    public void method() {
        System.out.println("a");
    }
}
class B extends A {
    public void method() {
        System.out.println("b");
        super.method(); // a
    }

    public void method2() {
        method(); // b, a
        super.method(); // a
    }
}
```

E se eu invocar o segundo método na primeira classe? Sem o `this`?

```
class A{
    public void method() {
        System.out.println("a");
        method2();
    }
    public void method2() {
        System.out.println("parent method2");
    }
}
class B extends A {
    public void method() {
        System.out.println("b");
        super.method();
    }
    public void method2() {
        System.out.println("c");
        method();
        super.method();
    }
    public static void main(String[] args) {
        new B().method2();
    }
}
```

O programa entra em loop infinito: `B.method2` invoca `B.method` que invoca `A.method` que invoca o método no objeto, `B.method2`. Isto é, o *lookup* do `method2` é feito dinamicamente, encontrando o `method2` que chama `method`, que chama `method` do pai, que chama novamente `method2`.

Note que o *lookup* dos métodos de instância (seu *binding*) é feito em execução, mesmo se invocarmos dentro de um próprio objeto. Até mesmo o uso da palavra-chave `this` não evitaria essa situação, ainda causando o loop:

```
class A{
    public void method() {
        System.out.println("a");
        this.method2();
    }
    public void method2() {
        System.out.println("parent method2");
    }
}
class B extends A {
    public void method() {
        System.out.println("b");
        super.method();
    }
    public void method2() {
        System.out.println("c");
        method();
        super.method();
    }
    public static void main(String[] args) {
        new B().method2();
    }
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    void x() throws IOException {
        System.out.println("c");
    }
}
class C extends B {
    void x() throws FileNotFoundException {
        System.out.println("b");
    }
}
class A {
    public static void main(String[] args) {
        new C().x();
    }
}
```

a) Não compila.

b) Compila e imprime `b`.

c) Compila e imprime `c`.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x() throws IOException {
        System.out.println("c");
    }
}
class C extends B {
    void x() throws FileNotFoundException {
        System.out.println("b");
    }
}
```

```
class A {
    public static void main(String[] args) throws IOException {
        new C().x();
    }
}
```

a) Não compila.

b) Compila e imprime `b`.

c) Compila e imprime `c`.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(double i) throws IOException {
        System.out.println("c");
    }
}
class C extends B {
    void x(int i) throws FileNotFoundException {
        System.out.println("b");
    }
}
class A {
    public static void main(String[] args) throws IOException {
        new C().x(3.2);
    }
}
```

a) Não compila.

b) Compila e imprime `b`.

c) Compila e imprime `c`.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(double i) throws IOException {
        System.out.println("c");
    }
}
class C {
    void x(int i) throws FileNotFoundException {
        System.out.println("b");
    }
}
class A {
    public static void main(String[] args) throws IOException {
        new C().x(3.2);
    }
}
```

a) Não compila.

b) Compila e imprime `b`.

c) Compila e imprime `c`.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
interface B {
    public void x(double i) throws IOException {
        System.out.println("c");
    }
}
class C implements B {
    public void x(int i) throws FileNotFoundException {
        System.out.println("b");
    }
}
class A {
    public static void main(String[] args) throws IOException {
        new C().x(3);
    }
}
```

a) Não compila.

b) Compila e imprime `b`.

c) Compila e imprime `c`.

d) Compila e não imprime nada.

e) Compila e dá exception.

f) Compila e entra em loop.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(int i) throws IOException {
        System.out.println("c");
    }
}
```

```
abstract class C extends B {
    abstract void x(int i) throws IOException;
}
abstract class D extends C {
    void x(int i) throws IOException {
        System.out.println("d");
    }
}
class E extends D {
}
class A {
    public static void main(String[] args) throws IOException {
        new E().x(32);
    }
}
```

a) Não compila.

b) Compila e imprime `b`.

c) Compila e imprime `c`.

d) Compila e imprime `d`.

e) Compila e não imprime nada.

f) Compila e dá exception.

g) Compila e entra em loop.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(int i) throws IOException {
        if(++i<0) return;
        x(-1);
        System.out.println("c");
    }
}
abstract class C extends B {
```

```
    void x(int i) throws IOException {
        System.out.println("b");
        super.x(i);
    }
}
abstract class D extends C {
    void x(int i) throws IOException {
        super.x(i);
    }
}
class E extends D {
}
class A {
    public static void main(String[] args) throws IOException {
        new E().x(32);
    }
}
```

a) Não compila.

b) Compila e imprime `b`.

c) Compila e imprime `c`.

d) Compila e imprime `d`.

e) Compila e não imprime nada.

f) Compila e dá exception.

g) Compila e entra em loop.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(int i) throws IOException {
        if(i<0) return;
        this.x(-1);
        System.out.println("c");
    }
```

```
}
abstract class C extends B {
    void x(int i) throws IOException {
        System.out.println("b");
        super.x(i);
    }
}
abstract class D extends C {
    void x(int i) throws IOException {
        super.x(i);
    }
}
class E extends D {
}
class A {
    public static void main(String[] args) throws IOException  {
        new E().x(32);
    }
}
```

a) Não compila.

b) Compila e imprime `b`, `b`, `c`.

c) Compila e imprime `c`, `b`, `b`.

d) Compila e imprime `c`, `b`, `c`.

e) Compila e não imprime nada.

f) Compila e dá exception.

g) Compila e entra em loop.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
import java.io.*;
class B {
    void x(int i) throws IOException {
        if(i<0) return;
        super.x(-1);
```

```
        System.out.println("c");
    }
}
abstract class C extends B {
    void x(int i) throws IOException {
        System.out.println("b");
        super.x(i);
    }
}
abstract class D extends C {
    void x(int i) throws IOException {
        super.x(i);
    }
}
class E extends D {
}
class A {
    public static void main(String[] args) throws IOException {
        new E().x(32);
    }
}
```

a) Não compila.

b) Compila e imprime `b`.

c) Compila e imprime `c`.

d) Compila e imprime `b`, `c`.

e) Compila e não imprime nada.

f) Compila e dá exception.

g) Compila e entra em loop.

7.3 DIFERENCIE ENTRE O TIPO DE UMA REFERÊNCIA E O DE UM OBJETO

Sempre que estendemos alguma classe ou implementamos

alguma interface, estamos relacionando nossa classe com a classe mãe ou interface usando um relacionamento chamado de **é um**. Se `Car extends Vehicle` , dizemos que *Car* ***é um*** *Vehicle*. Ou se `ArrayList implements List` dizemos que *ArrayList* ***é um*** *List*.

O relacionamento de **é um** é um dos recursos mais poderosos da orientação a objetos. E é chamado formalmente de **polimorfismo**.

Polimorfismo é a capacidade que temos de referenciar um objeto de formas diferentes, segundo seus relacionamentos de *é um*. Em especial, usamos polimorfismo quando escrevemos:

```
Vehicle v = new Car();
List l = new ArrayList();
```

O polimorfismo pode ser aplicado à passagem de parâmetros (e é aí que está seu grande poder). Considere os tipos a seguir:

```
class Vehicle {}
class Car extends Vehicle {}
class Motorcycle extends Vehicle {}
class Bus extends Vehicle {}
class HybridCar extends Car {}
```

Se temos um método que recebe `Vehicle` , podemos passar qualquer um daqueles objetos como parâmetro:

```
class Test{
    void method (Vehicle v) {

    }

    void method2() {
        method(new Car());
        method(new Motorcycle());
        method(new Bus());
        method(new Vehicle());
        method(new HybridCar());
```

```
    }
}
```

Dessa forma, conseguimos obter um forte reaproveitamento de código.

Repare que, quando usamos polimorfismo, estamos mudando o tipo da referência, mas nunca o tipo do objeto. Em Java, objetos nunca mudam seu tipo, que é aquele onde demos `new` . O que fazemos é chamar (referenciar) o objeto de várias formas diferentes. Chamar de várias formas é o polimorfismo.

Podemos referenciar um objeto pelo seu próprio tipo, por uma de suas classes pai, ou por qualquer interface implementada por ele, direta ou indiretamente. Os exemplos a seguir compilam pois o compilador entende que o tipo da referência que está sendo atribuída sempre é do tipo da esquerda:

```
interface A {}
interface B {}
class C implements A {}
class D extends C implements B {}
public class Test {
    public static void main(String[] args) {

        D d = new D(); // ok

        // D extends C, todo D é um C, ok
        C c = new D();
        C c2 = d;

        // D implements B, todo D implementa B, ok
        B b = new D();
        B b2 = d;

        // D implements A indiretamente, ok
        A a = new D();
        A a2 = a;
    }
```

```
}
```

Os exemplos seguintes mostram situações nas quais o compilador reclama, por exemplo, quando tentamos forçar que um tipo é outro que não é, ou quando como desenvolvedores sabemos que em execução a referência será para um tipo determinado, mas o compilador não tem como saber isso.

```
interface A {}
interface B {}
class C implements A {}
class D extends C implements B {}
public class Test {
    public static void main(String[] args) {

        D d2 = new C(); // compile error, C não é D

        D d3 = new D();
        C c3 = d3; // compila
        D d4 = c3; // não compila, por mais que o ser humano
                   // saiba que sim, em execução, nem todo C é
                   // um D.
    }
}
```

E como funciona o acesso às variáveis membro e aos métodos? Se temos uma referência para a classe mãe, não importa o que o valor seja em tempo de execução, o compilador não conhece o tempo de execução, então ele só compila chamadas aos métodos definidos na classe mãe:

```
class Vehicle {
    public void turnon() { }
}
class Car extends Vehicle {
    public void changeGear() {}
}

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

```
        Vehicle v = new Vehicle();
        v.turnon(); // ok

        Car c = new Car();
        c.changeGear(); // ok

        Vehicle v2 = c; // ok
        v2.turnon(); // vehicle.turnon => ok
        v2.changeGear(); // compile error
    }
}
```

Mesmo em casos em que "achamos" que todo veículo tem, se o método não foi definido na classe de referência, o codigo não compila:

```
abstract class Vehicle {
    public void turnon() { }
}
class Car extends Vehicle{
    public void turnoff() { }
}
class Motorbike extends Vehicle{
    public void turnoff() { }
}

class Test{
    public static void main(String[] args) {
        Car c = new Car();
        c.turnoff(); // ok

        Vehicle v2 = c;
        v2.turnoff(); // vehicle.turnoff??? compile error
    }
}
```

O mesmo valerá para variáveis membro:

```
class Vehicle {
    int speed;
}
class Car extends Vehicle {
    int gear;
```

```
}

class Test {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        v.speed = 3; // ok

        Car c = new Car();
        c.gear = 1; // ok
        c.speed = 2; // ok

        Vehicle v2 = c;
        v2.speed = 5; // ok
        v2.gear = 7; // vehicle.gear????? compile error
    }
}
```

E temos de cuidar de mais um caso específico: o que acontece se estamos trabalhando com pacotes distintos?

Se o método da classe pai que está sendo sobrescrito é `public`, os métodos que sobrescrevem devem ser `public`, então não tem muita graça. Já se o método da classe pai é `protected`, os filhos são `protected` ou `public`, que também não tem graça, pois o filho — mesmo em outro pacote — já tinha acesso ao método do pai.

Mas o que acontece se o método no pai é `private` e eu tento sobrescrevê-lo? Ou se o método é `default` e tento sobrescrevê-lo em outro pacote? O exemplo a seguir valerá tanto para `private` quando para o modificador de escopo padrão:

```
package savings;
public class SavingsAccount extends model.Account {
    void close() {
        System.out.println("closing savings account");
    }
}

package model;
```

```
public class Account {
    void close() {
        System.out.println("closing base account");
    }
}
```

Ao invocar o método `close` através de uma referência para `Account` ou `SavingsAccount`, o resultado pode variar, como no exemplo a seguir:

```
package ????;

class Test {
    public static void main(String[] args) {
        SavingsAccount c = new SavingsAccount();
        c.close();

        Account d = c;
        d.close();
    }
}
```

Esse código pode não compilar, dependendo do pacote onde ele está. Como assim? Acontece que o método não foi sobrescrito, a classe filha nem sabe da existência do método (`privado` ou `default`) do pai, portanto, o que ela fez foi criar um método totalmente novo.

Nesse caso, ao invocarmos o método durante compilação, o *binding* é feito para o método específico de cada uma delas, uma vez que são métodos totalmente diferentes. Se o código está no pacote de modelo, a chamada ao método `close` de `Account` compila e imprimiria `closing base account`, já a chamada ao método `close` da referência `SavingsAccounts` não compila.

```
package model;

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

```
        SavingsAccount c = new SavingsAccount();
        c.close(); // compile error

        Account d = c;
        d.close(); // closing base account
    }
}
```

Se estivermos no pacote `savings`, o resultado é o inverso:

```
package savings;

class Test {
    public static void main(String[] args) {
        SavingsAccount c = new SavingsAccount();
        c.close(); // closing savings account

        Account d = c;
        d.close(); // compile error
    }
}
```

Lembre-se de que os métodos privados terão um efeito equivalente: eles só são vistos internamente à classe onde foram definidos.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class D extends C {
    void x() { System.out.println(1); }
}
class C extends B {
    void x() { System.out.println(2); }
}
class B {
    void x() { System.out.println(3); }
    void y(B b) {
        b.x();
```

```
    }
    void y(C b) {
        c.x();
    }
    void y(D b) {
        d.x();
    }
}
class A {
    public static void main(String[] args) {
        new B().y(new C());
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class D extends C {
    void x() { System.out.println(1); }
}
class C extends B {
    void x() { System.out.println(2); }
}
class B {
    void x() { System.out.println(3); }
    void y(B b) {
        b.x();
    }
    void y(C c) {
        c.x();
    }
    void y(D d) {
        d.x();
    }
}
```

```
class A {
    public static void main(String[] args) {
        new B().y(new C());
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class D extends C {
    void x() { System.out.println(1); }
    void y(C b) {
        x();
    }
}
class C extends B {
    void x() { System.out.println(2); }
}
class B {
    void x() { System.out.println(3); }
    void y(B b) {
        b.x();
    }
}
class A {
    public static void main(String[] args) {
        new B().y(new C());
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class D extends C {
    void x() { System.out.println(1); }
    void y(C b) {
        x();
    }
}
class C extends B {
    void x() { System.out.println(2); }
}
class B {
    void x() { System.out.println(3); }
    void y(B b) {
        b.x();
    }
}
class A {
    public static void main(String[] args) {
        new D().y(new C());
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e imprime 3.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package savings;
public class SavingsAccount extends model.Account {
```

```
    void close() {
        System.out.println("closing savings");
    }
}
```

```
package model;
public class Account {
    void close() {
        System.out.println("closing base");
    }
}
```

```
package code;
import model.*;
import savings.*;
class A {
    public static void main(String[] args) {
        new Account().close();
    }
}
```

a) Não compila.

b) Compila e roda jogando exception.

c) Compila e roda, imprimindo `closing savings` .

d) Compila e roda, imprimindo `closing base` .

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package savings;
public class SavingsAccount extends model.Account {
    public void close() {
        System.out.println("closing savings");
    }
}
```

```
package model;
public class Account {
    public void close() {
        System.out.println("closing base");
```

```
    }
}
```

```
package model;
import model.*;
import savings.*;
class A {
    public static void main(String[] args) {
        new Account().close();
    }
}
```

a) Não compila.

b) Compila e roda jogando exception.

c) Compila e roda, imprimindo `closing savings` .

d) Compila e roda, imprimindo `closing base` .

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package savings;
public class SavingsAccount extends model.Account {
    void close() {
        System.out.println("closing savings");
    }
}
```

```
package model;
public class Account {
    protected void close() {
        System.out.println("closing base");
    }
}
```

```
package code;
import model.*;
import savings.*;
class A {
    public static void main(String[] args) {
        new Account().close();
```

```
    }
}
```

a) Não compila.

b) Compila e roda jogando exception.

c) Compila e roda, imprimindo `closing savings` .

d) Compila e roda, imprimindo `closing base` .

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
package savings;
public class SavingsAccount extends model.Account {
    public void close() {
        System.out.println("closing savings");
    }
}
```

```
package model;
public class Account {
    public void close() {
        System.out.println("closing base");
    }
}
```

```
package code;
import savings.*;
import model.*;
class A {
    public static void main(String[] args) {
        Account c = new SavingsAccount();
        c.close();
    }
}
```

a) Não compila.

b) Compila e roda jogando exception.

c) Compila e roda, imprimindo `closing savings` .

d) Compila e roda, imprimindo `closing base` .

9) O que acontece com o código a seguir?

```
interface Vehicle {
    int getSpeed();
    void turnon();
}

abstract class Car implements Vehicle {
    public void turnon()  {
        System.out.println("on!");
    }
}

class ConcreteCar extends Car implements Vehicle {
    public int getSpeed() {
        return 1;
    }
}
```

7.4 DETERMINE QUANDO É NECESSÁRIO FAZER CASTING

Às vezes, temos referências de um tipo, mas sabemos que lá há um objeto de outro tipo, um mais específico:

```
public class Test{
    public static void main(String...args){
        Object[] objects = new Object[100];

        String s = "certification";
        objects[0] = s;

        String recovered = objects[0];
    }
}
```

O código anterior não compila:

```
Test.java:3: incompatible types
found   : java.lang.Object
required: java.lang.String
                String recovered = objects[0];
                                              ^
1 error
```

Temos um array de referências para `Object`. Nem todo `Object` é uma `String`, então o compilador não vai deixar você fazer essa conversão. Lembre-se que, em geral, o compilador não conhece os *valores* das variáveis, apenas seus tipos.

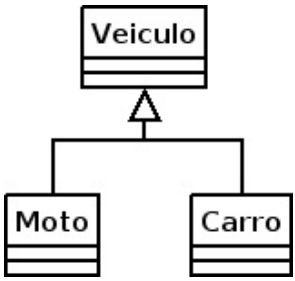
Vamos precisar *moldar* a referência para que o código compile:

```
String recovered = (String) objetos[0];
```

A partir de agora, esse código compila. Mas será que roda? Durante a execução, o **casting** vai ver se aquele objeto é mesmo compatível com o tipo `String` (no nosso caso é). Se não fosse, ele lançaria uma `ClassCastException` (exceção *unchecked*).

Considere as classes:

```
class Vehicle {}
class Motorcycle extends Vehicle {}
class Car extends Vehicle {}
```



E o código:

```
Vehicle v = new Car();
Motorcycle m = v;
```

Na primeira linha, usamos polimorfismo para chamar um `Car` de `Vehicle` (**é um**). Na segunda linha, o que o compilador sabe é que `v` é do tipo `Vehicle`. E *nem todo* `Vehicle` *é uma* `Motorcycle`. Por isso, essa linha não compila.

Mas existem *alguns* `Vehicle` que são `Motorcycle`. Então, o compilador deixa que façamos o casting:

```
Vehicle v = new Car();
Motorcycle m = (Motorcycle) v;
```

Com isso, o código compila, mas repare que, em tempo de execução, `v` aponta para um objeto do tipo `Car`. Quando o código for executado, haverá um erro de execução: `ClassCastException`. `Car` **não** *é uma* `Motorcycle`.

Cuidado que, se o *casting* for totalmente impossível, o compilador já acusará erro:

```
Car c = new Car();
Motorcycle m = (Moto) c;
```

Um `Car` **nunca** poderá ser uma `Motorcycle`. Então, nem com casting isso compila.

É importante lembrar de que, quando não precisamos de casting, ele é opcional, portanto, todas as linhas a seguir funcionam com ou sem casting:

```
String guilherme = "guilherme";
String name = guilherme;
String name2 = (String) guilherme;
Object name3 = guilherme;
```

```
Object name4 = (String) guilherme;
Object name5 = (Object) guilherme;
```

Regra geral!

Se você está subindo na hierarquia de classes, a autopromoção vai fazer tudo sozinho; e se você estiver descendo, vai precisar de casting. Se não houver um caminho possível, não compila nem com casting.

Na prova, faça sempre os diagramas de hierarquia de tipos que fica extremamente fácil resolver esses castings.

Casting com interfaces

Dado o código a seguir:

```
Car c = new Car();
Motorcycle m = (Motorcycle) c;
```

Quando dizemos que ele não compila, é porque um `Car` nunca pode ser uma `Motorcycle`. Mas como o compilador sabe que isso é impossível mesmo? Existe alguma chance de algum objeto de qualquer tipo ser, ao mesmo tempo, `Car` e `Motorcycle`?

```
class X extends Motorcycle, Car { // compile error

}
```

A única maneira de isso acontecer seria se Java suportasse herança múltipla; aí escreveríamos uma classe que herdasse de `Car` e `Motorcycle` ao mesmo tempo. Como Java não tem

herança múltipla, isso realmente é impossível de acontecer.

Mas e quando fazemos casting com interfaces envolvidas? Apesar de não existir herança múltipla, podemos implementar múltiplas interfaces! Fazer casting para interfaces sempre é possível e vai compilar (há apenas uma exceção a essa regra).

Pegue uma interface *qualquer*, por exemplo `Runnable`. O código a seguir compila:

```
Car c = new Car();
Runnable r = (Runnable) c;
```

Um `Car` *pode ser* um `Runnable`? Sabemos que a classe `Car` propriamente não implementa essa interface. Mas existe a possibilidade de existir algum objeto em Java que seja, ao mesmo tempo, `Car` e `Runnable`?

A resposta é sim! E se tivéssemos uma classe `RunnableCar`?

```
javaclass RunnableCar extends Car implements Runnable { ... }
```

O compilador não sabe o valor da variável `c` naquele exemplo. Ele não sabe que na verdade é uma instância de `Car` e não de `RunnableCar`. Ele sabe apenas que é do tipo `Car` e, pela simples possibilidade de existir um objeto que seja `Car` e `Runnable`, ele deixa o código compilar.

Mas repare que a classe `RunnableCar` não existe no diagrama original. Mesmo assim, o código compila! Apenas com a *possibilidade* de existir uma classe dessa, o compilador já aceita aquele casting, mesmo que uma classe dessas não exista na prática.

Claro que o objeto é do tipo `Car`, que não implementa

`Runnable` e, em tempo de execução, vai ocorrer uma `ClassCastingException`.

E FINAL?

Dizemos que o código anterior compila porque há a possibilidade de uma classe como `RunnableCar` existir algum dia. Mas será que sempre há essa possibilidade mesmo?

Se a classe `Car` for `final`, é impossível existir uma classe filha dela. E como a própria `Car` não implementa `Runnable`, nesse caso, será impossível fazer o casting para `Runnable` (o próprio compilador já acusa erro). Essa é a única exceção a regra de o compilador sempre aceitar casts para interfaces.

Dica

Muitos exercícios são sobre casting de referência. Uma dica é seguir o que é possível, impossível e óbvio. Se é óbvio que o casting funciona, isso é, se a conversão é sempre verdade, a autopromoção faz sozinha.

Se o casting é possível, mas nem sempre é verdade, o casting compila, mas pode lançar erro em tempo de execução. Se o casting é impossível, isto é, ele nunca pode dar certo, o código não vai compilar nem com casting.

Em alguns livros, você encontra tabelas complicadas e grandes que o "ajudam" a decidir se o casting compila e roda, mas é muito mais fácil seguir pela lógica.

instanceof

O operador `instanceof` (`variable instanceof ClassName`) devolve `true` caso a referência `variable` aponte para um objeto do tipo ou subtipo de `ClassName` .

```
Object c = new Car();
boolean b1 = c instanceof Car; // true
boolean b2 = c instanceof Motorcycle; // false
```

O `instanceof` não compila se a referência em questão for obviamente incompatível, por exemplo:

```
String s = "a";
boolean b  = s instanceof java.util.List; // compile erro
```

CODE SMELL

O `instanceof` é um operador que deve ser usado com extremo cuidado no dia a dia. Em muitos casos, ele indica a fraca modelagem de um sistema, com blocos que parecem "*switchs*" e poderiam ser trocados por polimorfismo.

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {};
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
    public static void main(String[] args) {
        B b = new C();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
    public static void main(String[] args) {
        C c = (C) new B();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
    public static void main(String[] args) {
        Y y = new D();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
    public static void main(String[] args) {
        Y y = (Y) new D();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main` .

c) Compila e roda sem exception.

d) Compila e roda dando exception.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
    public static void main(String[] args) {
        Z z = (Z) (B) new D();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
    public static void main(String[] args) {
        Y y = (Y) new A();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda sem exception.

d) Compila e roda dando exception.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
```

```
class E extends C {}
class A {
    public static void main(String[] args) {
        D d = (D) (Y) (B) new D();
    }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e roda.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
  public static void main(String[] args) {
    System.out.println(((B) (Z) (W) (Y) new D()) instanceOf D);
  }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e imprime `true`.

d) Compila e imprime `false`.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
interface Z {}
```

```
interface W {}
interface Y extends Z, W {}
class B {}
class C extends B implements Y {}
class D extends B implements Z, W {}
class E extends C {}
class A {
  public static void main(String[] args) {
    System.out.println(((B) (Z) (W) (Y) new D()) instanceof D);
  }
}
```

a) Não compila na definição das classes e interfaces.

b) Não compila dentro do método `main`.

c) Compila e imprime `true`.

d) Compila e imprime `false`.

e) Compila mas lança exception.

7.5 USE SUPER E THIS PARA ACESSAR OBJETOS E CONSTRUTORES

Um construtor pode ser sobrecarregado assim como os métodos, e pode ter qualquer modificador de visibilidade. O ponto mais importante sobre os construtores é que, para construir um objeto de uma classe filha, obrigatoriamente, precisamos chamar um construtor da classe mãe antes. Sempre, em todos os casos.

Para chamar o construtor da mãe, usamos a chamada ao `super` (passando ou não argumentos):

```
class Parent {
    public Parent(String msg) {
        System.out.println(msg);
    }
```

```
}

class Child extends Parent {
    public Child(String name) {
        super("parent");
        System.out.println("child");
    }
}
```

Mas, na maioria dos casos, não chamamos o construtor da mãe explicitamente. Se nenhum construtor da mãe foi escolhido através da palavra `super(...)`, o compilador coloca **automaticamente** `super();` no começo do nosso construtor, sem nem olhar para a classe mãe.

```
class Parent {
    public Parent() {
        System.out.println("parent");
    }
}

class Child extends Parent {
    public Child(String name) {
        // super(); // implicit!
        System.out.println("child");
    }
}
```

Considerando agora o código:

```
public class X{
    public static void main(String [] args){
        Child child = new Child("test");
    }
}
```

Vai primeiro imprimir `"parent"`, e só depois `"child"`.

Uma outra possibilidade, no caso de termos mais de um construtor, é chamarmos outro construtor da própria classe através do `this()`:

```
class Parent {
    public Parent() {
        System.out.println("parent");
    }
}

class Child extends Parent {
    public Child() {
        // super(); // implicit
        System.out.println("child 1");
    }

    public Child(String name) {
        this();
        System.out.println("child 2");
    }
}

public class X{
    public static void main(String [] args){
        Child child = new Child("test");
    }
}
```

Agora vai produzir `"parent"`, `"child 1"` e `"child 2"`.

Atenção, a chamada do construtor com `super` ou `this` só pode aparecer como primeira instrução do construtor. Portanto, só podemos fazer uma chamada desses tipos.

```
class Child extends Object {
    public Child() {
        // super(); // implicit
    }

    public Child(String name) {
        this();
    }

    public Child(int age) {
        super();
        this(); // compile error
    }
```

```
    public Child(long l) {
        this();
        this(); // compile error
    }

    public Child(char c) {
        super();
        super(); // compile error
    }
}
```

this e variáveis membro

Por vezes, temos variáveis membro com o mesmo nome de variáveis locais. O acesso sempre será a variável local, exceto quando colocamos o `this`, que indica que a variável membro será acessada. O código a seguir imprimirá `3` e depois `5`:

```
class Test {
    int i = 5;
    void run(int i) {
        System.out.println(i);
        System.out.println(this.i);
    }
    public static void main(String[] args) {
        new Test().run(3);
    }
}
```

No acesso a variáveis membro com o `this`, podem parecer que serão acessados somente valores da classe atual, mas buscam também nas classes da qual ela herda:

```
class A{
    int i = 5;
}
class Test extends A{
    void run(int i) {
        System.out.println(this.i); // 5
    }
    public static void main(String[] args) {
```

```
        new Test().run(3);
    }
}
```

Tentar acessar uma variável local com `this` não compila:

```
class Test {
    void run(int i) {
        System.out.println(this.i); // compile error: this.i
    }
    public static void main(String[] args) {
        new Test().run(3);
    }
}
```

Como mostramos, caso a variável seja escondida por uma variável com mesmo nome em uma classe filha, podemos diferenciar o acesso à variável membro da classe filha ou da pai, explicitando `this` ou `super`:

```
class A{
    int i = 5;
}
class Test extends A{
    int i = 10;
    void run(int i) {
        System.out.println(i); // 3
        System.out.println(this.i); // 10
        System.out.println(super.i); // 5
    }
    public static void main(String[] args) {
        new Test().run(3);
    }
}
```

O `this` é em geral opcional para acessar um método do nosso objeto atual (se ele não foi redefinido, da classe mãe):

```
class A{
    int i() { return 5; }
}
class Test extends A{
```

```
    void run() {
        System.out.println(this.i());
    }
    public static void main(String[] args) {
        new Test().run(); // 5
    }
}
class Test2 {
    int i() { return 10; }
    void run() {
        System.out.println(this.i());
    }
    public static void main(String[] args) {
        new Test2().run(); // 10
    }
}
```

this e super ao acessar uma variável membro

E o que acontece quando uma variável membro tem o mesmo nome que a definida na classe que herdamos? Se não definirmos o acesso por meio de `this` nem `super`, o acesso é à variável da classe filha. Se usarmos `this`, é à classe filha novamente e, se usarmos `super`, é à classe pai:

```
class Vehicle {
    double speed = 30;
}
class Car extends Vehicle {
    double speed = 50;
    void print() {
        System.out.println(speed); // 50
        System.out.println(this.speed); // 50
        System.out.println(super.speed); // 30
    }
}
class Test {
    public static void main(String[] args) {
        Car c = new Car();
        c.print();
    }
}
```

Lembre-se de que o binding de uma variável ao tipo é feito em compilação. Portanto, se tentarmos acessar a variável speed fora do `Car` através de uma referência a `Car`, o valor alterado é o da variável `Car.speed`:

```
class Vehicle {
    double speed = 30;
}
class Car extends Vehicle {
    double speed = 50;
    void print() {
        System.out.println(speed); // 1000
        System.out.println(this.speed); // 1000
        System.out.println(super.speed); // 30
    }
}
class Test {
    public static void main(String[] args) {
        Car c = new Car();
        c.speed = 1000;
        c.print();
    }
}
```

E se fizermos o mesmo através de uma referência a `Vehicle`, alteramos a speed do `Vehicle`:

```
class Vehicle {
    double speed = 30;
}
class Car extends Vehicle {
    double speed = 50;
    void print() {
        System.out.println(speed); // 50
        System.out.println(this.speed); // 50
        System.out.println(super.speed); // 1000
    }
}
class Test {
    public static void main(String[] args) {
        Car c = new Car();
        ((Vehicle) c).speed = 1000;
```

```
        c.print();
    }
}
```

Estático não tem this nem super

Contextos estáticos não possuem nem `this` nem `super`, uma vez que o código não é executado dentro de um objeto:

```
class A{
    int i = 5;
}
class Test extends A{
    int i = 10;
    public static void main(String[] args) {
        this.i = 5; // this? compile error
        super.i = 10; // super? compile error
    }
}
```

Por fim, uma última restrição: interfaces não podem ter métodos estáticos, não compila (métodos `default` adicionados no Java 8 podem existir).

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    int x = 1;
}
class A extends B {
    static int x = 2;
    public static void main(String[] args) {
        System.out.println(x);
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e dá exception.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    int x = 1;
}
class A extends B {
    static int x = 2;
    public static void main(String[] args) {
        System.out.println(this.x);
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e dá exception.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    int x = 1;
}
class A extends B {
    static int x = 2;
    public static void main(String[] args) {
        System.out.println(super.x);
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e dá exception.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    int x = 1;
}
class A extends B {
    static int x = 2;
    public static void main(String[] args) {
        System.out.println(new A().super.x);
    }
}
```

a) Não compila.

b) Compila e imprime 1.

c) Compila e imprime 2.

d) Compila e dá exception.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    void B() {
    }
    void B(String s) {
        this();
        this(s);
    }
}
class A {
    public static void main(String[] args) {
        B b = new B();
```

```
    }
}
```

a) Não compila.

b) Compila e dá exception.

c) Compila e não imprime nada.

d) Compila e entra em loop infinito.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    B() {
    }
    B(String s) {
        this();
        this(s);
    }
}
class A {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

a) Não compila.

b) Compila e dá exception.

c) Compila e não imprime nada.

d) Compila e entra em loop infinito.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    B() {
```

```
    }
    B(String s) {
        this();
    }
}
class A {
    public static void main(String[] args) {
        B b = new B();
    }
}
```

a) Não compila.

b) Compila e dá exception.

c) Compila e não imprime nada.

d) Compila e entra em loop infinito.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    B() {
    }
    B(String s) {
        this();
    }
}
class A {
    public static void main(String[] args) {
        String s = null;
        B b = new B(s);
    }
}
```

a) Não compila.

b) Compila e dá exception.

c) Compila e não imprime nada.

d) Compila e entra em loop infinito.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    int x() { return y();}
    int y() { return 3; }
}
class C extends B {
    C() {
        this(x());
    }
    C(int i) {
        System.out.println(i);
    }
    int y() { return 2; }
}
class A {
    public static void main(String[] args) {
        new C();
    }
}
```

a) Não compila.

b) Compila e imprime `2`.

c) Compila e imprime `3`.

10) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class B {
    int x() { return y();}
    int y() { return 3; }
}
class C extends B {
    C() {
        super();
        z(x());
    }
```

```
    void z(int i) {
        System.out.println(i);
    }
    int y() { return 2; }
}
class A {
    public static void main(String[] args) {
        new C();
    }
}
```

a) Não compila.

b) Compila e imprime `2`.

c) Compila e imprime `3`.

7.6 USE CLASSES ABSTRATAS E INTERFACES

Classes e métodos podem ser abstratos. Mesmo que soe estranho, uma classe abstrata pode não ter nenhum método abstrato:

```
abstract class NoMethods {
}
```

Se uma classe tem um método que é abstrato, ela deve ser declarada como abstrata, ou não compilará, como no exemplo a seguir:

```
class BasicClass { // compile error
    public abstract void execute();
}
```

Uma classe abstrata não pode ser instanciada diretamente:

```
abstract class X{

}
```

```
public class Test{
    public static void main(String[] args) {
        X x = new X(); // compile error
    }
}
```

Como o compilador indica:

```
Test.java:7: X is abstract; cannot be instantiated
        X x = new X();
                  ^
1 error
```

Um método abstrato é um método sem corpo, somente com a definição. Uma classe que tem um ou mais métodos abstratos precisa ser declarada como abstrata.

```
abstract class Vehicle {
    public abstract void turnon();
}
```

Uma classe concreta que herda de uma abstrata precisa implementar os métodos para compilar:

```
class Motorcycle extends Vehicle { // ok
    public void turnon() {

    }
}
```

O exemplo a seguir mostra a herança que não compila, uma vez que o método herdado não foi implementado:

```
class SemRodas extends Vehicle { // compile error
}
```

Caso a subclasse seja abstrata, os métodos abstratos da superclasse não precisam ser implementados:

```
abstract class Truck extends Vehicle { // ok
}
```

Um método abstrato tem de ser reescrito ou herdado pelas suas filhas concretas. Agora veja o exemplo a seguir:

```
abstract class Vehicle {
    public abstract void turnon();
}

class Motorcycle extends Vehicle {
    public void turnon() {

    }
}
```

O método `turnon` foi implementado na classe filha, então ela pode ser concreta. Basta pensar que métodos abstratos herdados são *responsabilidades herdadas*: você não poderá ser um objeto concreto enquanto tiver responsabilidades a serem tratadas.

Quando herdamos de uma classe abstrata que possui um método abstrato, temos de escolher: ou implementamos o método, ou somos abstratos também e *passamos adiante* a responsabilidade. Note que a classe pode implementar o método e mesmo assim também ser abstrata.

```
abstract class Vehicle {
    public abstract void turnon();
}

abstract class Motorcycle extends Vehicle { // ok

}

class SpecialMotorcycle extends Motorcycle {
    public void turnon() {

    }
}
```

Note que, mesmo implementando o método, posso decidir

manter a classe abstrata:

```
abstract class Truck extends Vehicle { // ok
    public void turnon() {
    }
}
```

O código de uma classe abstrata pode invocar métodos dela mesma, que ainda são abstratos, uma vez que ele só será executado quando o objeto for criado:

```
abstract class X{
    void x() {
        System.out.println(y());
    }
    abstract String y();
}
class Y extends X {
    String y() {
        return "code";
    }
}
public class Test {
    public static void main(String[] args) {
        new Y().x(); // code
    }
}
```

Interfaces

Uma interface declara métodos que deverão ser implementados pelas classes concretas que queiram ser consideradas como tal. Por padrão, são todos métodos públicos e abstratos.

```
interface Vehicle {
    /* public abstract */ void turnon();
    public abstract int getSpeed();
}
```

Quando você implementa a interface em uma classe concreta, é preciso implementar todos os métodos:

```
interface Vehicle {
    /* public abstract */ void turnon();
    public abstract int getSpeed();
}

class Car implements Vehicle {
    public void turnon() {
    }
    public int getSpeed() {
        return 0;
    }
}
```

Similarmente, ao herdar uma classe abstrata, a classe concreta deve implementar todos os métodos que não foram implementados ainda:

```
interface Vehicle {
    void turnon();
    int getSpeed();
}

class Motorcycle implements Vehicle {
                    // compile error, falta getSpeed()
    public void turnon() {
    }
}

class Truck implements Vehicle {
    public void turnon() {
    }
    int getSpeed() { // PUBLIC!!! compile error
        return 0;
    }
}
```

Valem as mesmas regras de quando você herda de uma classe abstrata: ou você tem todos os métodos reescritos, e aí pode

declará-la como concreta, ou então você precisa declará-la como abstrata.

```
interface Vehicle {
    void turnon();
    int getSpeed();
}

abstract class Motorcycle implements Vehicle { // ok
    public void turnon() {
    }
}
```

Uma classe pode implementar diversas interfaces:

```
abstract class MyType implements Serializable, Runnable {
}
```

Justamente por isso, a prova vê como um bom uso de interfaces quando você quer herdar de dois lugares, mas a herança de classes não permite. Para a prova, essa razão é suficiente, porém na prática existe uma diferença grande entre composição (herança de interfaces não envolve herdar comportamento e variáveis membro) e herdar comportamento e variáveis membro de uma classe mãe. Como a implementação de uma interface nos obriga a escrever todos os métodos ou delegarmos a execução para outros objetos, estamos compondo nossa classe de diversas interfaces.

Lembre-se de que uma interface pode herdar de outra, inclusive de diversas interfaces:

```
interface A extends Runnable {}
interface B extends Serializable {}
interface C extends Runnable, Serializable {}
```

Note que uma interface nunca implementa outra interface:

```
interface A implements Runnable {} // compile error
```

Você pode declarar variáveis em uma interface, todas elas serão `public final static` , isto é, constantes.

```
interface X {
    /* public static final */ int i = 5;
    public /* static final */ int j = 5;
    public static /* final */ int k = 5;
    public static final       int l = 5;
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
abstract class B {
    void x() {
        System.out.println(y());
    }
    abstract int y();
}
abstract class C extends B {
    int y() { return 1; }
}
class D extends C {
    int y() { return 2; }
}
class A {
    public static void main(String[] args) {
        D d  = (D) (C) new D();
        d.x();
    }
}
```

a) Não compila.

b) Compila e imprime `1` .

c) Compila e imprime `2` .

d) Compila e roda com exception.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
abstract class B {
    abstract void x() {
        System.out.println(y());
    }
    abstract int y();
}
abstract class C extends B {
    int y() { return 1; }
}
class D extends C {
    int y() { return 2; }
}
class A {
    public static void main(String[] args) {
        D d  = (D) (C) new D();
        d.x();
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e roda com exception.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
abstract class B {
    void x() {
        System.out.println(y());
    }
    abstract int y();
}
abstract class C extends B {
    abstract int y();
}
```

```
class D extends C {
    int y() { return 1; }
}
class A {
    public static void main(String[] args) {
        D d  = (D) (C) new D();
        d.x();
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e roda com exception.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
abstract class B {
    void x() {
        System.out.println(y());
    }
    int y() {
        return 2;
    }
}
abstract class C extends B {
    abstract int y();
}
class D extends C {
    int y() { return 1; }
}
class A {
    public static void main(String[] args) {
        D d  = (D) (C) new D();
        d.x();
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e roda com exception.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
abstract class B {
    void x() {
        System.out.println(y());
    }
    final int y() {
        return 2;
    }
}
abstract class C extends B {
    int y() {
        return 3;
    }
}
class D extends C {
    int y() { return 1; }
}
class A {
    public static void main(String[] args) {
        D d  = (D) (C) new D();
        d.x();
    }
}
```

a) Não compila.

b) Compila e imprime `1`.

c) Compila e imprime `2`.

d) Compila e roda com exception.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
abstract class B {
    void x() {
        System.out.println(y());
    }
    Object y() { return "a"; }
}
abstract class C extends B {
    abstract String y();
}
class D extends C {
    String y() { return "b"; }
}
class A {
    public static void main(String[] args) {
        D d  = (D) (C) new D();
        d.x();
    }
}
```

a) Não compila.

b) Compila e imprime `a`.

c) Compila e imprime `b`.

d) Compila e roda com exception.

Capítulo 8

LIDANDO COM EXCEÇÕES

8.1 DIFERENCIE ENTRE EXCEÇÕES DO TIPO CHECKED, RUNTIME E ERROS

Durante a execução de uma aplicação, erros podem acontecer. A linguagem Java oferece um mecanismo para que o programador possa definir as providências apropriadas a serem tomadas na hora em que um *erro de execução* ocorrer. Esse mecanismo nos permite controlar o que deve acontecer quando tais situações surgirem.

Os erros de execução são classificados em algumas categorias. É fundamental que você seja capaz de, dado um erro de execução, determinar seu tipo. A classificação das categorias depende exclusivamente da hierarquia das classes que modelam os erros de execução como objetos em Java.

A classe principal dessa hierarquia é a classe `Throwable`. Qualquer erro de execução é um objeto dessa classe ou de uma que deriva dela.

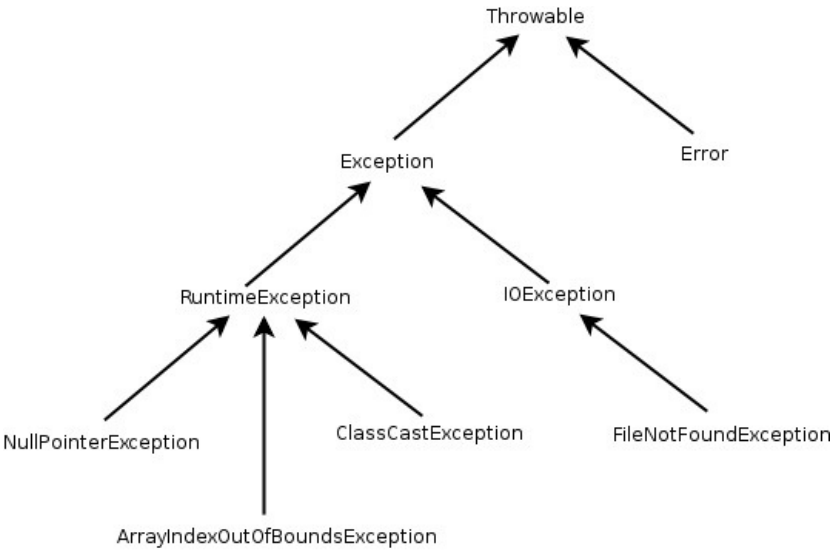
Como filhas diretas de `Throwable` temos: `Error` e `Exception`. Os `Error`s são erros de execução gerados por uma situação totalmente anormal que não deveria ser prevista pela aplicação. Por exemplo, um `OutOfMemoryError` é gerado quando

a JVM não tem mais memória RAM disponível para oferecer para as aplicações. Em geral, esse tipo de erro não é responsabilidade das aplicações, pois quem cuida do gerenciamento de memória é a JVM.

Por outro lado, as `Exception` s são erros de execução que são de responsabilidade das aplicações, ou seja, são as aplicações que devem tratar ou evitar esses erros. Por exemplo, um `SQLException` é gerado quando algum erro ocorre na comunicação entre a aplicação e o banco de dados. Esse tipo de erro deve ser tratado ou evitado pela aplicação.

Por sua vez, as `Exception` s são divididas em duas categorias: as **unchecked**s e as **checked**s. As *uncheckeds* são exceptions que teoricamente podem ser mais facilmente evitadas pelo próprio programador se ele codificar de maneira mais cuidadosa. As *checkeds* são exceptions que teoricamente não são fáceis de evitar, de modo que a melhor abordagem é estar sempre preparado para seu acontecimento.

As uncheckeds são definidas pelas classes que derivam de `RuntimeException` , que por sua vez é filha direta de `Exception` . As outras classes na árvore da `Exception` definem as checkeds.



Essas diferenças não ficam apenas na teoria. O compilador vai verificar se seu programa pode lançar alguma checked exception e, neste caso, obrigá-lo a tratar essa exception de alguma maneira. No caso das exceptions unchecked, não há nenhuma verificação por parte do compilador pelo tratamento ou não.

Exercícios

1) Dentre as classes a seguir, qual delas não é checked?

a) `java.io.IOException`

b) `java.sql.SQLException`

c) `java.lang.Exception`

d) `java.lang.IndexOutOfBoundsException`

e) `java.io.FileNotFoundException`

8.2 EXCEÇÕES: O QUE SÃO E PARA QUE SÃO USADAS EM JAVA

Imagine a situação em que tentamos acessar uma posição em um array:

```
public void execute(Integer[] ages) {
    System.out.println(ages[0]);
}
```

O que acontece se o array enviado para o método é vazio? Mas e se o valor fosse nulo, teríamos de nos preocupar com esse caso específico para não imprimir nulo:

```
public void execute(Integer[] ages) {
    if(ages[0]==null) return;

    System.out.println(ages[0]);
}
```

Pense como seria difícil tratar todas as situações possíveis que *fogem do padrão* de comportamento que estamos desejando. Nesse caso, o comportamento padrão, aquilo que acontece 99% das vezes e que esperamos que aconteça, é que a posição acessada dentro do array seja válida.

Não queremos ter de verificar toda vez se o valor é válido, e não queremos entupir nosso código com diversos `ifs` para diversas condições. As exceções à regra, as *exceptions*, são a alternativa para o controle de fluxo: em vez de usarmos `ifs` para controlar o fluxo que foge do padrão, é possível usar as *exceptions* para esse papel. Veremos adiante como tratar erros, como o acesso a posições inválidas, tentar acessar variáveis com valores inválidos

etc.

Caso uma exception estoure e sua *stack trace* seja impressa, teremos algo como:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 0
    at YourClass.doSomething(YourClass.java:20)
    at YourClass.main(YourClass.java:30)
```

Note como a *stack trace* indica que método estava sendo invocado, em qual linha do arquivo fonte está essa invocação, quem invocou este método etc.

O importante é lembrar de que as *exceptions* permitem que isolemos o tratamento de um comportamento por blocos, separando o bloco de lógica de nosso negócio do bloco de tratamentos de erros (sejam eles *Exceptions* ou *Errors*, como veremos adiante). O stack trace de uma `Exception` também ajuda a encontrar onde exatamente o problema ocorreu e o que estava sendo executado naquela *Thread* naquele instante.

Exercícios

1) Escolha 2 opções. Exception é um mecanismo para...

a) tratar entrada de dados do usuário.

b) que você pode usar para determinar o que fazer quando algo inesperado acontece.

c) que a VM usa para fechar o programa caso algo inesperado aconteça.

d) controlar o fluxo da aplicação.

e) separar o tratamento de erros da lógica principal.

2) De que maneira a API de exceptions pode ajudar a melhorar o código de seu programa? Escolha 2 opções.

a) Permitindo separar o tratamento de erro da lógica do programa.

b) Permitindo tratar o erro no mesmo ponto onde ele ocorre.

c) Permitindo estender as classes que já existem e criar novas exceptions.

d) Disponibilizando várias classes com todas as exceptions possíveis prontas.

e) Aumentando a segurança da aplicação disponibilizando os erros nos logs.

8.3 BLOCO TRY-CATCH E ALTERAÇÃO DO FLUXO NORMAL DE UM PROGRAMA

O programador pode definir um tratamento para qualquer tipo de erro de execução. Antes de definir o tratamento propriamente, é necessário determinar o trecho de código que pode gerar um erro na execução. Isso tudo é feito com o comando `try-catch` .

```
try {
    // trecho que pode gerar um erro na execução.
} catch (Throwable t) { // pegando todos os possíveis erros de
                        //execução.
    // tratamento para o possível erro de execução.
}
```

A sintaxe do `try-catch` tem um bloco para o programador

definir o trecho de código que *pode* gerar um erro de execução. Esse bloco é determinado pela palavra `try`. O programador também pode definir quais tipos de erro ele quer pegar para tratar. Isso é determinado pelo argumento do `catch`. Por fim, o tratamento é definido pelo bloco que é colocado após o argumento do `catch`.

Durante a execução, se um erro acontecer, a JVM redireciona o fluxo de execução da linha do bloco do `try` que gerou o erro para o bloco do `catch`. Importante! As linhas do bloco do `try` abaixo daquela que gerou o erro não serão executadas.

Fazer um `catch` em `Throwable` não é uma boa prática, pois todos os erros possíveis são tratados pela aplicação. Porém, os `Error`s não deveriam ser tratados pela aplicação, já que são de responsabilidade da JVM. Assim, também não é boa prática dar `catch` em `Error`s.

Modificando o argumento do `catch`, o programador define quais erros devem ser pegos para serem tratados.

```
try {
    // trecho que pode gerar um erro na execução.
} catch (Exception e) { // pegando todas as exceptions.
    // tratamento para o possível erro de execução.
}
```

Para a prova, é fundamental saber quando o programador pode ou não pode usar o `try-catch`. A única restrição de uso do `try-catch` envolve as checked exceptions.

Qual é a regra? O programador só pode usar `try-catch` em uma checked exception se o código do bloco do `try` pode realmente lançar a checked exception em questão.

```
try {
    System.out.println("não acontece SQLException");
} catch(java.sql.SQLException e){
// pegando SQLException, erro de compilação.
    // tratamento de SQLException.
}
```

Esse código **não** compila, pois o trecho envolvido no bloco do `try` nunca geraria a checked `SQLException`. O compilador avisa com um erro de "unreachable code". Já o exemplo a seguir compila, pois pode ocorrer um `FileNotFoundException`:

```
try {
    new java.io.FileInputStream("a.txt");
} catch(java.io.FileNotFoundException e){
    // tratamento de FileNotFoundException.
}
```

O código a seguir não tem nenhum problema, pois o programador pode usar o `try-catch` em qualquer situação para os erros de execução que não são checked exceptions.

```
try {
    System.out.println("Ok");
} catch (RuntimeException e) { // pegando RuntimeException
                               // (unckecked).
    // tratamento.
}
```

Quando a exception é pega, o fluxo do programa é sair do bloco `try` e entrar no bloco `catch`, portanto, o código a seguir imprime `peguei` e `continuando normal`:

```
String name = null;
try {
    name.toLowerCase();
    System.out.println("segunda linha do try");
} catch(NullPointerException ex) {
    System.out.println("peguei");
}
System.out.println("continuando normal");
```

Mas, se a exception que ocorre não é a que foi definida no catch, a chamada do método para e volta, jogando a exception como se não houvesse um `try/catch`. O cenário a seguir demonstra essa situação e não imprime nada:

```
String name = null;
try {
    name.toLowerCase();
    System.out.println("segunda linha do try");
} catch(IndexOutOfBoundsException ex) {
    System.out.println("peguei");
}
System.out.println("continuando normal");
```

Lembre-se sempre do polimorfismo. Portanto, pegar `IOException` é o mesmo que pegar todas as filhas de `IOException` também. O código a seguir trata o caso de o arquivo não existir além de todas as outras filhas de `IOException`:

```
try {
    new java.io.FileInputStream("a.txt");
} catch(java.io.IOException e){
    // tratamento de IOException.
}
```

Bloco finally

Tem coisas que não podemos deixar de fazer em hipótese alguma. Seja no sucesso ou no fracasso, temos obrigação de cumprir com algumas tarefas.

Imagine um método que conecta com um banco de dados. Não importa o que aconteça, no fim desse método a conexão deveria ser fechada. Durante a comunicação com o banco de dados, há o risco de ocorrer uma `SQLException`.

```
void method(){
    try {
        abreConexao();
        fazConsultas();
        fechaConexao();
    } catch (SQLException e) {
        // tratamento de SQLException
    }
}
```

Nesse código, há um grande problema: se um `SQLException` ocorrer durante as consultas, a conexão com o banco de dados não será fechada. Para tentar resolver esse problema, o bloco do `catch` poderia invocar o método `fechaConexao()`. Então, se acontecesse um `SQLException`, o bloco do `catch` seria executado e, consequentemente, a conexão seria fechada.

Mas ainda não solucionamos o problema, pois outro tipo de erro poderia acontecer nas consultas. Por exemplo, uma `NullPointerException` que não está sendo tratada. Para resolver o problema de fechar a conexão, um outro recurso do Java será usado, o bloco **finally**. Esse bloco é sempre executado, tanto no sucesso quanto no fracasso por qualquer tipo de erro.

```
void method(){
    try {
        abreConexao();
        fazConsultas();
        // Não precisa mais fechar a conexao aqui
    } catch(SQLException e) {
        // tratamento de SQLException
    } finally {
        fechaConexao(); // independetemente de sucesso ou
                        // fracasso, fecha a conexão
    }
}
```

Para melhor entender o fluxo do `try-cacth` com o `finally`, veja o próximo exemplo.

```
class A {
    void method() {
        try{
            //A
            //B
        }catch(SQLException e){
            //C
        }finally{
            //D
        }
        //E
    }
}
```

- Em uma execução normal, sem erros nem exceções, ele executaria `A`, `B`, `D`, `E`.
- Com `SQLException` em `A`, ele executaria `C`, `D`, `E`.
- Com `NullPointerException` em `A`, ele executaria apenas `D` e sairia.
- Se `A` fosse um `System.exit(0);`, ele apenas executa `A` e encerra o programa.
- Se ocorresse um erro `A`, executaria apenas `D` (dependendo do erro).

Uma outra maneira um pouco menos convencional de usar o `finally` é sem o bloco `catch`, como no exemplo a seguir.

```
class A{
  void method() {
    try {
      System.out.println("oi");
    } finally {
      // fecha algo
    }
  }
}
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String name;
        try {
            name.toLowerCase();
            System.out.println("a");
        } catch(NullPointerException ex) {
            System.out.println("b");
        }
        System.out.println("c");
    }
}
```

a) Não compila.

b) Compila e, ao rodar, imprime `"abc"`.

c) Compila e, ao rodar, imprime `"bc"`.

d) Compila e, ao rodar, imprime `"a"`.

e) Compila e, ao rodar, imprime `"b"`.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String name = null;
        try {
            name.toLowerCase();
            System.out.println("a");
        } catch(NullPointerException ex) {
            System.out.println("b");
        }
        System.out.println("c");
    }
}
```

a) Não compila.

b) Compila e, ao rodar, imprime `"abc"`.

c) Compila e, ao rodar, imprime `"bc"`.

d) Compila e, ao rodar, imprime `"a"`.

e) Compila e, ao rodar, imprime `"b"`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        String name = null;
        try {
            name.toLowerCase();
            System.out.println("a");
        } catch(NullPointerException ex) {
            System.out.println("b");
        } finally {
            System.out.println("c");
        }
        System.out.println("d");
    }
}
```

a) Compila e, ao rodar, imprime `"abcd"`.

b) Compila e, ao rodar, imprime `"bcd"`.

c) Compila e, ao rodar, imprime `"bc"`.

d) Compila e, ao rodar, imprime `"ad"`.

e) Compila e, ao rodar, imprime `"bd"`.

f) Não compila.

8.4 INVOQUE UM MÉTODO QUE JOGA UMA EXCEÇÃO

Eventualmente, um método qualquer não tem condição de tratar um determinado erro de execução. Nesse caso, esse método pode deixar passar o erro para o próximo método na pilha de execução.

Para deixar passar qualquer erro de execução que não seja uma checked exception, é muito simples: basta não fazer nada.

```
class Test {

    void method1(){
        System.out.println("primeiro antes");
        this.method2();
        System.out.println("primeiro depois");
    }

    void method2() {
        String s = null;
        System.out.println("segundo antes");
        s.length();
        System.out.println("segundo depois");
    }
}
```

O segundo método declara uma variável não primitiva e a inicializa com `null`. Logo em seguida, ele utiliza o operador `.` em uma referência que sabemos estar nula. Nesse ponto, na hora da execução, um `NullPointerException` é gerado.

Perceba que não há `try-catch` no segundo método, então ele não está pegando e tratando o erro, mas sim deixando-o passar. O primeiro método não define o `try-catch`, ou seja, também deixa passar o `NullPointerException`. O resultado é a impressão de `primeiro antes`, `segundo antes`.

Agora, para deixar passar uma checked exception, o método é obrigado a deixar explícito (avisado) que pretende deixar passar. Na assinatura do método, o programador pode deixar avisado que pretende deixar passar determinados erros de execução. Isso é feito através da palavra-chave `throws`.

```
class Test {

    void method1(){
        try {
            System.out.println("primeiro antes");
            this.method2();
            System.out.println("primeiro depois");
        } catch(IOException e) {
            // tratamento.
            System.out.println("primeiro catch");
        }
        System.out.println("primeiro fim");
    }

    void method2() throws IOException {
        System.out.println("segundo antes");
        System.in.read(); // pode lançar IOException
        System.out.println("segundo depois");
    }
}
```

O segundo método invoca o `read()` no `System.in`. Essa invocação pode gerar um `IOException`, de modo que o segundo método tem duas alternativas: ou pega e trata o possível erro ou o deixa passar. Para deixar passar, o comando `throws` deve ser utilizado na sua assinatura do segundo método. Isso indicará que um `IOException` pode ser lançado.

Dessa forma, o primeiro método que invoca o segundo pode receber uma `IOException`. Então, ele também tem duas escolhas: ou pega e trata usando `try-catch`, ou deixa passar usando o `throws`. O resultado é a impressão de `primeiro antes`,

`segundo antes`, `primeiro catch` e `primeiro fim`.

Caso a chamada a `System.in.read` fosse um sucesso, o resultado seria o caminho padrão de execução: `primeiro antes`, `segundo antes`, `segundo depois`, `primeiro depois` e `primeiro fim`.

Gerando um erro de execução

Qualquer método, ao identificar uma situação errada, pode criar um erro de execução e lançar para quem o chamou. Vale lembrar de que os erros de execução são representados por objetos criados a partir de alguma classe da hierarquia da classe `Throwable`, logo, basta o método instanciar um objeto de qualquer uma dessas classes e depois lançá-lo.

Se o erro não for uma checked exception, basta criar o objeto e utilizar o comando `throw` para lançá-lo na pilha de execução (não confunda com o `throws`):

```
class Test {

    void method1(){
        try {
            this.method2();
        } catch (RuntimeException e) {
            // tratamento.
        }
    }

    void method2() {
        throw new RuntimeException();
    }
}
```

Se o erro for uma checked exception, é necessário também declarar na assinatura do método o comando `throws`:

```
class Test {

    void method1(){
        try {
            this.method2();
        } catch(Exception e) {
            // tratamento.
        }
    }

    void method2() throws Exception {
        throw new Exception();
    }
}
```

Tome cuidado, somente instanciar a `Exception` não implica em jogá-la. Logo, o código a seguir mostra uma situação na qual não será impresso `error`:

```
class Test {

    void method1(){
        try {
            this.method2();
        } catch(Exception e) {
            System.err.println("error");
        }
    }

    void method2() throws Exception {
        new Exception(); // throw?????
    }
}
```

Podemos ainda criar nossas próprias exceções, bastando criar uma classe que entre na hierarquia de `Throwable`.

```
class FundoInsuficienteException extends Exception{}
```

Em qualquer lugar do código, é opcional o uso do `try` e `catch` de uma unchecked exception para compilar o código. Em uma checked exception, é obrigatório o uso do `try/catch` ou

`throws`.

O exemplo a seguir mostra uma unchecked exception sendo ignorada e o erro vazando. Assim, `finishing main` não será impresso, mas a aplicação morrerá com uma mensagem de erro da `Exception` que ocorreu.

```
public class Test {

    public static void main(String[] args) {
        method();
        System.out.println("finishing main");
    }

    private static void method() {
        int[] i= new int[10];
        System.out.println(i[15]);
        System.out.println("after i[15]");
    }

}
```

Ao pegarmos a exception, será impresso também `"finishing main"` uma vez que, após o `catch`, o fluxo volta ao normal. O exemplo a seguir imprime `Exception caught` e `finishing main`.

```
public class Test {

    public static void main(String[] args) {
        try {
            method();
        } catch(RuntimeException ex) {
            System.out.println("Exception caught");
        }
        System.out.println("finishing main");
    }

    private static void method() {
        int[] i= new int[10];
        System.out.println(i[15]);
```

```
        System.out.println("after i[15]");
    }

}
```

Podemos ter também múltiplas expressões do tipo `catch`. Nesse caso, será invocada somente a cláusula adequada, e não as outras. No código a seguir, se o `method2` jogar uma `ArrayIndexOutOfBoundsException`, será impresso `runtime`:

```
void method1() {
    try {
        method2();
    } catch(IOException ex) {
        System.out.println("io");
    } catch(RuntimeException ex) {
        System.out.println("runtime");
    } catch(Exception ex) {
        System.out.println("any other exception");
    }
}
```

E a ordem faz diferença? Sim, o Java procura o primeiro `catch` que pode trabalhar a `Exception` adequada.

Repare que `RuntimeException` herda de `Exception` e, portanto, deve vir antes da mesma na ordem de *catches*.

Caso ela viesse depois, ela nunca seria invocada, pois o Java verificaria que toda `RuntimeException` é `Exception` e `Exception` teria tratamento de preferência (por sua ordem). O exemplo a seguir não compila por este motivo:

```
void method1() {
    try {
        method2();
    } catch(IOException ex) {
        System.out.println("io");
    } catch(Exception ex) {
        System.out.println("any other exception");
```

```
    } catch(RuntimeException ex) {
        System.out.println("runtime"); // compile error
    }
}
```

Cuidado também com exceptions nos inicializadores. O código a seguir não compila, uma vez que durante a inicialização de um `FileAccess` pode ser jogado um `FileNotFoundException`, mas o construtor dessa classe (o padrão) não fala nada sobre isso.

```
class FileAccess {
    // compile error
    private InputStream is = new FileInputStream("input.txt");
}
```

Nesses casos, precisamos dizer no construtor que a `Exception` pode ser jogada ao instanciar um objeto do tipo `AcessoAoArquivo`:

```
class FileAccess {
    private InputStream is = new FileInputStream("input.txt");

    FileAccess() throws IOException{
    }
}
```

Exercícios

1) Qual classe podemos colocar no código a seguir para que ele compile? Existem 2 alternativas corretas.

```
import java.io.*;
class X {
    InputStream y() throws ??????????? {
        return new FileInputStream("a.txt");
    }
    void z() throws ??????????? {
        InputStream is = y();
        is.close();
    }
```

```
}
```

a) `java.io.IOException`

b) `java.sql.SQLException`

c) `java.lang.Exception`

d) `java.lang.IndexOutOfBoundsException`

e) `java.io.FileNotFoundException`

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    void m2() {
        System.out.println("e");
        int[][]x = new int[15][20];
        System.out.println("f");
    }
    void m() {
        System.out.println("c");
        m2();
        System.out.println("d");
    }
    public static void main(String[] args) {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    void m2() {
        System.out.println("e");
        int[] x = new int[15];
        x[20] = 13;
        System.out.println("f");
    }
    void m() {
        System.out.println("c");
        m2();
        System.out.println("d");
    }
    public static void main(String[] args) {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
class A {
```

```
    void m2() {
        System.out.println("e");
        new java.io.FileInputStream("a.txt");
        System.out.println("f");
    }
    void m() {
        System.out.println("c");
        m2();
        System.out.println("d");
    }
    public static void main(String[] args) {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
class A {
    void m2() throws java.io.FileNotFoundException {
        System.out.println("e");
        new java.io.FileInputStream("a.txt");
        System.out.println("f");
    }
    void m() throws java.io.IOException {
        System.out.println("c");
        m2();
        System.out.println("d");
    }
```

```
    public static void main(String[] args)
                throws java.io.FileNotFoundException {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
class A {
    void m2() throws java.io.FileNotFoundException {
        System.out.println("e");
        new java.io.FileInputStream("a.txt");
        System.out.println("f");
    }
    void m() throws java.io.FileNotFoundException {
        System.out.println("c");
        m2();
        System.out.println("d");
    }
    public static void main(String[] args) throws
    java.io.IOException {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
class A {
    void m2() throws java.io.FileNotFoundException {
        System.out.println("e");
        new java.io.FileInputStream("a.txt");
        System.out.println("f");
    }
    void m() throws java.io.FileNotFoundException {
        System.out.println("c");
        try {
            m2();
        } catch(java.io.FileNotFoundException ex) {
        }
        System.out.println("d");
    }
    public static void main(String[] args) throws
    java.io.IOException {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `acedb`.

f) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

8) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
class MyException extends RuntimeException {

}
class A {
    void m2() throws java.io.FileNotFoundException {
        System.out.println("e");
        new MyException();
        System.out.println("f");
    }
    void m() throws java.io.FileNotFoundException {
        System.out.println("c");
        try {
            m2();
        } catch(java.io.FileNotFoundException ex) {
        }
        System.out.println("d");
    }
    public static void main(String[] args) throws
    java.io.IOException {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

9) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
class MyException extends RuntimeException {

}
class A {
    void m2() throws java.io.FileNotFoundException {
        System.out.println("e");
        throw new MyException();
        System.out.println("f");
    }
    void m() throws java.io.FileNotFoundException {
        System.out.println("c");
        try {
            m2();
        } catch(java.io.FileNotFoundException ex) {
        }
        System.out.println("d");
    }
    public static void main(String[] args) throws
    java.io.IOException {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

10) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
class MyException extends RuntimeException {

}
class A {
    void m2() throws java.io.FileNotFoundException {
        System.out.println("e");
        boolean sim = true;
        if(sim) throws new MyException();
        System.out.println("f");
    }
    void m() throws java.io.FileNotFoundException {
        System.out.println("c");
        try {
            m2();
        } catch(java.io.FileNotFoundException ex) {
        }
        System.out.println("d");
    }
    public static void main(String[] args) throws
    java.io.IOException {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime

`db`.

11) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir, sem que o arquivo exista?

```
class MyException extends RuntimeException {

}
class A {
    void m2() throws java.io.FileNotFoundException {
        System.out.println("e");
        boolean sim = true;
        if(sim) throw new MyException();
        System.out.println("f");
    }
    void m() throws java.io.FileNotFoundException {
        System.out.println("c");
        try {
            m2();
        } catch(java.io.FileNotFoundException ex) {
        }
        System.out.println("d");
    }
    public static void main(String[] args) throws
    java.io.IOException {
        System.out.println("a");
        new A().m();
        System.out.println("b");
    }
}
```

a) Não compila.

b) Compila e imprime `acefdb`.

c) Compila e imprime `ace` e joga uma `Exception`.

d) Compila e imprime `acedb` e joga uma `Exception`.

e) Compila e imprime `ace`, joga uma `Exception` e imprime `db`.

8.5 RECONHEÇA CLASSES DE EXCEÇÕES COMUNS E SUAS CATEGORIAS

Para a prova, é necessário conhecer algumas exceptions clássicas do Java. Na sequência, vamos conhecer essas exceptions e entender em que situações elas ocorrem.

ArrayIndexOutOfBoundsException e IndexOutOfBoundsException

Um `ArrayIndexOutOfBoundsException` ocorre quando se tenta acessar uma posição que não existe em um array.

```
class Test {
    public static void main(String[] args) {
        int[] array = new int[10];
        array[10] = 10; // ArrayIndexOutOfBoundsException.
    }
}
```

Da mesma maneira, quando tentamos acessar uma posição não existente em uma lista, a exception é diferente, no caso `IndexOutOfBoundsException`:

```
class Test {
    public static void main(String[] args) {
        ArrayList<String> lista = new ArrayList<String>();

        String valor = lista.get(2); // IndexOutOfBoundsException
    }
}
```

NullPointerException

Toda vez que o operador `.` é usado em uma referência nula, um `NullPointerException` é lançado.

```
class Test {
    public static void main(String[] args) {
        String s = null;
        s.length(); // NullPointerException
    }
}
```

ClassCastException

Quando é feito um casting em uma referência para um tipo incompatível com o objeto que está na memória em tempo de execução, ocorre um `ClassCastException` .

```
class Test {
    public static void main(String[] args) {
        Object o = "SCJP"; // String
        Integer i = (Integer)o; // ClassCastException.
    }
}
```

NumberFormatException

Um problema comum que o programador enfrenta no dia a dia é ter de "transformar" texto em números. A API do Java oferece diversos métodos para tal tarefa. Porém, em alguns casos não é possível "parsear" o texto, pois ele pode conter caracteres incorretos.

```
class Test {
    public static void main(String[] args) {
        String s = "ABCD1";

        int i =    Integer.parseInt(s); // NumberFormatException
    }
}
```

IllegalArgumentException

Qualquer método deve verificar se os valores passados nos seus parâmetros são válidos. Se um método constata que os parâmetros estão inválidos, ele deve informar quem o invocou que há problemas nos valores passados na invocação. Para isso, é aconselhado que o método lance `IllegalArgumentException`.

```
class Test {
    public static void main(String[] args) {
        try {
            divideAndPrint(5,0);
        } catch (IllegalArgumentException e) {
            System.err.println("illegal!");
        }
    }

    public static void divideAndPrint(int i, int j) {
        if(j == 0) { // division by 0 ???
            throw new IllegalArgumentException();
        }
        System.out.println(i/j);
    }
}
```

IllegalStateException

Suponha que uma pessoa possa fazer três coisas: dormir, acordar e andar. Para andar, a pessoa precisa estar acordada. A classe `Person` modela o comportamento de uma pessoa. Ela contém um atributo `boolean` que indica se a pessoa está acordada ou dormindo, e um método para cada coisa que uma pessoa faz (`sleep()` , `wakeup()` e `walk()`).

O método `walk()` não pode ser invocado enquanto a pessoa está dormindo. Mas, se for, ele deve lançar um erro de execução. A biblioteca do Java já tem uma classe pronta para essa situação, a classe é a `IllegalStateException` . Ela significa que o estado atual do objeto não permite que o método seja executado.

```
class Person {
    boolean sleeping = false;

    void sleep() {
        this.sleeping = true;
        System.out.println("sleeping...");
    }
    void wakeup() {
        this.sleeping = false;
        System.out.println("waking up...");
    }
    void walk() {
        if(this.sleeping) {
            throw new IllegalStateException("Sleeping!!");
        }

        System.out.println("walking...");
    }
}
```

ExceptionInInitializerError

No momento em que a máquina virtual é disparada, ela não carrega todo o conteúdo do *classpath*. Em outras palavras, ela não carrega em memória todas as classes referenciadas pela sua aplicação.

Uma classe é carregada no momento da sua primeira utilização. Isso se dá quando algum método estático ou atributo estático são acessados, ou quando um objeto é criado a partir da classe em questão.

No carregamento de uma classe, a JVM pode executar um trecho de código definido pelo programador. Esse trecho deve ficar no que é chamado bloco estático.

```
class A {
    static {
        // trecho a ser executado no carregamento da classe.
```

```
    }
}
```

É totalmente possível que algum erro de execução seja gerado no bloco estático. Se isso acontecer, a JVM vai "embrulhar" esse erro em um `ExceptionInInitializerError` e dispará-lo.

Esse erro pode ser gerado também na inicialização de um atributo estático se algum problema ocorrer. Exemplo:

```
class A {
    static {
        if(true)
            throw new RuntimeException("tchau...");
    }
}

public class Test {

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

Gera o erro de inicialização:

```
Exception in thread "main" java.lang.ExceptionInInitializerError
    at Test.main(Test.java:11)
Caused by: java.lang.RuntimeException: tchau...
    at A.<clinit>(Test.java:4)
    ... 1 more
```

O exemplo a seguir mostra outra variação de `ExceptionInInitializerError`:

```
class P{
    static int a = Integer.parseInt("a");
}
```

StackOverflowError

Todos os métodos invocados pelo programa Java são empilhados na **Pilha de Execução**. Essa pilha tem um limite, ou seja, ela pode estourar:

```
class Test {
    public static void main(String[] args) {
        loopMethod();
    }

    static void loopMethod() {
        loopMethod();
    }
}
```

Repare que, nesse exemplo, o `loopMethod()` chama ele mesmo (recursão), que chama ele mesmo, que chama ele mesmo (loop eterno). Do jeito que está, os métodos serão empilhados eternamente e a pilha de execução vai estourar.

NoClassDefFoundError

Na etapa de compilação, todas as classes referenciadas no código-fonte precisam estar no *classpath*. Na etapa de execução também. O que será que acontece se uma classe está no *classpath* na compilação, mas não está na execução? Quando isso acontecer, será gerado um `NoClassDefFoundError` .

Para gerá-lo, podemos criar um arquivo com duas classes onde uma referencia a outra:

```
class AnotherClass {

}
class Test {
    public static void main(String[] args) {
        new AnotherClass();
    }
}
```

Compilamos o arquivo, gerando dois arquivos `.class`. Aí apagamos o arquivo `AnotherClass.class`. Pronto, o Java não será capaz de encontrar a classe, dando um erro, `NoClassDefFoundError`.

OutOfMemoryError

Durante a execução de nosso código, o Java vai gerenciando e limpando a memória usada por nosso programa automaticamente, usando o **garbage collector** (GC). O GC vai remover da memória todas as referências de objetos que não são mais utilizados, liberando o espaço para novos objetos.

Mas o que acontece quando criamos muito objetos, e não os liberamos? Nesse cenário, o GC não vai conseguir liberar memória, e eventualmente a memória livre vai acabar, ocasionando um `OutOfMemoryError`.

O código para fazer um erro do gênero é simples, basta instanciar infinitos objetos, sem permitir que o garbage collector jogue-os fora. Fazemos isso com `String`s para que o erro aconteça logo:

```
void method() {
    ArrayList<String> objetos = new ArrayList<String>();
    String atual = "";
    while(true) {
        atual += " ficou maior";
        objetos.add(atual);
    }
}
```

Exercícios

1) Escolha a opção adequada que indica o `Throwable` que

ocorrerá no código a seguir:

```
class A {
    public static void main(String[] args) {
        main(args);
    }
}
```

a) `IndexOutOfBoundsException`

b) `ArrayIndexOutOfBoundsException`

c) `NullPointerException`

d) `OutOfMemoryError`

e) `StackOverflowError`

f) `ExceptionInInitializationError`

2) Escolha a opção adequada que indica o `Throwable` que ocorrerá no código a seguir:

```
import java.util.*;
class A {
    public static void main(String[] args) {
        ArrayList<String> strings = new ArrayList<String>();
        for(int i=0;i<10;i++)
            for(int j=0;j<10;i++)
                strings.add("string " + i + " " + j);
        System.out.println(strings.get(99999));
    }
}
```

a) `IndexOutOfBoundsException`

b) `ArrayIndexOutOfBoundsException`

c) `NullPointerException`

d) `OutOfMemoryError`

e) `StackOverflowError`

f) `ExceptionInInitializationError`

Capítulo 9

JAVA 8 — JAVA BASICS

9.1 RODAR UM PROGRAMA JAVA A PARTIR DA LINHA DE COMANDO

A seção *Crie aplicações Java executáveis com um método main* foi renomeada para *Crie aplicações Java executáveis com um método main, e rode um programa Java a partir da linha de comando*. O conteúdo não sofre alterações.

9.2 TRABALHANDO COM SAÍDA NO CONSOLE

Imprimindo no console padrão

Para imprimir na saída do console, podemos usar o método `print*` do objeto `out`, da classe `System`:

```
System.out.print("hello world");
```

SYSTEM.OUT É UM PRINTSTREAM

O `out` é um atributo estático e público da classe `System`, do tipo `PrintStream`. A classe `PrintStream` possui vários métodos que permitem escrever diversos tipos de dados de maneira padronizada em um `OutputStream`.

O método `print` possui várias sobrecargas, recebendo desde *strings* até tipos primitivos:

```
System.out.print(false); // a boolean
System.out.print(10.3);  // a double value
System.out.print("Some text"); // some text
```

Também existe uma versão de `print` que recebe `Object`, ou seja, qualquer objeto. Neste caso, será invocado o método `toString` do objeto passado:

```
class Test {

    public String toString(){
        return "My test object...";
    }

    public static void main(String[] args){
        System.out.print(new Test()); // My test object...
    }
}
```

Existe um único caso especial que foge à regra; além de primitivos, *strings* e `Object`, existe também uma sobrecarga que recebe um array de `char`. Todos os outros *arrays* são tratados como `Object`

```
char[] c = {'a','b','c'};
```

```
int[] i = {1,2,3};
System.out.print(c); // abc
System.out.print(i); // [I@9d8643e (ou similar)]
```

O método `print` apenas imprime o valor passado. Se invocado várias vezes em sequência, todos os valores serão impressos em uma única linha:

```
System.out.print(false);
System.out.print(10.3);
System.out.print("Some text");
```

Esse código imprime:

```
false10.3Some text
```

Podemos imprimir cada conteúdo em uma linha diferente, concatenando um `"\n"` após cada impressão. Para este caso, existe o método `println`, que adiciona uma quebra de linha após cada chamada.

```
System.out.println(false);
System.out.println(10.3);
System.out.println("Some text");
```

Já esse código imprime:

```
false
10.3
Some text
```

O `println` possui as mesmas sobrecargas que o método `print`, além de uma versão sem nenhum parâmetro, que apenas quebra uma linha:

```
System.out.print("foo");
System.out.println(); // line break
System.out.print("bar");
```

Isso imprime:

```
foo
bar
```

Formatando a impressão

Na versão 5 do Java, foram incluídos dois métodos para permitir a impressão no console de modo formatado, `format` e `printf`. Ambos se comportam exatamente da mesma maneira, sendo que o `printf` foi incluído provavelmente apenas para manter uma sintaxe próxima à da linguagem `C`, na qual existe um método com comportamento similar com este nome. Sendo assim, tudo o que discutirmos sobre um método serve para o outro.

O `printf` recebe dois parâmetros: o primeiro é uma `String` que pode conter apenas texto normal ou incluir caracteres especiais de formatação; o segundo é um `varargs` de objetos a serem usados na impressão. Vamos começar com um exemplo simples:

```
System.out.printf("Hello %s, have a nice day!", "Mario");
```

Esse código imprime:

```
Hello Mario, have a nice day!
```

Perceba o caractere `%` na mensagem. É ele que indica que ali temos uma formatação especial. O `s` logo em seguida ao `%` indica que lá incluiremos alguma outra `String`, que é a que passamos como segundo argumento deste método.

O próprio método se encarrega de fazer a concatenação no lugar certo e exibir o resultado no console, o que chamamos de interpolação de `String` (*string interpolation*). Para indicar como a formatação deve ser feita, usamos a seguinte estrutura, que

veremos com mais detalhes a partir de agora:

```
%[index$][flags][width][.precision]type
```

Todas as opções entre `[ ]` são opcionais. Somos obrigados a informar apenas o caractere de `%` e o tipo do argumento que será concatenado. Vamos analisar cada opção separadamente:

O **type** é o tipo de argumento que será passado e suporta os seguintes valores:

- `b` — `boolean`
- `c` — `char`
- `d` — Números inteiros
- `f` — Números decimais
- `s` — `String`
- `n` — Quebra de linha

Vejamos alguns exemplos:

```
System.out.printf("%s %n", "foo"); //foo
System.out.printf("%b %n", false); //false
System.out.printf("%d %n", 42);    //42
System.out.printf("%d %n", 1024L); //1024
System.out.printf("%f %n", 23.9f); //23.900000
System.out.printf("%f %n", 44.0);  //44.000000
```

Repare que podemos passar mais de uma instrução por impressão, sendo que cada `%` está relacionado com o próximo parâmetro passado na sequência:

```
System.out.printf("%s, it's %b, the result is %d", "yes",
                  true, 100);
```

Esse código imprime:

```
yes, it's true, the result is 100
```

O **index** é um número inteiro delimitado pelo caractere `$`, que indica qual dos argumentos deve ser impresso nessa posição se desejarmos fugir do padrão sequencial. Por exemplo:

```
System.out.printf("%2$s %1$s", "World","Hello"); // Hello World
```

Mesmo passando os argumentos fora de ordem, conseguimos indicar na *string* qual a ordem que queremos na impressão. Esse recurso é extremamente interessante quando queremos repetir o mesmo valor em dois pontos da nossa formatação.

O **width** indica a quantidade mínima de caracteres para imprimir. Completa com espaços à esquerda caso o valor seja menor que a largura mínima. Caso seja maior, não faz nada:

```
System.out.printf("[%5d]%n", 22);       //[   22]
System.out.printf("[%5s]%n", "foo");    //[  foo]
System.out.printf("[%5s]%n", "foofoo"); //[foofoo]
```

As **flags** são caracteres especiais que alteram a maneira como a impressão é feita. Para a prova, é importante conhecer alguns, dentre os quais os dois que indicam se o número é positivo ou negativo:

- `+` — Sempre inclui um sinal de positivo (+) ou negativo (-) em números.
- `(` — Números negativos são exibidos entre parênteses.

Dois de alinhamento à esquerda ou direita:

- `-` — Alinha à esquerda. Precisa de tamanho para ser usado.
- `0` — Completa a esquerda com zeros. Precisa de tamanho para ser usado.

Juntamente com o tamanho mínimo, podemos usar a *flag* de alinhamento e de completar com zeros:

```
System.out.printf("[%05d]%n", 22);    //[00022]
System.out.printf("[%-5s]%n", "foo"); //[foo  ]
```

Só é possível completar com zeros quando estamos formatando números. Tentar usar esta flag com *strings* lança uma `FormatFlagsConversionMismatchException`, como em:

```
System.out.printf("[%05s]%n", "foo");
```

Temos uma flag para separar casa de milhares e decimais:

- `,` — Habilita separadores de milhar e decimal.

Vamos ver alguns exemplo:

```
System.out.printf("%+d %n", 22);       //+22
System.out.printf("%,f %n", 1234.56);  //1,234.560000
System.out.printf("%(f %n", -1234.56); //(1234.560000)
```

A impressão de separadores de milhar e decimal depende da linguagem do sistema onde o código é executado. Podemos alterar o padrão usando um objeto da classe `Locale`, que altera as regras padrões para as regras da língua informada:

```
Locale br = new Locale("pt","BR");
System.out.printf(br,"%,f %n", 123456.789); // 123.456,789000
```

Quando da formatação de números com casas decimais, **precision** indica quantas casas queremos depois da vírgula. Basta usar um `.` seguido do número de caracteres. Vale lembrar de que só é possível mudar a precisão quando estamos formatando números decimais.

```
System.out.printf("[%.2f]%n", 22.5); //[22.50]
```

Exercícios

1) O que acontece ao tentar compilar e executar o código a seguir?

```
class Test{
    public static void main(String[] args){
    System.out.print("a");
    System.out.println('b'); // A
    System.out.print();      // B
    System.out.println("c");
    }
}
```

a) Compila e imprime:

```
ab
c
```

b) Compila e imprime:

```
ab
c
```

c) Compila e imprime:

```
a
bc
```

d) Não compila por erro na linha **A**.

e) Não compila por erro na linha **B**.

2) Ao tentar compilar e executar o código a seguir, o que acontece?

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

        System.out.print("a");
        System.out.println("b");
```

```
        System.out.printf("c");
        System.out.print("d");
        System.out.println("\n");
        System.out.print("e");
    }
}
```

a) Compila e imprime:

```
ab
cd

e
```

b) Compila e imprime:

```
ab
c
d

e
```

c) Compila e imprime:

```
ab
cd
e
```

d) Não compila.

3) O que acontece ao compilar e executar o código a seguir?

```
public class Test {
    public static void main(String[] args) {
        System.out.printf("%s",12); //A
        System.out.printf("%d",new Integer(321)); //B
        System.out.printf("%d",(short)(byte)(double) 127); //C
    }
}
```

a) Erro de compilação na linha **B**.

b) Erro de compilação nas linhas **B** e **C**.

c) Compila e executa normalmente.

d) Erro de compilação na linha **C**.

e) Erro de compilação na linha **A**.

4) Qual dos códigos a seguir **não** imprime `>00012.45<` ? Escolha 1 alternativa.

a) `System.out.printf(">%0,8.2f<",12.45);`

b) `System.out.printf(">%-(8.2f<",12.45);`

c) `System.out.format(">%0(8.2f<",12.45);`

d) `System.out.format(">%1$08.2f<",12.45);`

e) `System.out.printf(">%0,(8.2f<",12.45);`

5) Ao tentar compilar e executar o seguinte código, o que é impresso?

```
public class Testes {
    public static void main(String[] args) {
        System.out.println(new char[]{'a','b','c'}); // A
        System.out.println(new byte[]{'a','b','c'}); // B
        System.out.println("abc"); // C
        System.out.println(new String[]{"abc"}); // D
    }
}
```

a) Não compila na linha **B**.

b) Linhas **A** e **C** imprimem `abc` .

c) Linha **C** imprime `abc` .

d) Linhas **A**, **C** e **D** imprimem `abc` .

e) Todas as linhas imprimem `abc`.

9.3 COMPARE E CONTRASTE AS FUNCIONALIDADES E COMPONENTES DA PLATAFORMA

Linguagem X plataforma

Quando pensamos na palavra **Java**, geralmente pensamos na linguagem, mas na verdade o Java é muito mais que isso. É toda uma plataforma de desenvolvimento, com características bem distintas que permitiram sua adoção em massa por empresas e desenvolvedores.

Quando escrevemos um programa na linguagem Java, para conseguirmos executá-lo, é necessário compilar o código. Uma das diferenças do Java para outras linguagens mais tradicionais é que o resultado da compilação não é exatamente código de máquina, que será interpretado pelo sistema operacional do computador.

O código Java compilado é chamado de **bytecode**. É um arquivo binário que possui a extensão `.class`. Esse arquivo será lido e interpretado pela **máquina virtual java**, ou simplesmente **JVM**.

A JVM é como um computador genérico. Seu papel é interpretar as instruções do bytecode e converter estas instruções para código de máquina, para ser executado pelo sistema operacional nativo do computador.

Existem implementações de JVM para os principais sistemas operacionais, além de diversos dispositivos, como máquinas

industriais e até mesmo geladeiras e televisões. Estas implementações são escritas e testadas de maneira que elas sempre interpretem o bytecode da mesma forma, ou seja, não existem diferenças entre as instruções, independente de qual a plataforma nativa. Isso torna a linguagem Java muito poderosa, pois permite que um programa escrito e compilado uma única vez possa ser executado em diversos sistemas operacionais e dispositivos, sem precisar de grandes adaptações.

Esse era até mesmo o lema comercial do Java, *Write Once and Run Anywhere*. Uma vez escrito e compilado o código, ele pode ser executado em qualquer ambiente que possua uma virtual machine do Java.

Portanto, note que o termo Java é usado em diversos contextos: a linguagem em si, um pacote de bibliotecas — Java Standard Edition *JSE*, Java Enterprise Edition *JEE* etc. — o compilador e a Java Virtual Machine. Tudo isso é o que chamamos de plataforma Java.

Orientação a Objetos

Java é uma linguagem de alto nível, orientada a objetos. Mas o que significa ser orientada a objetos?

Orientação a Objetos é um paradigma de programação, segundo o qual estruturamos nosso código em entidades conhecidas como **objetos**, que possuem dentro de si **dados** na forma de atributos, e **comportamento**, na forma de métodos.

Objetos são como pequenos componentes especializados em executar uma funcionalidade em um sistema. Inclusive, uma das

boas práticas da área é que cada objeto tenha uma única responsabilidade. Assim como no mundo real, para executarmos um determinado comportamento, precisamos da interação de vários objetos/componentes. Por serem componentes independentes e especializados, objetos favorecem o reúso de código e reduzem o custo de manutenção, já que a mudança no comportamento de um objeto será refletida em todos os lugares onde esse objeto é usado.

Encapsulamento

Um dos conceitos mais básicos e importantes da Orientação a Objetos é o de encapsulamento, a técnica de esconder atributos e detalhes de implementação de um objeto, para que quando eles sejam alterados, tal alteração não tenha de ser replicada em vários lugares do sistema.

Em Java, a forma mais simples de se obter um código encapsulado é declarando os atributos de uma classe como `private`, evitando com isso que outros objetos possam acessar e manipular estes atributos. Caso desejemos que outros objetos tenham acesso a estes dados, liberamos por meio de métodos, controlando como será feita a leitura e escrita das informações.

Vamos ver um exemplo simples, veja a classe `Person`:

```
public class Person{
    public String firstName;
    public String lastName;

    public Person(String firstName, String lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Desse jeito, qualquer classe no sistema consegue ver estes atributos, e ainda manipulá-los, mudar seus valores. Vamos começar a alterar este código, restringindo o acesso aos atributos usando o modificador de acesso `private` :

```
public class Person{
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
```

Ótimo, agora nossos atributos estão encapsulados. Imagine agora que uma outra classe precise imprimir o nome completo de uma pessoa. Vamos implementar esta funcionalidade:

```
//Person.java
public class Person{
    private String firstName;
    private String lastName;

    public Person(String firstName, String lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }

    // Expose the full name of the person,
    // but not how it's stored internally
    public String getFullName(){
        return this.firstName + " "
            + this.lastName;
    }
}

//Test.java

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

```
        Person p = new Person("Mario", "Amaral");
        System.out.print(p.getFullName());
    }
}
```

Se, por algum motivo, você resolver não armazenar mais o nome e o sobrenome em campos separados, a seguinte alteração será necessária:

```
//Person.java
public class Person{
    private String fullName;

    public Person(String firstName, String lastName){
        //storing in just one field.
        this.fullName = firstName + " " + this.lastName;
    }

    public String getFullName(){
        return this.fullName;
    }
}

//Test.java
class Test{
    public static void main(String[] args){
        Person p = new Person("Mario", "Amaral");
        System.out.print(p.getFullName());
    }
}
```

Repare que mudamos a maneira como armazenamos o nome em nossa classe `Person`, mas não precisamos fazer nenhuma alteração na classe `Test`. Este é um exemplo de um código bem encapsulado.

Alterações são realizadas apenas nas classes cuja implementação vai mudar, e não no sistema inteiro. O encapsulamento vai além de variáveis membro privadas. Veremos

detalhes de sua implementação mais à frente, em uma seção dedicada ao tema. Nessa seção da prova, será cobrada a vantagem e características dessa técnica.

Exercícios

1) Para aumentar o encapsulamento de uma classe, devemos:

a) Deixar os métodos e atributos privados.

b) Deixar os métodos e atributos públicos.

c) Deixar os métodos públicos e atributos públicos.

d) Deixar os métodos públicos e atributos privados.

e) Deixar os métodos privados e atributos públicos.

Capítulo 10

JAVA 8 — TRABALHANDO COM TIPOS DE DADOS EM JAVA

10.1 DESENVOLVER CÓDIGO QUE USA CLASSES WRAPPERS

Wrappers são classes de objetos que representam tipos primitivos. Existe um *wrapper* para cada primitivo, conforme a lista a seguir:

- `boolean : Boolean`
- `byte : Byte`
- `short : Short`
- `char : Character`
- `int : Integer`
- `long : Long`
- `float : Float`
- `double : Double`

Criando wrappers

Todos os *wrappers* numéricos possuem dois construtores: um

que recebe o tipo primitivo, e um que recebe `String`. O construtor que recebe `String` pode lançar `NumberFormatException`, se a `String` fornecida não puder ser convertida ao tipo em questão.

```
Double d1 = new Double(22.5);
Double d2 = new Double("22.5");
Double d2 = new Double("abc"); //throws NumberFormatException
```

A classe `Character` possui apenas um construtor, que recebe um `char` como argumento:

```
Character c = new Character('d');
```

A classe `Boolean` também possui dois construtores: um que recebe `boolean` e outro que recebe `String`. Caso a `String` passada como argumento tenha o valor `"true"`, com maiúsculas ou minúsculas, o resultado será `true`; qualquer outro valor resultará em `false`.

```
Boolean b1 = new Boolean(true);   // true
Boolean b2 = new Boolean("true"); // true
Boolean b3 = new Boolean("TrUe"); // true
Boolean b4 = new Boolean("T");    // false
```

Convertendo de wrappers para primitivos

Para converter um *wrapper* em um primitivo, existem vários métodos no formato `xxxValue()`, em que `xxx` é o tipo para o qual gostaríamos de realizar a conversão.

```
Long l = new Long("123");

byte b = l.byteValue();
double d = l.doubleValue();
int i = l.intValue();
short s = l.shortValue();
```

Todos os tipos numéricos podem ser convertidos entre si. Os tipos `Boolean` e `Character` só possuem método para converter para o próprio tipo primitivo:

```
boolean b = new Boolean("true").booleanValue();
char c = new Character('z').charValue();
```

Convertendo de String para wrappers ou primitivos

Além dos construtores dos *wrappers* que recebem `String` como parâmetro, também existem métodos para realizar transformações entre *strings*, *wrappers* e primitivos.

Vamos começar convertendo de *strings* para primitivos. Cada *wrapper* possui um método no formato `parseXXX`, em que `XXX` é o tipo do *wrapper*. Este método também lança `NumberFormatException` caso não consiga fazer a conversão:

```
double d = Double.parseDouble("23.4");
long l = Long.parseLong("23");
int i = Integer.parseInt("444");
```

Os *wrappers* de números inteiros possuem uma variação do `parseXXX` que recebe como segundo argumento a base a ser usada na conversão:

```
short i1 = Short.parseShort("11",10); // 11  Decimal
int i2 = Integer.parseInt("11",16);   // 17  HexaDecimal
byte i3 = Byte.parseByte("11",8);     // 9   Octal
int i4 = Integer.parseInt("11",2);    // 3   Binary
int i5 = Integer.parseInt("A",16);    // 10  HexaDecimal
int i6 = Integer.parseInt("FF",16);   // 255 HexaDecimal
```

Já para converter uma `String` diretamente para um *wrapper*, podemos ou usar o construtor como vimos anteriormente, ou usar o método `valueOf`, disponível em todos os *wrappers*. A assinatura destes métodos é idêntica à do `parseXXX`, inclusive

com versões que recebem a base de conversão para tipo inteiros:

```
Double d = Double.valueOf("23.4");
Long l = Long.valueOf("23");
Integer i1 = Integer.valueOf("444");
Integer i2 = Integer.valueOf("5AF", 16);
```

Convertendo de primitivos ou wrappers para String

Assim como todo objeto Java, os *wrappers* também possuem um método `toString` :

```
Integer i = Integer.valueOf(256);
String number = i.toString();
```

Além do `toString` padrão, há uma versão estática sobrecarregada, que recebe o tipo primitivo como argumento. Ademais, os tipos `Long` e `Integer` possuem uma versão que, além do primitivo, também recebem a base:

```
String d = Double.toString(23.5);
String s = Short.toString((short)23);
String i = Integer.toString(23);
String l = Long.toString(20, 16);
```

Além destes, as classes `Long` e `Integer` ainda possuem outros métodos para fazer a conversão direta para a base escolhida:

```
String binaryString = Integer.toBinaryString(8); //1000, binary
String hexString = Long.toHexString(11);       // B, Hexadecimal
String octalString = Integer.toOctalString(22);  // 26 Octal
```

Autoboxing

Até o Java 1.4, não era possível executar operações em cima de *wrappers*. Por exemplo, para somar 1 em um `Integer` , era necessário o seguinte código:

```
Integer intWrapper = Integer.valueOf(1);
int intPrimitive = intWrapper.intValue();
intPrimitive++;
intWrapper = Integer.valueOf(intPrimitive);
```

A partir do Java 5, foi incluído um recurso chamado **autoboxing**. O próprio compilador é responsável por transformar os *wrappers* em primitivos (*unboxing*) e primitivos em *wrappers* (*boxing*). A mesma operação de somar 1 em um `Integer` agora é:

```
Integer intWrapper = Integer.valueOf(1);
intWrapper++; //will unbox, increment, then box again.
```

Repara que não há magia. A única diferença é que, em vez de você mesmo escrever o código que faz o *boxing* e *unboxing*, agora esse código é gerado pelo compilador.

Comparando wrappers

Veja o seguinte código:

```
Integer i1 = 1234;
Integer i2 = 1234;
System.out.println(i1 == i2);      //false
System.out.println(i1.equals(i2)); //true
```

Apesar de parecer que estamos trabalhando com primitivos, estamos usando objetos aqui. Logo, quando esses objetos são comparados usando `==` , o resultado é `false` , já que são duas instâncias diferentes de `Integer` .

Agora veja o seguinte código:

```
Integer i1 = 123;
Integer i2 = 123;
System.out.println(i1 == i2);      //true
System.out.println(i1.equals(i2)); //true
```

Repare que o resultado do `==` foi `true`, mas o que aconteceu? É que o Java, para economizar memória, mantém um **cache** de alguns objetos e, toda vez que é feito um *boxing*, ele os reutiliza. Os seguintes objetos são mantidos no cache:

- Todos `Boolean` e `Byte`;
- `Short` e `Integer` de **-128** até **127**;
- `Caracter` ASCII, como letras, números etc.

Sempre que você encontrar comparações usando `==`, envolvendo *wrappers*, preste muita atenção aos valores: se forem baixos, é possível que o resultado seja `true` mesmo sendo objetos diferentes!

NullPointerException em operações envolvendo wrappers

Fique atento, pois, como *wrappers* são objetos, eles podem assumir o valor de `null`. Qualquer operação executada envolvendo um objeto `null` resultará em um `NullPointerException`:

```
Integer a = null;
int b = 44;
System.out.println(a + b); //throws NPE
```

Exercícios

1) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

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

```
        int i = 10;
        m1(i);
    }

    private static void m1(Integer j) {
        System.out.println("go!");
    }
}
```

a) Erro de compilação.

b) Imprime `go!`.

c) Imprime `10`.

d) Compila, mas lança exception.

2) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {

    static int i;

    public static void main(String[] args) {
        i = Integer.parseInt("10");
        m1(i + 1);
    }

    private static void m1(Integer j) {
        System.out.println(i);
    }
}
```

a) Erro de compilação.

b) Imprime `11`.

c) Imprime `10`.

d) Compila, mas lança exception.

3) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {

    static int i;

    public static void main(String[] args) {
        i = Integer.valueOf("10");
        m1(i + 1);
    }

    private static void m1(Integer j) {
        System.out.println(j);
    }
}
```

a) Imprime `10`.

b) Erro de compilação.

c) Compila, mas lança exception.

d) Imprime `11`.

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
public class Test {
    static long i;

    public static void main(String[] args) {
        i = Integer.valueOf("10",8); // A
        m1(i); // B
    }

    private static void m1(Integer j) { // C
        System.out.println(j);
    }
}
```

a) Imprime `10`.

b) Erro de compilação na linha **A**.

c) Erro de compilação na linha **B**.

d) Erro de compilação na linha **C**.

e) Compila, mas lança exception.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
public class Test {
    public static void main(String[] args) {
        int a = Short.parseShort("126"); // A
        short s = Integer.parseInt("23").shortValue(); //B
        double h = Double.valueOf("27").floatValue();  //C
        System.out.println(a + s);
    }
}
```

a) Imprime `149`.

b) Imprime `176`.

c) Erro de compilação na linha **A**.

d) Erro de compilação na linha **B**.

e) Erro de compilação na linha **C**.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
public class Test {

    public static void main(String[] args) {
        char c = new Character('x').charValue();
        System.out.println(c);
```

```
    }
}
```

a) Imprime `x`.

b) Imprime `120`.

c) Erro de compilação.

7) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
public class Test {

    public static void main(String[] args) {
        int a = Integer.parseInt("10",2);
        int b = a == 10 ? null : 3;
        System.out.println(a + b);
    }
}
```

a) Imprime `13`.

b) Imprime `5`.

c) Imprime `10`.

d) Erro de compilação.

e) Ao executar, lança `NullPointerException`.

CAPÍTULO 11

JAVA 8 — TRABALHANDO COM ALGUMAS CLASSES DA JAVA API

11.1 CRIE E MANIPULE DADOS DE CALENDÁRIOS

No Java 8, após uma longa história de sofrimento dos desenvolvedores com as classes `java.util.Date` e `java.util.Calendar`, foi introduzida uma nova API para manipulação de datas e horas. Vamos entender quais classes e métodos foram incluídos, além de passar pelos detalhes que serão cobrados na prova.

As classes que serão cobradas são:

- `LocalDate` : representa uma data sem hora no formato `yyyy-MM-dd` (ano-mês-dia).
- `LocalTime` : representa uma hora no formato `hh:mm:ss.zzz` (hora:minuto:segundo.milissegundo).
- `LocalDateTime` : representa uma data com hora no formato `yyyy-MM-dd-HH-mm-ss.zzz` (ano-mês-dia-hora-minuto-segundo.milissegundo).

- `MonthDay` : representa um dia e mês, sem o ano.
- `YearMonth` : representa um mês e ano, sem o dia.
- `Period` : representa um período de tempo, em dia, mês e ano.
- `DateTimeFormatter` : classe que possui vários métodos para formatação.

Para utilizar essas classes, é necessário conhecer uma ou outra classe da API de `java.time` que não estão na lista da seção, sendo que também as veremos aqui.

Todas as classes do pacote `java.time` são imutáveis, ou seja, após serem instanciadas, seus valores não podem ser alterados, assim como a classe `String` . Portanto, lembre-se de que todos os métodos que parecem modificar os valores das datas retornam novas instâncias com os valores alterados, enquanto o objeto original segue inalterado.

Criando objetos de data

Todas as classes que representam datas têm métodos similares para criação, como nos exemplos a seguir:

```
LocalTime currentTime = LocalTime.now(); // 09:05:03.244
LocalDate today = LocalDate.now(); // 2014-12-10
LocalDateTime now = LocalDateTime.now();
                        // 2014-12-10-09-05-03.244
```

É possível escolher o fuso horário que será usando na criação das datas, passando como parâmetro para o método `now` um objeto do tipo `ZoneId` :

```
LocalTime time = LocalTime.now(ZoneId.of("America/Chicago"));
LocalDate date = LocalDate.now(ZoneId.of("America/Sao_Paulo"));
LocalDateTime dateTime =
```

```
    LocalDateTime.now(ZoneId.of("America/Los_Angeles"));
```

Caso queira representar uma data ou hora específica, usamos o método `of`. Cada um desses métodos possui versões sobrecarregadas, recebendo mais ou menos valores iniciais. Todos os tipos de datas que contêm meses possuem versões de `of`, que recebem tanto números inteiros quanto valores do *enum* `Month`.

Por exemplo, podemos criar uma representação do meio-dia:

```
LocalTime noon = LocalTime.of(12, 0);
```

Para criar o natal de 2014 e de 2015:

```
LocalDate christmas2014 = LocalDate.of(2014, 12, 25);
LocalDate christmas2015 = LocalDate.of(2015, Month.DECEMBER, 25);
```

Podemos representar qualquer natal:

```
MonthDay someChristmas = MonthDay.of(Month.DECEMBER, 31);
```

Ainda com o método `of`, podemos criar um momento exato no tempo:

```
LocalDateTime someDate =
            LocalDateTime.of(2017, Month.JANUARY, 25, 13, 45);
```

Ou ainda passar um dia e somente adicionar o horário:

```
LocalDate christmas2014 = LocalDate.of(2014, 12, 25);
LocalDateTime christmasAtNoon =
    LocalDateTime.of(christmas2014, meioDia);
```

Passar um valor inválido para qualquer um dos campos (mês 13, por exemplo) lançará um `DateTimeException`.

Manipulando datas

Uma das decisões de *design* que guia a nova API de datas é a

padronização dos nomes de métodos que têm o mesmo comportamento. Os nomes mais comuns são:

- `get` : obtém o valor de algo;
- `is` : verifica se algo é verdadeiro;
- `with` : lembra um *setter*, mas retorna um novo objeto com o valor alterado;
- `plus` : soma alguma unidade ao objeto, retorna um novo objeto com o valor alterado;
- `minus` : subtrai alguma unidade do objeto, retorna um novo objeto com o valor alterado;
- `to` : converte um objeto de um tipo para outro;
- `at` : combina um objeto com outro.

Veremos agora esses métodos na prática.

Extraindo partes de uma data

Para obter alguma porção de uma data, podemos usar os métodos precedidos por `get` :

```
LocalDateTime now = LocalDateTime.of(2014,12,15,13,0);
System.out.println(now.getDayOfMonth()); // 15
System.out.println(now.getDayOfYear());  // 349
System.out.println(now.getHour());       // 13
System.out.println(now.getMinute());     // 0
System.out.println(now.getYear());       // 2014
System.out.println(now.getDayOfWeek());  // MONDAY
System.out.println(now.getMonthValue()); // 12
System.out.println(now.getMonth());      // DECEMBER
```

Além desses métodos, temos um método `get()` , que recebe como parâmetro uma implementação da interface `TemporalField` , geralmente `ChronoField` , e retorna um inteiro. Note que como estamos falando de um campo que será

retornado. Usamos `ChronoField`, um campo de tempo cujo valor queremos saber:

```
LocalDateTime now = LocalDateTime.of(2014,12,15,13,0);
// 15
System.out.println(now.get(ChronoField.DAY_OF_MONTH));
// 349
System.out.println(now.get(ChronoField.DAY_OF_YEAR));
// 13
System.out.println(now.get(ChronoField.HOUR_OF_DAY));
// 0
System.out.println(now.get(ChronoField.MINUTE_OF_HOUR));
// 2014
System.out.println(now.get(ChronoField.YEAR));
// 1 (MONDAY)
System.out.println(now.get(ChronoField.DAY_OF_WEEK));
// 12
System.out.println(now.get(ChronoField.MONTH_OF_YEAR));
```

Nem todas as classes possuem todos os métodos. Por exemplo, objetos do tipo `LocalDate` não possuem a parte de horas, logo não há métodos como `getHour`. É necessário ficar atento ao tipo do objeto e a qual método está sendo chamado:

```
LocalDate d = LocalDate.now();
d.getHour(); //compile error, method not found.
```

Comparações entre datas

Usamos os métodos que começam com `is` para realizar comparações entre as datas:

```
MonthDay day1 = MonthDay.of(1, 1); //01/jan
MonthDay day2 = MonthDay.of(1, 2); //02/jan

System.out.println(day1.isAfter(day2)); //false
System.out.println(day1.isBefore(day2)); //true
```

Além de métodos de comparação, também existem aqueles para indicar se alguma porção da data é suportada pelo objeto:

```
LocalDate aprilFools = LocalDate.of(2015, 4, 1);
LocalDate foolsDay = LocalDate.of(2015, 4, 1);
// are equals
System.out.println(aprilFools.isEqual(foolsDay)); //true
// does this object support days?
System.out.println(aprilFools.isSupported(
    ChronoField.DAY_OF_MONTH)); //true
// does this object supports hours?
System.out.println(aprilFools.isSupported(
    ChronoField.HOUR_OF_DAY)); //false
// Can I make operations with days?
System.out.println(aprilFools.isSupported(ChronoUnit.DAYS));
//true
// Can I make operations with hours?
System.out.println(aprilFools.isSupported(ChronoUnit.HOURS));
//false
```

Alterando as datas

Todos os objetos da nova API de datas são imutáveis, ou seja, não podem ter o seu valor alterado após a criação. Mas existem alguns que podem ser utilizados para obter versões modificadas destes objetos. Vamos começar com o método `with`, que é como um *setter*, mas retornando um novo objeto em vez de alterar o valor do objeto atual:

```
LocalDate d = LocalDate.of(2015, 4, 1); //2015-04-01

d = d.withDayOfMonth(15).withMonth(3); //chaining
System.out.println(d); //2015-03-15
```

Cada método `with` chamado retorna um novo objeto, com o valor modificado. O objeto original nunca tem seu valor alterado:

```
LocalDate d = LocalDate.of(2013, 9, 7);
System.out.println(d); // 2013-09-07
d.withMonth(12);
System.out.println(d); // 2013-09-07
```

Lembre-se de que só é possível manipular partes da data em

objetos que têm estas partes. O exemplo a seguir não compila, pois *"LocalTime does not have a day of month field"*.

```
LocalTime d = LocalTime.now();
d.withDayOfMonth(15); // compile error
```

Caso o objetivo seja incrementar ou decrementar alguma parte da data, temos os métodos `plus` e `minus`:

```
LocalDate d = LocalDate.of(2013, 9, 7);
d = d.plusDays(1).plusMonths(3).minusYears(2);
System.out.println(d); // 2011-12-08
```

Podemos adicionar os mesmos três meses usando uma `ENUM` de unidade de tempo. Cuidado que não estamos falando para alterar o campo `ChronoField.WEEK`. Isso seria errado, pois não queremos alterar um campo de semana, ou mesmo de dia. Queremos somar uma unidade de tempo, isto é, a API tem que se virar sozinha para adicionar os dias, meses etc., de acordo com o que nós pedirmos, mesmo se for ano bissexto etc. Portanto, não falamos o campo que desejamos alterar mas, sim, a unidade, `ChronoUnit`:

```
LocalDate d = LocalDate.of(2013, 9, 7);
d = d.plusWeeks(3).minus(3, ChronoUnit.WEEKS);
System.out.println(d); // 2013-09-07
```

Para fixar se é `ChronoField` ou `ChronoUnit`, lembre-se de que você deseja saber (`get`) o valor de um campo, então `ChronoField`, `.DAY` para o dia específico. Caso você deseje adicionar dias, para adicionar N dias, usa-se `ChronoUnit.DAYS` (plural).

Caso você tente manipular uma data usando uma unidade não suportada, será lançada a exception `UnsupportedTemporalTypeException`:

```
LocalDate d = LocalDate.of(2013, 9, 7);
                        // UnsupportedTemporalTypeException
                        //LocalDate does not support hours!
d = d.plus(3, ChronoUnit.HOURS);
System.out.println(d);
```

> **A classe MonthDay**
>
> Atenção, a classe `MonthDay` não possui nenhum método para somar ou subtrair unidades de tempo, logo, ela não tem nenhum método `plus` ou `minus`, e também não possui um método `isSupported` que receba `ChronoUnit`.

Convertendo entre os diversos tipos de datas

A classe `LocalDateTime` possui métodos para converter esta data/hora em objetos que só possuem a data (`LocalDate`) ou que só possuem a hora (`LocalTime`):

```
LocalDateTime now = LocalDateTime.now();
LocalDate dateNow = now.toLocalDate(); // from datetime to date
LocalTime timeNow = now.toLocalTime(); // from datetime to time
```

As classes também possuem métodos para combinar suas partes e criar um novo objeto modificado:

```
LocalDateTime now = LocalDateTime.now();
LocalDate dateNow = now.toLocalDate(); // from datetime to date
LocalTime timeNow = now.toLocalTime(); // from datetime to time

// from date to datetime
LocalDateTime nowAtTime1 = dateNow.atTime(timeNow);
// from time to datetime
LocalDateTime nowAtTime2 = timeNow.atDate(dateNow);
```

Trabalhando com a API legada

Para tornar a API legada compatível com a API nova, foram introduzidos vários métodos nas antigas classes `java.util.Date` e `java.util.Calendar` .

O exemplo a seguir converte uma `java.util.Date` em `LocalDateTime` , usando a timezone padrão do sistema:

```
Date d = new Date();
Instant i = d.toInstant();
LocalDateTime ldt1 =
    LocalDateTime.ofInstant(i, ZoneId.systemDefault());
```

O próximo exemplo transforma um `Calendar` pelo mesmo processo:

```
Calendar c = Calendar.getInstance();
Instant i = c.toInstant();
LocalDateTime ldt2 = LocalDateTime.ofInstant(i,
                                        ZoneId.systemDefault());
```

Repare que, para fazer a conversão, usamos como intermediário a classe `Instant` , que representa o número de milissegundos desde `01/01/1970` . Também podemos usar essa classe para fazer o caminho de volta:

```
Date d = new Date();
Instant i = d.toInstant();
LocalDateTime ldt1 =
    LocalDateTime.ofInstant(i, ZoneId.systemDefault());

Instant instant = ldt1.toInstant(ZoneOffset.UTC);
Date date = Date.from(instant);
```

Cálculos de intervalo de tempo com datas

Quando necessitamos realizar algum tipo de cálculo

envolvendo duas datas, podemos usar as classes `Duration`, `Period` e o método `between` da classe `ChronoUnit`:

`Duration` é a classe de mais baixo nível, usada para manipular objetos do tipo `Instant`. O exemplo a seguir soma 10 segundos ao instante atual:

```
Instant now = Instant.now(); // now
Duration tenSeconds = Duration.ofSeconds(10); // 10 seconds
Instant t = now.plus(tenSeconds); // now after 10 seconds
```

O próximo exemplo mostra como pegar o intervalo em segundos entre dois instantes:

```
Instant t1 = Instant.EPOCH; // 01/01/1970 00:00:00
Instant t2 = Instant.now();
long secondsSinceEpoch = Duration.between(t1, t2).getSeconds();
```

Note que `Duration` só tem a opção de `getSeconds`. Não existem métodos do tipo `getDays` etc.

`ChronoUnit` é uma das classes mais versáteis, pois permite ver a diferença entre duas datas em várias unidades de tempo:

```
LocalDate birthday = LocalDate.of(1983, 7, 22);
LocalDate base = LocalDate.of(2014, 12, 25);

// 31 years total
System.out.println(ChronoUnit.YEARS.between(birthday, base));
// 377 months total
System.out.println(ChronoUnit.MONTHS.between(birthday, base));
// 11479 days total
System.out.println(ChronoUnit.DAYS.between(birthday, base));
```

Já a classe `Period` pode ser usada para fazer cálculos de intervalos, quebrando as unidades de tempo do maior para o menor. Vamos tentar calcular a idade de uma pessoa:

```
LocalDate birthday = LocalDate.of(1983, 7, 22);
```

```
LocalDate base = LocalDate.of(2014, 12, 25);

Period lifeTime = Period.between(birthday, base);

System.out.println(lifeTime.getYears()); // 31 years
System.out.println(lifeTime.getMonths()); // 5 months
System.out.println(lifeTime.getDays()); // 3 days
```

Formatando e convertendo em texto

Para formatar a impressão de nossas datas, usamos a classe `DateTimeFormatter`, do pacote `java.time.format`. Ele segue o mesmo padrão da clássica `SimpleDateFormat`.

```
LocalDate birthday = LocalDate.of(1983, 7, 22);
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("yyyy MM dd");
System.out.println(formatter.format(birthday)); // 1983 07 22
```

Também podemos passar o formatador como parâmetro para o método `format` dos objetos de data:

```
LocalDate birthday = LocalDate.of(1983, 7, 22);
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("yyyy MM dd");
System.out.println(birthday.format(formatter)); // 1983 07 22
```

Já para transformar um texto em uma data, usamos o `DateTimeFormater` juntamente com o método `parse` da classe que desejamos instanciar:

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate d = LocalDate.parse("23/04/1986",formatter);
System.out.println(formatter.format(d)); // 23/04/1986
```

Caso usemos algum caractere não suportado ou passemos uma data no formato inválido, será lançada uma `DateTimeParseException`:

```
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("dd/MM/yyyy");
LocalDate d =
    LocalDate.parse("23/15/1986",formatter);
    // throws DateTimeParseException
System.out.println(formatter.format(d)); // 23/04/1986
```

Exercícios

1) Todas as classes da nova API de datas são:

a) Imutáveis

b) Mutáveis

2) A classe que representa um horário sem data é:

a) `LocalDateTime`

b) `TimeZone`

c) `LocalTime`

d) `Time`

3) Qual código cria um objeto com a data e hora atual?

a) `LocalDate.now();`

b) `LocalDateTime.now();`

c) `LocalDateTime.current();`

d) `new LocalDateTime();`

4) Qual o código que calcula a diferença em dias entre dois `LocalDate d1 e d2`?

a) `ChronoUnit.Days.between(d1,d2);`

b) `ChronoUnit.DAYS.difference(d1,d2);`

c) `ChronoUnit.DAYS.between(d1,d2);`

d) `Period.between(d1,d2);`

5) Qual o código que calcula a diferença em horas entre dois `LocalDateTime d1 e d2`?

a) `Duration.between(d1, d2)`

b) `Period.between(d1,d2).getHours();`

c) `Duration.between(d1, d2).toHours();`

d) `Period.between(d1,d2);`

6) Veja o seguinte código:

```
LocalDate birthday = LocalDate.of(1975, 9, 23);
LocalDate today = LocalTime.now();

//code here

System.out.println("You are " + d.getYears() + " years, " +
                   d.getMonths() + " months, and " +
                   d.getDays() + " days old.");
```

Qual trecho que, se inserido no local indicado, vai imprimir corretamente a idade?

a) `Period d = Period.between(birthday, today);`

b) `Duration d = Duration.between(birthday, today);`

c) `Period d = Period.between(today,birthday);`

d) `Period d = Period.difference(birthday, today);`

7) Qual o trecho de código que converte um objeto `Date` em um objeto `LocalDate` ?

a)

```
LocalDate.from(d.toInstant());
```

b)

```
LocalDateTime.from(d.toInstant()).toLocalDate();
```

c)

```
LocalDateTime.ofInstant(d.toInstant()).toLocalDate();
```

d)

```
LocalDateTime.ofInstant(d.toInstant(),
                    ZoneId.systemDefault()).toLocalDate();
```

8) Sobre o código a seguir, ao tentarmos compilar e executá-lo, o que acontece?

```
System.out.println(YearMonth.now().isSupported(
    ChronoField.DAY_OF_MONTH));
System.out.println(MonthDay.now().isSupported(
    ChronoUnit.DAYS));
System.out.println(LocalDate.now().isSupported(
    ChronoUnit.DAYS));
System.out.println(LocalDateTime.now().isSupported(
    ChronoField.DAY_OF_MONTH));
```

a) Compila e imprime `false` , `true` , `true` , `true` .

b) Compila e imprime `true` , `true` , `true` , `true` .

c) Compila e imprime `true` , `true` , `false` , `true` .

d) Compila, mas lança exception ao executar.

e) Não compila.

9) Sobre o código a seguir, ao tentarmos compilar e executá-lo, o que acontece?

```
System.out.println(YearMonth.now().isSupported(
    ChronoUnit.MONTHS));
System.out.println(YearMonth.now().isSupported(
    ChronoField.DAY_OF_MONTH));
System.out.println(MonthDay.now().isSupported(
    ChronoField.DAY_OF_MONTH));
System.out.println(LocalDate.now().isSupported(
    ChronoUnit.DAYS));
System.out.println(LocalDate.now().isSupported(
    ChronoUnit.YEAR));
System.out.println(LocalDateTime.now().isSupported(
    ChronoField.HOUR_OF_AMPM));
System.out.println(LocalDateTime.now().isSupported(
    ChronoField.YEAR));
```

a) Compila e imprime `true`, `false`, `true`, `true`, `true`, `true`, `true`, `true`.

b) Compila e imprime `true`, `true`, `true`, `true`, `true`, `true`, `false`, `true`.

c) Compila, mas lança exception ao executar.

d) Não compila.

11.2 EXPRESSÃO LAMBDA SIMPLES QUE CONSUME UMA LAMBDA PREDICATE

Entre as diversas novidades do Java 8, uma que se destaca bastante é a inclusão de *lambdas*, trazendo algumas características funcionais à linguagem. Para ver melhor a utilidade dos *lambdas* e entender o que será cobrado na prova, vamos focar no exemplo a

seguir.

Imagine que você precise escrever um método que recebe como parâmetros uma lista e um critério de busca, e retornar outra lista com os elementos que atendem ao critério. Poderíamos implementar este código da seguinte maneira:

```
class Person {
    private String name;
    private int age;
    //...
}

interface Matcher<T>{
    boolean test(T t);
}

class PersonFilter{

    public List<Person> filter(List<Person> input,
                               Matcher<Person> matcher){
        List<Person> output = new ArrayList<>();
        for (Person person : input) {
            if(matcher.test(person)){
                output.add(person);
            }
        }
        return output;
    }
}
```

Ótimo, nossa base está montada. Agora, para filtrar de uma lista de pessoas com apenas as maiores de idade, podemos implementar um `Matcher` da seguinte maneira:

```
class AgeOfMajority implements Matcher<Person>{
    @Override
    public boolean test(Person p) {
        return p.getAge() >= 18;
    }
}
```

E usar esta classe em nosso código:

```
PersonFilter pf = new PersonFilter();
List<Person> adults = pf.filter(persons, new AgeOfMajority());
```

O problema dessa abordagem é que, sempre que quisermos um critério diferente, precisamos criar uma nova classe que implemente `Matcher`, mesmo se for para usar apenas uma vez. Podemos reduzir um pouco esse impacto usando classes anônimas, mas a legibilidade do código fica prejudicada:

```
List<Person> adults = pf.filter(persons, new Matcher<Person>() {
    @Override
    public boolean test(Person p) {
        return p.getAge() >= 18;
    }
});
```

É para resolver este tipo de problema que existem os *lambdas*. Um *lambda* é um trecho de código que pode ser passado como parâmetro para um método, ou ser armazenado em uma variável para ser invocado posteriormente.

Para usar um *lambda* em Java, precisamos de uma **interface funcional**. Interfaces funcionais são interfaces normais, mas com apenas um método. Nossa interface `Matcher` pode ser considerada funcional. É possível checar se uma interface é funcional usando a *annotation* `FunctionalInterface`; se não for funcional, o código não compila.

```
@FunctionalInterface
interface Matcher<T>{
    boolean test(T t);
}
```

O Java já vem com várias dessas interfaces funcionais para os cenários comuns. Uma destas interfaces é a `Predicate`, que

recebe um objeto como parâmetro e retorna um `boolean`, exatamente igual a nosso `Matcher`. Vamos trocar o código da classe `PersonFilter` para usar o `Predicate`:

```
import java.util.function.Predicate;

class PersonFilter{

    public List<Person> filter(List<Person> input,
                               Predicate<Person> matcher){
        List<Person> output = new ArrayList<>();
        for (Person person : input) {
            if(matcher.test(person)){
                output.add(person);
            }
        }
        return output;
    }
}
```

E para fazer o filtro:

```
Predicate<Person> matcher = new Predicate<Person>() {
    @Override
    public boolean test(Person p) {
        return p.getAge() >= 18;
    }
};
List<Person> adults = pf.filter(persons, matcher);
```

Ok, vamos converter este código para um *lambda*. A sintaxe básica do lambda é a seguinte:

```
( parameters ) -> { code }
```

Usando esta fórmula no código anterior, temos o seguinte:

```
Predicate<Person> matcher =
    (Person p) -> {return p.getAge() >= 18;};
```

Perceba quanto código foi removido. Praticamente toda a

declaração do tipo, de que não precisamos, já que a declaração do tipo da variável tem o mesmo de forma explícita. Também removemos o nome do método, que também não é necessário, já que interfaces funcionais possuem apenas um método.

Podemos remover mais código ainda. Repare que a variável `matcher` é do tipo `Predicate<Person>`. Aqui podemos inferir o tipo do parâmetro pelo tipo *generics* da interface. O código fica:

```
Predicate<Person> matcher = (p) -> {return p.getAge() >= 18;};
```

Se temos apenas um argumento, podemos ainda remover os parênteses:

```
Predicate<Person> matcher = p -> {return p.getAge() >= 18;};
```

Se temos apenas uma linha de código dentro do *lambda*, podemos omitir as chaves. Se esta linha for o retorno, podemos omitir a palavra `return` também:

```
Predicate<Person> matcher = p -> p.getAge() >= 18;
```

Pronto, já retiramos bastante código, está bem mais limpo. Nosso código no final fica assim:

```
Predicate<Person> matcher = p -> p.getAge() >= 18;
List<Person> adults = pf.filter(persons, matcher);
```

Não somos obrigados a armazenar o *lambda* em uma variável, podemos passá-lo diretamente como parâmetro do método:

```
List<Person> adults = pf.filter(persons, p -> p.getAge() >= 18);
```

Vamos entender qual a vantagem desta abordagem. Se agora precisarmos de um outro filtro, que retorna apenas as pessoas cujo nome comece com a letra "A", podemos simplesmente fazer:

```
List<Person> namesStartingWithA =
    pf.filter(persons, p -> p.getName().startsWith("A"));
```

Não há necessidade de criar classes, nem mesmo anônimas. A inclusão dos *lambdas* nos permite escrever código altamente adaptável e ainda reduzir muito a verbosidade comum do Java.

Antes de passar para o próximo exemplo, vamos entendera regras para se escrever um *lambda*.

- *Lambdas* podem ter vários argumentos, como um método. Basta separá-los por `,`.
- O tipo dos parâmetros pode ser inferido e, assim, omitido da declaração.
- Se não houver nenhum parâmetro, é necessário incluir parênteses vazios, como em:

```
Runnable r = () -> System.out.println("a runnable objec
t!");
```

- Se houver apenas um parâmetro, podemos omitir os parênteses, como em:

```
Predicate<Person> matcher = p -> p.getAge() >= 18;
```

- O corpo do *lambda* pode conter várias instruções, assim como um método.
- Se houver apenas uma instrução, podemos omitir as chaves, como em:

```
Predicate<Person> matcher = p -> p.getAge() >= 18;
```

- Se houver mais de uma instrução, é necessário delimitar o corpo do *lambda* com chaves, como em:

```
        Runnable r = () ->  {
            int a = 10;
            int b = 20;
            System.out.println(a + b);
        }
```

Acessando variáveis do objeto com lambdas

Lambdas podem interagir com as variáveis de instância dos objetos onde foram declarados. Temos apenas de tomar cuidado com variáveis marcadas como `final`:

```
public class LambdaScopeTest {

    public int instanceVar = 1;
    public final int instanceVarFinal = 2;

    public static void main(String[] args) {
        new LambdaScopeTest().test();
    }

    private void test() {
        instanceVar++; // ok
        new Thread(() -> {
            System.out.println(instanceVar); // ok
            instanceVar++; // ok

            System.out.println(instanceVarFinal); // ok
            instanceVarFinal++; // compile error
        }).start();
    }
```

Já com variáveis locais de método e parâmetros, as regras são um pouco mais complexas. *Lambdas* só podem interagir com variáveis locais caso estas estejam marcadas como `final` (uma referência imutável), ou que sejam **efetivamente final** (não são `final`, mas não são alteradas). Não é possível alterar o valor de nenhuma variável local dentro de um *lambda*:

```
    private void test() {
```

```
        int unchangedLocalVar = 3;   // effectively final
        final int localVarFinal = 4; // final
        int simpleLocalVar = 0;
        simpleLocalVar = 9; // updated the value

        new Thread(() -> {
            System.out.println(unchangedLocalVar);   // can read
            System.out.println(localVarFinal);       // can read
            System.out.println(simpleLocalVar); // compile error
        }).start();
    }
```

Conflitos de nomes com lambdas

As variáveis do *lambda* são do mesmo escopo que o método onde ele foi declarado, portanto, não podemos declarar nenhuma variável, como parâmetro ou dentro do corpo, cujo nome conflite com alguma variável local do método:

```
private void test(String param) {
    String methodVar = "method"; //not final

    Predicate<String> a =
        param -> param.length() > 0; //compile error
    Predicate<String> b =
        methodVar -> methodVar.length() > 0; //compile error
    Predicate<String> c =
        newVar -> newVar.length() > 0; // ok
}
```

Exercícios

1)

```
interface Printer{
    void printMessage();
}

class A {
    public static void main(String[] args) {
        Printer p = null;
```

```
        //code here
        p.printMessage();
    }
}
```

Qual código, se incluído na linha indicada, imprime a mensagem `"Hello World"` ?

- `p = () -> System.out.println("Hello World");`
- `p = () => System.out.println("Hello World");`
- `p = -> System.out.println("Hello World");`
- `p -> System.out.println("Hello World");`

2) Qual das expressões a seguir é inválida?

a)

```
Predicate<List> bigger = list -> list.size() > 1000;
```

b)

```
Predicate<String> startsWithA = s -> s.startsWith("A");
```

c)

```
Predicate big = list -> list.size() > 100;
```

d)

```
String name="";
Predicate isEmpty = s -> s.equals("");
```

3) Considere a seguinte interface:

```
interface Calculator<T>{
    T function(T a, T b);
}
```

Qual dos códigos a seguir não é válido?

a)

```
Calculator<Integer> sum = (a,b) -> a + b;
```

b)

```
Calculator<Integer> multiply =
    (Integer a,Integer b) -> (int) a * b;
```

c)

```
Calculator<Integer> subtract = (a,b) -> {return a - b;};
```

d)

```
Calculator<Integer> divide =
    (int a, int b) -> {return (Integer) a / b;};
```

4) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++){
            new Thread(() -> System.out.println(i)).start();
        }
    }
}
```

a) Imprime 0 dez vezes.

b) Imprime os números de 0 a 9.

c) Compila e executa, mas não imprime nada.

d) Erro de compilação.

5) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A{
```

```
    int a = 0;

    public static void main(String[] args) {
        new A().method(1, a -> a > 0); // A
    }

    private void method(int a, Predicate<Integer> d) {
        if(d.test(a)){ // B
            System.out.println(a);
        }
    }
}
```

a) Compila e imprime `0`.

b) Compila e imprime `1`.

c) Compila e não imprime nada.

d) Não compila por erro na linha **A**.

e) Não compila por erro na linha **B**.

6) Escolha a opção adequada ao tentar compilar e rodar o arquivo a seguir:

```
class A{
    int a = 0;

    public static void main(String[] args) {
        new A().method(1);
    }

    private void method(int a) {
        Predicate<Integer> d = a -> a > 0; // A

        if(d.test(a)){ // B
            System.out.println(a);
        }
    }
}
```

a) Compila e não imprime nada.

b) Compila e imprime `1`.

c) Compila e imprime `0`.

d) Não compila por erro na linha **A**.

e) Não compila por erro na linha **B**.

Capítulo 12

BOA PROVA

Agora que você acabou todo o livro, está na hora de revisar os pontos em que ficou com dúvida. Refaça os exercícios para reforçar o conteúdo e faça diversos simulados como os citados no começo do livro.

Para efetuar a prova, primeiro pague no site da Pearson Vue (pearsonvue.com/oracle/) ou da Oracle. Busque no site dela pelo código da prova que deseja fazer, a Java SE 7 Programmer I (1Z0-803) ou Java SE 8 Programmer I (1Z0-808). Adicione-a ao carrinho e efetue a compra.

No site da Oracle, o pagamento é feito via boleto enviado por e-mail, tome cuidado pois ele pode cair em sua caixa de spam, e tome ainda mais cuidado para não pagar um boleto de spam. No site da Pearson, você pode pagar com cartão de crédito e agendar sua prova no mesmo instante. Depois de confirmado o recebimento por parte da Oracle, você receberá um código de confirmação com o qual será capaz de agendar a data e local de sua prova.

Lembre-se de reler os pontos em que tem mais dificuldade no dia que antecede a prova.

Mais uma vez, boa prova. Desejo que todo esse processo tenha

aberto sua mente sobre como a linguagem Java funciona, seus limites e suas características gerais.

Depois dessa certificação, que venha a próxima, que exige ainda mais conhecimento da linguagem e de suas APIs fundamentais.

CAPÍTULO 13

RESPOSTAS DOS EXERCÍCIOS

Exercícios da seção 1.1

1. A resposta certa é (a). A variável `i` não pode ser acessada fora do laço.
2. A resposta certa é (a). A variável `i` não pode ser acessada fora do laço. Repare que o laço não foi declarado com `{`, uma pegadinha clássica da prova.
3. A resposta certa é (c). A variável `i` declarada no `for` só é visível dentro do `for`.
4. A resposta certa é (c). A variável `x` declarada como parâmetro do método `main` efetua um shadowing. Nesse instante, ao dizermos `x = 200`, tentamos atribuir um `int` a um array de `String`, erro de compilação.
5. A resposta certa é (d). Isso porque o acesso à variável estática pode ser feito através da instância da classe, ou diretamente caso seja uma variável estática sendo acessada por um método estático.

Exercícios da seção 1.2

1. A resposta certa é (a). O arquivo não compila, pois não

podemos ter um `import` após a definição de uma classe.

2. A resposta certa é (e). O código compila e roda, não imprimindo nada, pois não chamamos o método `Test`, somente o construtor `Test()`.
3. A resposta certa é (c). O código compila sem erros: a ordem `package` e `import` está adequada, e os tipos são opcionais dentro de um arquivo Java.

Exercícios da seção 1.3

1. O método `main` não pode devolver `int` nem `Void`. Ele também deve ser público e só receber um argumento: um array (ou varargs) de `String`, portanto, `public static void main(String... args)`. A resposta certa é (a).
2. Para compilar, estamos trabalhando com arquivos e diretórios, portanto `javac b/A.java;`. Já para rodar, estamos pensando em pacotes e classes: `java b.A`. A resposta certa é (d).
3. Ao rodar sem argumentos, nosso array tem tamanho zero, assim, ao tentar acessar seu primeiro elemento, recebemos um `ArrayIndexOutOfBoundsException` na linha C. A resposta certa é (c).
4. Para rodar um programa dentro de um `jar` sem ter um manifesto, devemos usar o classpath customizado. Colocamos o `jar` no classpath e dizemos qual classe desejamos rodar: `java -cp program.jar b.A`. A resposta certa é (f).
5. Durante a compilação, para adicionar o arquivo `program.jar` ao classpath, devemos usar `-cp program.jar` e, para especificar o arquivo adequado, usamos `b/A.java`. A opção que apresenta essas duas

características é `javac -cp program.jar:. b/A.java`. A resposta certa é (h).

Exercícios da seção 1.4

1. A resposta certa é (a). Ocorre um erro de compilação na classe `Test` ao tentar importar uma classe não acessível a outros pacotes.
2. A resposta certa é (a). O erro de ambiguidade é dado no `import` e não na utilização, portanto o arquivo não compila e o erro é na linha do `import`.
3. A resposta certa é (c). Não existe ambiguidade, uma vez que o `import` específico tem preferência em cima do `*`.
4. A resposta certa é (c). Não há erro de ambiguidade, simplesmente um `import` é desnecessário e não gera erro nenhum, apenas um *warning*.
5. A resposta certa é (e). O arquivo B compila, pois é uma classe normal. O arquivo `C` não compila, pois tenta acessar `B`, que está em outro pacote, mas lembre-se de que devemos acessar os pacotes diretamente, não existe subpacote. O mesmo vale para `A`. Portanto, nem `A` nem `C` compilam.
6. A resposta certa é (d). Um pacote pode ter nome começando com maiúsculo, isso não afeta em nada. Mas não é o padrão. O nome de variáveis locais e parâmetros não afetam a assinatura de um método em Java. Uma classe não precisa ser pública para ser rodada. Portanto, o código compila e roda.
7. A resposta certa é (b). Não importamos a classe A, somente seus membros, erro de compilação ao tentar referenciá-la na linha 5.
8. A resposta certa é (a). `import static` é o uso adequado, e

não `static import`, erro na linha 3.

9. A resposta certa é (a). `B` não compila, pois tenta acessar uma classe do pacote padrão (sem nome). Classes do pacote padrão só podem ser acessadas por outros tipos do pacote padrão. Não compila.

Exercícios da seção 2.1

1. A resposta certa é (d). O código compila e imprime 100. Podemos ter espaços em branco desde que não quebre uma palavra-chave, nome de método, classe etc. ao meio. Onde se pode ter um espaço em branco, podem haver vários.
2. A resposta certa é (a). O código não compila pois tentamos acessar a variável `age` que pode não ter sido inicializada. Não é certeza (somente se cair no `if` ela será inicializada).
3. A resposta certa é (d). Se o fluxo chegar na chamada do `println`, isso significa que ele passou pela inicialização da variável `age`.
4. A resposta certa é (f). Não compila, do lado direito da atribuição temos um array de boolean e do lado esquerdo uma variável simples do tipo boolean.
5. A resposta certa é (b). Imprime `false`, pois um array de tipos primitivos após a inicialização tem seus valores com o valor padrão do tipo. Para numéricos, é 0; para `boolean`, é `false`; e para referências, é `null`.
6. A resposta certa é (b). Não compila, pois `boolean` em Java só pode ser `false` ou `true`.
7. A resposta certa é (a). O número octal 09 não existe. Você não precisa aprender a transformar uma base em outra, mas é importante lembrar de que binários são compostos de 0s e

1s, octais são compostos de 0s até 7s, hexadecimais são de 0s até 9s e As até Fs (maiúsculo ou minúsculo).

O caractere `_` é permitido desde que dos dois lados dele tenhamos algarismos válidos, que é o caso de `1_000`. Portanto, o único número inválido é 09 (por curiosidade, o número 9 em base octal é 011).

8. A resposta certa é (c). Compila e imprime o alfabeto, pois caracteres são números em Java.

9. A resposta certa é (e). `instanceOf` não é palavra reservada: note a letra maiúscula no meio dela. **Nenhuma** palavra-chave em Java possui caractere maiúsculo.
10. A resposta certa é (d). Compila e roda, não imprimindo nada. Lembre-se de que os identificadores são *case-sensitive*.

Exercícios da seção 2.2

1. A resposta certa é (e). Imprime 47, pois a atribuição é por cópia do valor.
2. A resposta certa é (f). Imprime 48, pois a atribuição de objetos é feita por cópia da referência, criamos somente um único objeto do tipo `B`.

Exercícios da seção 2.3

1. A resposta certa é (d). Não existe conflito de nomes entre variável `membro` e método, ou variável `membro` e variável `local`. Ao invocar o método `c`, por causa do `shadowing` da variável `c`, não acessamos a variável membro, sem alterá-la. O resultado é a impressão dos valores 10 e 10

novamente.

Exercícios da seção 2.4

1. A resposta certa é (c). Compila e não podemos falar nada.
2. A resposta certa é (b). Somente 10 objetos podem ser garbage coletados, pois o último continua referenciado pela variável `b`.
3. A resposta certa é (b). O código compila, mas como não chamamos nenhum construtor, o único objeto criado que se assemelha a `B`, porém não é `B`, é um array do tipo `B`, com 100 espaços. Nenhum objeto é criado. Note que, para criar, devemos invocar o construtor por padrão.

Exercícios da seção 2.5

1. A resposta certa é (b). O código compila e, devido à regra de sempre invocar o mais específico, ele sempre invoca o método sem argumentos. Portanto, o resultado é vazio/vazio. Lembre-se de que, em Java, a ordem de definição de métodos não importa para a invocação. Já a ordem das variáveis pode importar, caso uma dependa da outra.
2. A resposta certa é (c). O código compila e imprime 2.
3. A resposta certa é (e). O código compila e imprime 2.
4. A resposta certa é (e). O código compila e imprime 2. Esse é o caso absurdo no qual o array é tanto um `Object` quanto um array de `Object`. Por padrão, o Java tratará como um array de `Object`.

Exercícios da seção 2.6

1. A resposta certa é (e). O array começa na posição 0, portanto, o primeiro caractere removido se encontra na posição 2, o `i`. Ele remove todos os caracteres até a posição 3, exceto o da posição 3, logo, somente o `i` é removido.
2. A resposta certa é (a). Os dois métodos retornam `-1` quando não encontram nada, portanto, o segundo resultado é `-2`. Como a posição começa em 0, o resultado das letras são `5` e `8`, totalizando `13`.

Exercícios da seção 2.7

1. A resposta certa é (a). Não compila, pois `length()` é um método de `String`, diferente dos arrays em que `length` é um atributo.
2. A resposta certa é (b). Dá `NullPointerException! msg` e `null`, e não dá para chamar `isEmpty` em `null`.
3. A resposta certa é (a). Não compila, pois a variável não foi inicializada.
4. A resposta certa é (a). 'Caelum' e 'Caelum - Ensino e Inovação'.
5. A resposta certa é (e). Compila e imprime `Welcome null`.
6. A resposta certa é (b). Não compila, pois a String não foi inicializada.
7. A resposta certa é (b). Não compila por outro motivo: a variável `empty` não é estática.
8. A resposta certa é (e). Compila e imprime `Welcome null`.
9. A resposta certa é (d). Dá `NullPointerException` ao tentar criar a segunda `String`.
10. A resposta certa é (c). O código compila e imprime `uda`.
11. A resposta certa é (a). Não compila, pois `String` possui diversos construtores que recebem um argumento: o

compilador não sabe qual deles você deseja invocar, já que os tipos que são argumentos do construtor não possuem herança entre si (um não herda necessariamente do outro).

12. A resposta certa é (c). Nenhuma das alternativas dadas com número, pois primeiro ele soma `val` e `div`, imprimindo 14. A conta de divisão é feita entre `int`, devolvendo um `int` de valor 2. Quando esse número é atribuído a um `double`, continua sendo 2. Logo, imprime `14 e 2.0`.
13. A resposta certa é (a). O código não compila, pois o método `replace` possui duas maneiras de ser invocado: com dois `chars`, ou com duas `Strings`. Foram passados uma `String` e um `char`, método que não existe.
14. A resposta certa é (a). Pensamos que pode ser `gualherme`, mas lembramos de que `String` em Java é imutável e ela não foi reatribuída. O `=` dá uma impressão de reatribuição de parte da `String`, mas isso não existe em Java. O lado esquerdo de uma atribuição deve ser sempre uma variável, e não uma chamada a um método. Por isso, a linha do `substring` não compila.

Exercícios da seção 3.1

1. As opções (a) e (c) não compilam e precisam do casting mesmo com `short` e `char` tendo 2 bytes.

   Na opção (a), pode ocorrer de o `short` ser negativo e, portanto, não caber no intervalo dos `chars`. Na opção (c), o `char` pode ser muito grande e sair fora do alcance dos positivos do `short`. A opção (b) compila, pois o `char` que possui 2 bytes pode ser atribuído para um `long` que possui 8 bytes.

2. A resposta certa é (b). Análise linha a linha: – divisão inteira: `i1` vale `1`. – divisão inteira, que depois é promovido a `double`: `i2` vale `1.0`. – divisão `double`: `i3` vale `1.5`. – `x` vale `0L` e `d` vale `0.0` (duas promoções). – O resultado é `3.5`.

3. A resposta certa é (d). Mesmo `c` sendo `null`, por estarmos usando o operador `&`, a segunda parte da expressão (`c.getPreco() > 10000`) será avaliada, causando uma `NullPointerException` na chamada do método `getPreco()`, caso `c` seja `null`. Poderíamos evitar isso usando o operador de curto-circuito, `&&`.

4. A resposta certa é (d): `i e c`.

5. A resposta certa é (d). Pode-se utilizar o operador booleano de ou exclusivo (`^`):

```
if (train ^ car) {
    // ....
}
```

6. 1ª linha: `ArithmeticException: / by zero`.

   2ª linha: `Infinity`.

   3ª linha: `Infinity`.

   4ª linha: `-Infinity`.

7. A resposta certa é (b). O código não compila na linha 7. O compilador não tem certeza se a variável `y` vai ser iniciada sempre. Como a declaração é feita e o único valor atribuído é dentro do `for`, o compilador não tem certeza se o `for` vai ser executado mesmo.

8. A resposta certa é (d).
9. A resposta certa é (e). Compila, roda e imprime `A75`. Cuidado ao compilar e rodar, pois alguns caracteres podem precisar ser escapados pelo seu `shell` ou `bash` (não cobrado na prova).
10. Não compila! Toda conta devolve no mínimo um `int`. O resultado de `b1 + b2` é `int`. Podemos fazer casting ou declarar `b3` como `int`.
11. A resposta certa é (c). Não compila, pois toda conta devolve no mínimo um `int`, e um `int` não cabe em um `byte`.
12. A resposta certa é (c), 3. O `for` externo vai contar de 0 a 5, mas dentro do `for` tem um `if` que pré-incrementa o `i`. Esse `if` vai quebrar o loop no momento que o valor retornado `for` divisível por 3, isto é, quando `i` valer 3 nesse caso.
13. A resposta certa é (a), 1. Dessa vez, o valor de `i` será usado no `if` e só depois incrementado. Como o resto de 0 dividido por qualquer número também é 0, o `for` só executa uma vez. Mas o valor de `i` ainda será incrementado, imprimindo o valor de 1.
14. A resposta certa é (f). A segunda linha do método `main` não compila, pois estoura o limite de byte.
15. A resposta certa é (c). O código não compila, pois não podemos declarar um `char` negativo.
16. A resposta certa é (f). O código compila e imprime um outro valor (`68`).
17. A resposta certa é (b). O código compila e joga uma exception por causa da divisão inteira (são `ints`) por zero.
18. A resposta certa é (c). O código compila e imprime positivo infinito. A precedência de operadores é primeiro a divisão,

por isso compila.

19. A resposta certa é (a). Não compila, não há comparação entre `boolean` e números.

Exercícios da seção 3.2

1. A resposta certa é (a). Não compila, pois o resultado do parênteses é uma `String` que não possui o operador de divisão.
2. A resposta certa é (b). `true==false` é `false`. O inverso disso é `true`. Comparando com `true`, é `true`. Portanto, o operador ternário devolve 1 que é diferente de 0, imprimindo `false`.

Exercícios da seção 3.3

1. A resposta certa é (b). O código compila e imprime `true` e `false`.
2. A resposta certa é (d). Compila e imprime `false`, `false`, uma vez que a String 2 vale `s`.
3. A resposta certa é (c). Compila e imprime `true`, `true`. Por mais que `substring` devolva uma nova String, nesse caso ele devolveu a String inteira, a própria String.
4. A resposta certa é (e). O código imprime `false` e `true`. As duas referências são diferentes (`false`), e o método `equals` não foi sobrescrito, mas está sendo chamado diretamente (`true`).
5. A resposta certa é (a). O código não compila, pois `D` não é do tipo `C`.

Exercícios da seção 3.4

1. A resposta certa é (c). O código compila normalmente e imprime `0` caso não seja passado nenhum argumento.
2. A resposta certa é (a). O código não compila, pois a variável `valor` é `final` e não pode ser alterada, mas tentamos efetuar uma atribuição dentro do `if`.
3. A resposta certa é (a). O código não compila, já que tenta atribuir 15 a uma variável e conferir o valor 15 como se fosse um `boolean`.
4. A resposta certa é (a). O código não compila, pois não existe palavra chave `elseif`. Devemos fazer um `else if` para compilar.
5. A resposta certa é (b). O código não compila, porque o `else` não está aplicado ao `if`: para ser aplicado ao `if`, ele deve vir imediatamente após seu bloco. Como o `if` não possui chaves, somente a primeira linha pertence a ele.

Exercícios da seção 3.5

1. A resposta certa é (d). Ao rodar com 5 argumentos, o código imprime `+++`.
2. A resposta certa é (a). O código não compila, pois `t2` não é uma constante. Somente podemos verificar `case` de `switch` em variáveis finais inicializadas diretamente.
3. A resposta certa é (b). A `String "42"` é uma `String`, uma vez que ela está entre aspas. Portanto, o código imprime `Guilherme`.
4. A resposta certa é (a). A sintaxe do `case` é com `:` e não com `{`, então o código não compila.
5. A resposta certa é (a). O código não compila, já que o `case` não aceita expressões como `< x`, mas sim um valor definido em tempo de compilação.

6. A resposta certa é (a). O código não compila, pois há código que não será executado após `break`.

Exercícios da seção 4.1

1. A resposta certa é (e). Não faz sentido ter colchetes antes da declaração do tipo, portanto `[]int x` não compila.
2. A resposta certa é (b). A segunda linha não compila pois, ou você passa o tamanho, ou passa os valores.
3. A resposta certa é (b). O programa não compila, porque a segunda e a terceira linha tentam redefinir uma variável já definida. Caso o nome da variável seja corrigido, o código compila e imprime nada ao rodar (um array pode ter tamanho zero).
4. A resposta certa é (c). O programa inicializa `i` para o tamanho do array, acessando uma posição inexistente. Portanto dá erro em execução (exception).
5. A resposta certa é (b). Não tenha medo de simular o código na mão. Simule a memória e perceba que dá uma `Exception`.

6. A resposta certa é (f). Não tenha medo de simular o código na mão. Simule a memória e perceba que o resultado é `2, -5`. Para isso, desenhe os três espaços do array, aponte os valores iniciais 0 e continue atribuindo valores, executando o código.

   Durante a prova, simular os arrays e os ponteiros é ideal para não se perder em códigos complexos de referências e valores com arrays.

7. A resposta certa é (d). Compila e imprime `true`: note que

não existe criação de um novo array, nós simplesmente temos duas referências (`valores` e `vals` para o mesmo array na memória).

8. As respostas corretas são (a), (b), (f) e (j).

Exercícios da seção 4.2

1. A resposta certa é (a). Não compila pois, ao inicializarmos o array `zyx` , utilizamos um array de uma única dimensão.
2. A resposta certa é (f). Na posição `2` , temos o array `z` , que tem 30 casas, portanto, temos o resultado `30` .
3. A resposta certa é (g). O código compila e imprime `30` normalmente. Não há problema algum em apontar para um novo array.

4. A resposta certa é (a). Nesse exemplo, é guardado um valor `double` em uma das posições do array de `int id` . Isso está incorreto, logo, não compila.

   Na declaração do array de duas dimensões `tb` , são informados os tamanhos das dimensões. Errado pois os tamanhos devem ser definidos na inicialização, e não na declaração. Na inicialização do array `cb` , não foi colocado o tamanho de nenhuma das dimensões.

Exercícios da seção 4.3

1. A resposta certa é (a). O código não compila, pois a classe `ArrayList` não foi importada.
2. A resposta certa é (d). O código roda e imprime `true` , porque foi removido um elemento da lista.

3. A resposta certa é (f). O código roda e imprime `1`, já que ele remove o primeiro elemento igual ao elemento passado.
4. A resposta certa é (h). O código compila e imprime `5`.
5. A resposta certa é (a). O código não compila, pois o método `toArray` sem argumentos retorna um array de `Object`.
6. A resposta certa é (b). O código inclui os elementos sempre no final da `ArrayList`, portanto imprime `a` e depois `d`.
7. A resposta certa é (a). O código não compila, porque a ordem dos parâmetros para o método `add` é `int, String`.
8. A resposta certa é (b). O código compila e imprime somente `a`. Isso porque ele executa um `next` durante o passo de iteração do laço `for`, o que acaba consumindo o segundo elemento sem imprimi-lo.
9. A resposta certa é (c). O código compila e imprime `a, b, c, d`, pois o laço está alterando o valor referenciado pela variável `s`, e não o valor contido dentro da nossa `ArrayList`.

Exercícios da seção 5.1

1. A resposta certa é (e). O código compila e, ao rodar, a não é maior que `100`, portanto imprime `10`.
2. A resposta certa é (c). O código já compila, pois a variável não é final, e entra em loop infinito.

Exercícios da seção 5.2

1. A resposta certa é (a). O código não compila, pois o laço nunca é quebrado e nunca chega a executar o código que imprime `b`.

2. A resposta certa é (a). O código não compila, já que o código dentro de `for` nunca será executado.
3. A resposta certa é (a). O código não compila, pois tentamos definir o tipo de duas variáveis no nosso `for` , mesmo que os tipos sejam o mesmo.
4. A resposta certa é (b). Compila e imprime os valores 0 até 9.
5. A resposta certa é (a). O código não compila, porque só podemos ter uma condição para o laço `for` .
6. A resposta certa é (e). O código compila e imprime `0 1 1 2` .

Exercícios da seção 5.3

1. A resposta certa é (b). O código compila e imprime `false` , pois ele sempre entra no laço pelo menos uma vez.
2. A resposta certa é (b). Compila e entra em loop infinito caso seja passado zero, um ou dois argumentos. Não imprime nada caso 3 a 9 argumentos. Imprime `finished` caso 10 ou mais argumentos.
3. A resposta certa é (c). O código compila e imprime `0` , já que a condição é `i` maior que 10.
4. A resposta certa é (a). O código não compila, pois faltou um ponto e vírgula.
5. A resposta certa é (c). Compila e sai.

Exercícios da seção 5.4

1. A resposta certa é (a). Quando iteramos por duas coleções ao mesmo tempo, podemos usar tanto o `for` quanto o `while` , mas o `for` é mais simples, pois passa por todos os elementos já com a inicialização e incremento bem definidos

dentro do laço.

2. A resposta certa é (a). Usamos o `for` tradicional (com ou sem `Iterator`) para remover elementos. Poderíamos usar o `while`, mas ele não está na lista de respostas.
3. A resposta certa é (b). Devemos usar o `do...while`, que garante a execução pelo menos uma vez do código.
4. A resposta certa é (d). `for`, `while` ou `do...while` resolvem o problema, mas o mais comum é o `while`.
5. A resposta certa é (b). Não é possível inicializar os valores de um array com o `enhanced for`, portanto usamos o `for`.

Exercícios da seção 5.5

1. O código não compila, pois o segundo `if` está fora do bloco do `for` e tenta acessar uma variável definida dentro dele. Lembre-se de que o escopo de um bloco `for` sem chaves é uma única instrução, no caso o primeiro `if else if`.
2. A resposta certa é (d). Compila e imprime 0 até 19, 21 até 24, 26 até 29.
3. A resposta certa é (j). Compila e, ao rodar com 0 argumentos, imprime `1` até `15`, `end`.

Exercícios da seção 6.1

1. A resposta certa é (a). O código não compila, pois existe um `return` sem valor, e o método `x` deve retornar um `int`.
2. A resposta certa é (a). Não compila, pois a variável `l`, apesar de ser `final`, não é considerada uma constante pelo compilador. Para ser uma constante, a variável tem de ser `final` e ter seu valor atribuído durante a inicialização.
3. A resposta certa é (b). O código não compila, porque o

método `c` retorna um `long`, e esse `long` é utilizado como retorno no método `a` e no método `b`. Ambos precisam de um retorno do tipo `int`, que não tem conversão automática.

4. A resposta certa é (a). O código não compila, pois não existe retorno de método com dois valores como `int, int`.

Exercícios da seção 6.2

1. A resposta certa é (a). O código não compila, pois os métodos não possuem tipo de retorno definido. Típica pegadinha: parece focar em `static`, mas está focado em outra coisa.
2. A resposta certa é (b). Imprime `x`, depois `y`.
3. A resposta certa é (d). Compila e imprime `z`.
4. A resposta certa é (a). O código não compila, pois tenta acessar `this` dentro de um contexto estático.
5. A resposta certa é (b). O código compila e imprime `x e y`.

Exercícios da seção 6.3

1. A resposta certa é (a). O código não compila, já que não há sobrecarga de método ao alterar só o retorno.
2. A resposta certa é (c). Compila e imprime `15`, `15 e 15.0`.
3. A resposta certa é (a). O código não compila.
4. A resposta certa é (a). Não compila: os métodos não são estáticos.
5. A resposta certa é (d). O código não compila. O método `a` não é estático.
6. A resposta certa é (b). Compila e imprime `1`.
7. A resposta certa é (a). Não compila, pois tem várias variáveis

locais (parâmetros) com o mesmo nome.

8. A resposta certa é (b). Compila e imprime `1`.

Exercícios da seção 6.4

1. A resposta certa é (d). O código compila e joga um `NullPointerException`.

Exercícios da seção 6.5

1. A resposta certa é (a). O código não compila por causa do loop, quando um construtor de um tipo chama outro construtor do mesmo tipo em loop direto.
2. A resposta certa é (d). O código não compila, pois as classes definem parênteses a mais. Cuidado.
3. A resposta certa é (f). O código compila e não imprime nada.
4. A resposta certa é (f). O código compila e não imprime nada.
5. A resposta certa é (e). O código compila e joga exception ao entrar em loop infinito.

Exercícios da seção 6.6

1. A resposta certa é (a). Ocorre um erro de compilação na classe `Teste` ao tentar chamar o construtor com acesso `default` de outro pacote.
2. A resposta certa é (b). Compila e imprime `3`.
3. A resposta certa é (b). Não compila na declaração do método `private public`.
4. A resposta certa é (a). O código não compila, porque a classe `A` é a própria classe do método `main`, e ela não tem método `a`.

5. A resposta certa é (b). Imprime `1`, pois o método que recebe `String` não está visível no pacote principal.
6. A resposta certa é (a). Não compila, pois a palavra `default` não pode ser usada como modificadora de visibilidade de método.
7. A resposta certa é (b). O código compila e imprime `1`.

Exercícios da seção 6.7

1. A resposta certa é (b). Compila e imprime 0, são duas instâncias de `B!`.
2. A resposta certa é (c). Compila e imprime `5`.
3. A resposta certa é (b). Compila e imprime 0, existe shadowing aqui no `setter`.
4. A resposta certa é (b). Compila e imprime 0.
5. A resposta certa é (d). Compila e imprime 10, existe *shadowing* aqui no `setter`, então não há problema de a variável ser `final`.

Exercícios da seção 6.8

1. A resposta certa é (a). Não compila, pois somente variáveis podem ter aplicadas *autoincrement* e *autodecremento*.
2. A resposta certa é (c). Compila e imprime `151`.

Exercícios da seção 7.1

1. Aqui não ocorre sobrescrita. Como os parâmetros são diferentes, ocorre uma sobrecarga (não confundir na prova *overload* com *override*). Ou seja, é um *overload* com herança.
2. A reescrita é válida, pois `FileNotFoundException` é

subclasse de `IOException`.

3. A resposta certa é (a). O código não compila, porque há um ciclo na herança.
4. A resposta certa é (a). O código não compila, pois usa herança múltipla de classes, que não existe em Java.
5. A resposta certa é (c). O código compila e não imprime nada.
6. A resposta certa é (a). O código não compila, já que não existe construtor de `B` ao qual `A` tenha acesso para herdar do mesmo.
7. A resposta certa é (a). O código não compila, pois `A` não tem acesso a variável de `B`.
8. A resposta certa é (b). Compila e imprime `t`.

Exercícios da seção 7.2

1. A resposta certa é (a). O código não compila, pois faltou o `import de java.io.*`.
2. A resposta certa é (b). O código compila e imprime `b`.
3. A resposta certa é (c). O código compila e imprime `c`.
4. A resposta certa é (a). O código não compila: `C` não possui método que receba `double`.
5. A resposta certa é (a). Não compila, pois interface não pode ter método com corpo da maneira como foi definido aqui.
6. A resposta certa é (d). Compila e imprime `d`.
7. A resposta certa é (g). Compila e entra em loop.
8. A resposta certa é (b). Compila e imprime `b`, `b`, `c`.
9. A resposta certa é (a). Não compila, não existe `super.x` na classe `B`.

Exercícios da seção 7.3

1. A resposta certa é (a). Não compila, porque há um erro de *copy* e *paste* nos nomes das variáveis.
2. A resposta certa é (c). Compila e imprime `2`.
3. A resposta certa é (c). Compila e imprime `2`.
4. A resposta certa é (b). Compila e imprime `1`.
5. A resposta certa é (a). O código não compila, pois o método `close` não é público.
6. A resposta certa é (d). O código imprime `closing base`.
7. A resposta certa é (a). O código não compila pois, ao sobrescrevê-lo, tentamos definir um escopo menor. Não compila também porque o método `close` é *package protected* dentro de `Account`.
8. A resposta certa é (c). O código compila e imprime `closing savings`.

9. O código compila normalmente. Apesar de o método `turnon` não estar declarado na classe `ConcreteCar`, a classe herda estemétodo de `Car`, logo, não é necessário reescrevê-lo (poderia reescrever se achasse necessário).

   A declaração de que `ConcreteCar` implementa `Vehicle` também não era necessária, pois `Car` já implementa `Vehicle` e `ConcreteCar` é um `Car`.

Exercícios da seção 7.4

1. A resposta certa é (c). O código compila e roda e, ao rodar, não dá exception.
2. A resposta certa é (d). O código compila e roda e, ao rodar, dá exception.
3. A resposta certa é (b). O código não compila no `main : D`

até implementa `Z` e `W`, mas não implementa `Y`.

4. A resposta certa é (d). O código compila: algum subtipo de `D` pode implementar `Y`. Ao rodar, ele dá exception.
5. A resposta certa é (c). Compila, pois apesar de `B` não implementar `Z`, um subtipo dele pode (e na prática já o faz) implementá-lo. Ao rodar, não dá exception nenhuma.
6. A resposta certa é (d). O código compila e roda dando exception.
7. A resposta certa é (c). compila e roda, porém na execução lança uma `ClassCastException`, pois a classe `D` não implementa a interface `Y`.
8. A resposta certa é (b). O código não compila: `instanceof` é minúsculo.
9. A resposta certa é (e). O código compila, mas lança um `ClassCastException`, já que `D` não é um `Y`.

Exercícios da seção 7.5

1. A resposta certa é (c). O código compila e imprime `2`.
2. A resposta certa é (a). O código não compila.
3. A resposta certa é (a). O código não compila.
4. A resposta certa é (a). O código não compila: não faz sentido acessar o `} super` de outro objeto que não eu mesmo.
5. A resposta certa é (a). O código não compila, não podemos chamar o `this()` de dentro de um método.
6. A resposta certa é (a). Não compila, pois tentamos invocar dois `this`.
7. A resposta certa é (c). Compila e não imprime nada.
8. A resposta certa é (c). Compila e não imprime nada.
9. A resposta certa é (a). Não compila: não podemos referenciar um método de instância ao invocar um construtor `this`.

10. A resposta certa é (b). Compila e imprime `2`.

Exercícios da seção 7.6

1. A resposta certa é (c). Compila e imprime `2`.
2. A resposta certa é (a). A classe `B` não compila.
3. A resposta certa é (b). Compila e imprime `1`.
4. A resposta certa é (b). Compila e imprime `1`.
5. A resposta certa é (a). Não compila, pois o método é final.
6. A resposta certa é (c). Compila e imprime `b`.

Exercícios da seção 8.1

1. A resposta certa é (d). A única exception da lista que não é checked é a `IndexOutOfBoundsException`.

Exercícios da seção 8.2

1. (b) e (e) são corretas. (a) está errada, pois podemos usar exceptions mesmo sem entradas do usuário. (c) está errada, porque podemos manter o programa rodando mesmo que uma exception ocorra. (d) está errada, pois devemos usar outras estruturas para controlar o fluxo de nosso programa, como `if`, por exemplo.
2. (a) e (c) estão corretas. (b) e (d) estão incorretas por serem os opostos das certas, e (e) está incorreta, pois exceptions não são maneiras de aumentar a segurança.

Exercícios da seção 8.3

1. A resposta certa é (a). O código não compila, pois a variável local nunca foi inicializada.

2. A resposta certa é (c). Quando ocorre a exception, o fluxo desvia para imprimir `b` e depois continua normal com o `c`.
3. A resposta certa é (b). Quando ocorre a exception, o fluxo desvia para imprimir `b`, passa pelo `finally` imprimindo `c`, e depois continua normal com o `d`.

Exercícios da seção 8.4

1. A resposta certa é (a) e (c). Devemos colocar uma `java.io.IOException` ou `java.lang.Exception`.
2. A resposta certa é (b). O código compila, pois ele cria um array de dimensão 2. Ele imprime `acefdb`.
3. A resposta certa é (c). O código compila e imprime `ace`, jogando uma `Exception`.
4. A resposta certa é (a). O código não compila, já que o método `m2` deve tratar ou jogar `java.io.FileNotFoundException`.
5. A resposta certa é (a). O código não compila, porque o método `main` deve tratar ou jogar `java.io.IOException`.
6. A resposta certa é (c). Compila, imprime `ace` e joga a `Exception`.
7. A resposta certa é (e). Compila e imprime `acedb`.
8. A resposta certa é (b). Compila e imprime `acefdb`. Note que não jogamos a exception, somente a instanciamos.
9. A resposta certa é (a). Não compila: o `System.out` do `f` é `unreachable`.
10. A resposta certa é (a). Não compila: a palavra `throw` deveria ter sido usada para jogar a `Exception`.
11. A resposta certa é (c). Compila, imprime `ace` e estoura uma `Exception`.

Exercícios da seção 8.5

1. A resposta certa é (e), `StackOverflowError`.
2. A resposta certa é (d). `OutOfMemoryError`, pois tem um loop infinito.

Exercícios da seção 9.2

1. A resposta certa é (e). O método `print` não possui versão sem argumentos.

2. A resposta certa é (a). Compila e imprime:

```
ab
cd
e
```

3. A resposta certa é (c). Compila e executa normalmente.

4. A resposta certa é (b). Esta alternativa imprime `>12.45 <`.
5. A resposta certa é (b). Linhas `A` e `C` imprimem `abc`.

Exercícios da seção 9.3

1. A resposta certa é (d). Deixar os métodos públicos e atributos privados.

Exercícios da seção 10.1

1. A resposta certa é (b). Imprime `go!`.
2. A resposta certa é (c). Imprime `10`.
3. A resposta certa é (d). Imprime `11`.
4. A resposta certa é (c). Erro de compilação na linha B.
5. A resposta certa é (d). Erro de compilação na linha B.

6. A resposta certa é (a). Imprime `x`.
7. A resposta certa é (b). Imprime `5`.

Exercícios da seção 11.1

1. A resposta certa é (a). As classes são imútaveis.
2. A resposta correta é (c). Para representar horas sem data, usamos `LocalTime`.
3. A resposta certa é (b). A classe que representa a data com hora é a `LocalDateTime`, e o método para pegar a data corrente é `now`.
4. A resposta certa é (c). `ChronoUnit.DAYS.between(d1,d2);`
5. A resposta certa é (c). `ChronoUnit.DAYS.between(d1,d2);`
6. A resposta certa é (a). `Period d = Period.between(birthday, today);`
7. A resposta certa é (d).

```
LocalDateTime.ofInstant(d.toInstant(),
    ZoneId.systemDefault()).toLocalDate();
```

8. A resposta certa é (e). Não compila. O método `MonthDay.isSupported` não aceita parâmetros do tipo `ChronoUnit`, apenas `ChronoField`.
9. A resposta certa é (d). Não compila. Não existe um valor `ChronoUnit.YEAR`, e sim `ChronoUnit.YEARS`.

Exercícios da seção 11.2

1. A resposta certa é (a). `p = () ->`

```
System.out.println("Hello World");
```

2. A resposta certa é (c). `Predicate big = list -> list.size() > 100;`

3. A resposta certa é (d).

```
Calculator<Integer> divide =
    (int a, int b) -> {return (Integer) a / b;}
```

4. A resposta certa é (d). Erro de compilação.
5. A resposta certa é (b). Compila e imprime `1`.
6. A resposta certa é (d). Não compila por erro na linha A.