1Z0-808 - Notas de Estudo

Ricardo Alvarenga

2024

Como Tirar a Certificação Entre 3 e 6 Meses de Estudo

Referência: `https://www.youtube.com/watch?v=t-7GVhoM0bY&t=889s&ab_channel=Jos%C3%A9Castilho-DesenvolvedorDisputado`

Por Quê Tirar a Certificação Java

- Poucos profissionais são certificados;
- Gera reconhecimento;
- São duas certificações, a primeira **Java SE Programmer I exam (1Z0-808)**. A segunda querer a primeira e possui várias áreas distintas;

Método Sugerido

- Aprender a linguagem, não começar por livro preparatório. Usar livros básicos primeiro;
- Estudar com data definida;
- "Comer"um livro de certificação;
- Usar várias metodologias. Alguns assuntos são assimilados melhor com leitura, outros com vídeos, outros *codando* e vendo funcionar;
- Fazer muitos simulados. Simular provas até conseguir notas altas em várias provas simuladas;
- Aprender com os erros e acertos dos simulados;
- Repetir o ciclo de fazer simulados e analisar erros e acertos, várias vezes;

Sumário

Lista de Figuras

1 Java Basics

1.1 Define the scope of variables

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

1.1.1 Variáveis locais

Variáveis locais são declaradas dentro de métodos ou construtores. O ciclo de vida de uma variável local vai do ponto onde ela foi declarada até o fim do daquele bloco.

Quando a execução sai do escopo a variável não pode mais ser usada e seu valor é perdido. A qualquer momento o JVM pode realocar a memória ocupada por ela.

Bloco é um trecho de código entre chaves . Pode ser um método, um construtor, o corpo de um ***if***, de um ***for*** etc.

Analise o código abaixo e identifique os escopos de ***x*** e ***y***:

```
public void m1() { // inicio do bloco do metodo
    int x = 10; // variavel local do metodo

    if (x >= 10) { // inicio do bloco do if
        int y = 50; // variavel local do if
        System.out.print(y);
    } // fim do bloco do if
} // fim do bloco do metodo
```

Tome cuidado especial com loops ***for***. As variáveis declaradas na área de inicialização do loop só podem ser usadas no corpo do loop:

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

System.out.println(j); // erro, ja nao esta mais no escopo
```

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

```
class Teste {

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

    public void m2() {
        // erro de compilacao pois bla nao existe neste
        // escopo
        System.out.println(bla);
    }
}
```

Os parâmetros podem ser primitivos ou objetos. Um método pode ter até 255 parâmetros. Essas variáveis fazem parte do escopo do bloco do método inteiro e são definidas na declaração do método.

```java
float findMilesPerHour(float milesTraveled, float hoursTraveled){
    //o metodo contem dois parametros
    return milesTraveled / hoursTraveled;
}
```

1.1.2 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. Fazem parte do escopo da classe. Podem ser acessadas por qualquer membro da classe e ficam em escopo enquanto o objeto existir:

```java
class Pessoa {
    // variavel de instancia ou variavel de objeto
    String nome;

    public void setNome(String n) {
        // acessando a variavel de instancia no metodo
        this.nome = n;
    }
}
```

1.1.3 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).

```java
class Pessoa {
    static int id = 1;
}

class Teste {
    public static void main(String[] args) {
        Pessoa p = new Pessoa();
        System.out.println(p.id); // acessando pelo objeto. id = 1
        System.out.println(Pessoa.id); // acessando direto pela classe.
                                       // id = 1
        p.id ++; // soma 1
        System.out.println(Pessoa.id); // id = 2
        p.id ++;
        Pessoa pp = new Pessoa();
        p.id ++;
        System.out.println(pp.id); // id = 4
        System.out.println(Pessoa.id); // id = 4
    }
}
```

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.

1.1.4 Variáveis com o mesmo nome

Não é possível declarar duas variáveis no mesmo escopo com o mesmo nome:

```
public void bla() {
    int a = 0;
    int a = 10;// erro
}
```

Mas, eventualmente, podemos ter variáveis em escopos diferentes que podem ser declaradas com o mesmo nome. Em casos em que possa haver ambiguidade na hora de declará-las, o próprio compilador irá 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; // erro de compilacao,
}
...
System.out.println(new Bla().a); // qual variavel estamos acessando?
```

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

```
public void metodo(String par) {
    int par = 0; // erro de compilacao

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

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 Pessoa {

    static int x = 0;
    int y = 0;

    public static void setX(int x) {
        // Usando a referencia da classe
        Pessoa.x = x;
    }

    public void setY(int y) {
        // usando o this
        this.y = y;
    }
}
```

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 = 10;

    public void metodo() {
        int a = 20; // shadowing
        System.out.println(a); // imprime 20
    }
}
```

1.1.5 Difference Between *Life* and *Scope* For Local Variables

Life: A local variable is *alive* as long as its Stack frame is on the Stack. In other words, *until the method completes.*

Scope: A local variable is in *scope* only within the method in which the variable was declared. When its own method calls another, the variable is alive, but not in scope until its method resumes. **You can use a variable only when it is in scope**.

Cenário e Solução	
Variável mais adequada para um contador em um laço?	Variável local
Variável deve ser usada para armazenar informações sobre um objeto?	Variável de instância
Que variável deve ser usada para a passagem de informações para um método?	Parâmetro de método

1.2 Define the structure of a Java class

Nesta seção, iremos entender a estrutura de um arquivo java, onde inserir as declarações de pacotes e imports e como declarar classes e interfaces.

Para entender a estrutura de uma classe, vamos ver o arquivo ***Pessoa.java***:

```
// Declaracao de pacote
package br.com.alvarenga.certificacao;

// imports
import java.util.Date;

// Declaracao da classe
class Pessoa {
    // conteudo da classe
}
```

1.2.1 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**.

```java
// Uma classe no pacote padrao
class Pessoa {
    //...
}
```

Para definir qual o pacote a 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 ***;***. **Podem existir comentários antes da definição de um pacote**:

```java
// declaracao do pacote
package br.com.alvarenga.certificacao;

class Pessoa {
    //...
}
```

Package deve ser a primeira instrução de código que temos declarada em nosso arquivo. Comentários não são considerados parte do código, portanto, podem existir em qualquer lugar do arquivo Java sem restrições.

Formas para inserir comentários:

```java
// comentario de linha

/*
comentario de
multiplas linhas
*/
class /* comentario no meio da linha */ Pessoa {

    /**
    *  JavaDoc, repare que a primeira linha do comentario tem
    *  2 asteriscos
    */
    public void metodo() {
    }
}
```

1.2.2 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 **{}**:

```java
class Pessoa {}
```

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 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**.:

```java
class Pessoa {

    String nome;
    String sobrenome;

    Pessoa(String nome, String sobrenome) {
        this.nome = nome;
        this.sobrenome = sobrenome;
    }

    public String getNomeCompleto() {
        return this.nome + this.sobrenome;
    }
}
```

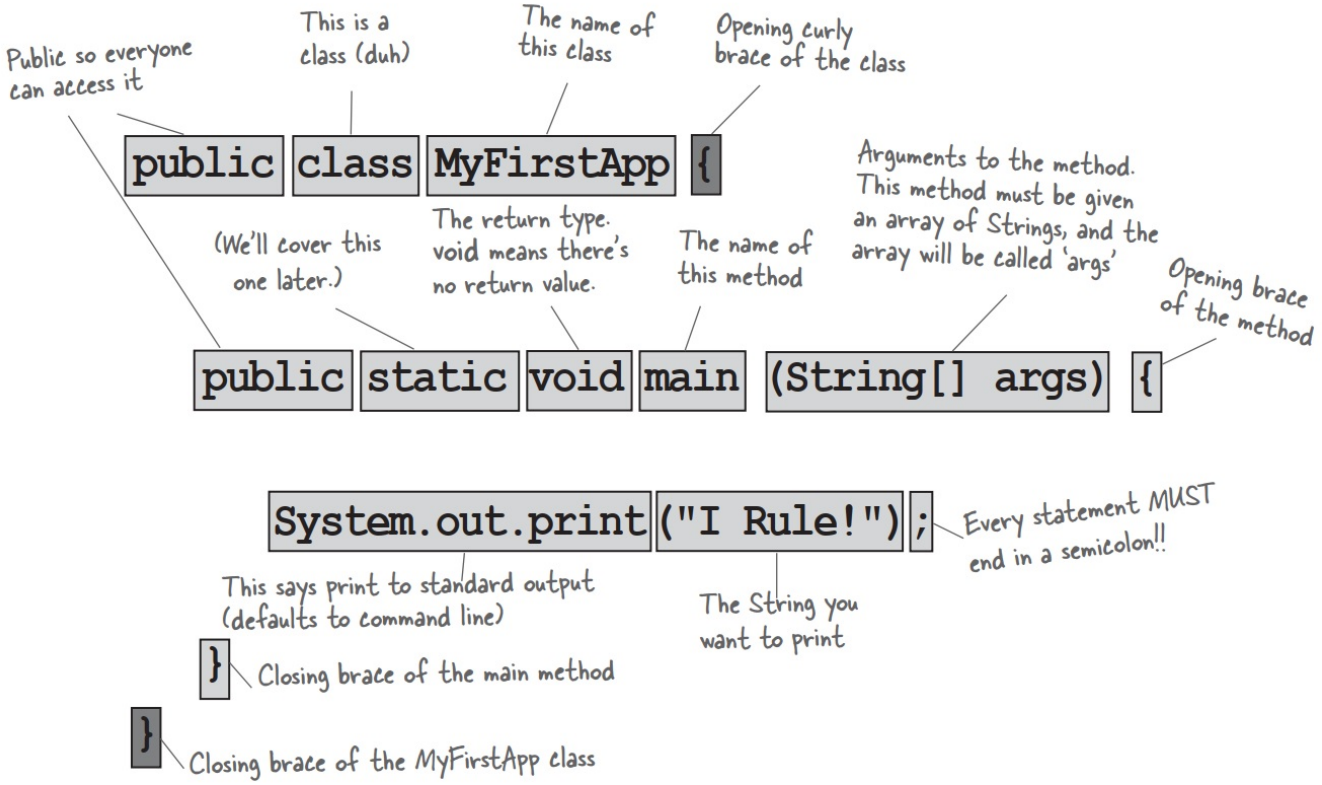

Figura 1: Anatomy of a class

In Java, everything goes in a class. You'll type your source code file (with a *.java* extension), then compile it into a new class file (with a *.class* extension). When you run your program, you're really running a class.

1.2.3 Variáveis

No exemplo da classe Pessoa definida anteriormente, *nome* e *sobrenome* são **variáveis**. A declaração de variáveis: *tipo nome_da_variável;*.

Essas são variáveis de instância, pois existe uma cópia delas para cada objeto *Pessoa* criado. Cada cópia guarda o estado de uma certa instância desses objetos.

Há 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***.

1.2.4 Métodos

Na declaração de métodos precisamos do tipo do retorno, seguido do nome do método e seguido 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, onde está definido o retorno, o nome e os parâmetros é onde temos a assinatura do método. ***A assinatura de um método inclui somente o nome do método e os tipos dos parâmetros***. Métodos também podem ser ***static***.

1.2.5 Construtores

Uma classe pode possuir zero ou vários construtores. A classe *Pessoa* possui 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**.

1.2.6 Métodos com o mesmo nome da classe

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 seja maior que ***1***, o valor de ***j*** será ***-100***, caso contrário, será o mesmo valor de ***i***.

1.2.7 Interfaces

Para definir uma interface usamos a palavra reservada ***interface***:

```
interface Autenticavel {

    final int TAMANHO_SENHA = 8;

    void autentica(String login, String senha);
}
```

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.

1.2.8 Múltiplas estruturas em um arquivo

Em java, é possível definir mais de uma classe/interface em um mesmo arquivo java, 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.

```java
// arquivo1.java
interface Bar {}

class Foo {}
```

```java
// Foo.java
public class Foo {}

interface X {}
```

1.2.9 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***.

1.2.10 Convenções de nomenclatura

Element	Lettering	Characteristic	Example
Class name	Begins uppercase, continues CamelCase	Noun	`SpaceShip`
Interface name	Begins uppercase, continues CamelCase	Adjective ending with "able" or "ible" when providing a capability; otherwise a noun	`Dockable`
Method name	Begins lowercase, continues CamelCase	Verb, may include adjective or noun	`orbit`
Instance and static variables names	Begins lowercase, continues CamelCase	Noun	`moon`
Parameters and local variables	Begins lowercase, continues CamelCase if multiple words are necessary	Single words, acronyms, or abbreviations	`lop` (line of position)
Generic type parameters	Single uppercase letter	The letter *T* is recommended	`T`
Constant	All uppercase letters	Multiple words separated by underscores	`LEAGUE`
Enumeration	Begins uppercase, continues CamelCase; the set of objects should be all uppercase	Noun	`enum Occupation {MANNED, SEMI_MANNED, UNMANNED}`
Package	All lowercase letters	Public packages should be the reversed domain name of the org	com.ocajexam.sim

Figura 2: Nomenclatura Java

Symbol	Description	Purpose
( )	Parentheses	Encloses set of method arguments, encloses cast types, adjusts precedence in arithmetic expressions
{ }	Braces	Encloses blocks of codes, initializes arrays
[ ]	Box brackets	Declares array types, initializes arrays
< >	Angle brackets	Encloses generics
;	Semicolon	Terminates statement at the end of a line
,	Comma	Separates identifiers in variable declarations, separates values, separates expressions in a for loop
.	Period	Delineates package names, selects an object's field or method, supports method chaining
:	Colon	Follows loop labels
' '	Single quotes	Defines a single character
->	Arrow operator	Separates left-side parameters from the right-side expression
" "	Double quotes	Defines a string of characters
//	Forward slashes	Indicates a single-line comment
/* */	Forward slashes with asterisks	Indicates a blocked comment for multiple lines
/** */	Forward slashes with a double and single asterisk	Indicates Javadoc comments

Figura 3: Símbolos e separadores

1.3 Create executable Java applications with a main method; run a Java program from the command line; produce console output

- Compilar *javac* ***nome_arquivo.java***
- Executar *java* ***nome_pacote.nome_classe***
- Método ***main()*** deve ser ***public static void***

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.

1.3.1 Método *main()*

Rregras 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...*).

Exemplos:

```
    //Parametro como array
    public static void main (String[] args) {}

    //Parametro como varargs
    public static void main (String... args) {}

    //A ordem dos modificadores nao importa
    static public void main(String[] args) {}

    //O nome do parametro nao importa
    public static void main (String... argumentos){}

    //Tambem e uma definicao valida de array
    public static void main (String args[]) {}
```

1.3.2 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*. Exemplo com arquivo *HelloWorld.java*:

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

1.3.3 Passando parâmetros pela linha de comando

```
public class HelloWorld{
    public static void main(String[] args) {
        //Lendo o valor da primeira posicao do array args
        System.out.println("Hello " + args[0] + "!");
    }
}
```

Informando o parâmetro:

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

Os parâmetros devem ser separados por espaço. Cada parâmetro informado será armazenado em uma posição do array, na mesma ordem em que foi informado.

1.3.4 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.

1.3.5 javac

Arquivo *Prova.java* dentro do diretório do projeto:

```
1 class Prova {
2     double tempo;
3 }
```

```
1 $ javac Prova.java
```

O bytecode da classe *Prova* gerado na compilação é colocado no arquivo *Prova.class* dentro do diretório de trabalho, projeto:

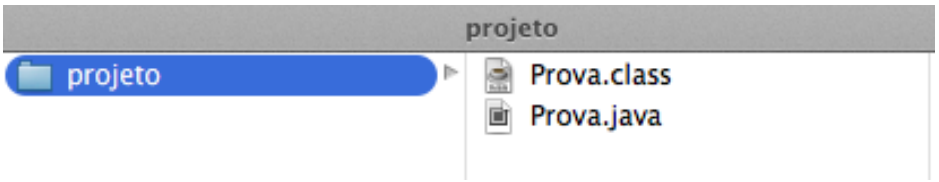

Figura 4: Pasta de trabalho

Vejamos qual é o comportamento do *javac* com a utilização de pacotes. Colocamos o arquivo *Prova.java* no diretório *certificacao*:

```
1 package certificacao;
2 class Prova {
3     double tempo;
4 }
```

```
1 [certificacao]$ javac certificacao/Prova.java
```

Nesse exemplo, o arquivo *Prova.class* é colocado no diretório certificacao.

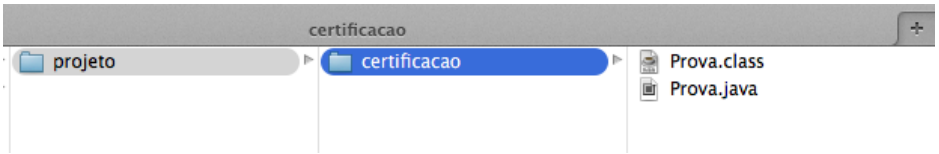

Figura 5: Pasta de trabalho com *package*

1.3.6 Escolhendo a versão do Java na hora de compilar

Na 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 MinhaClasse.java -source 1.3*).

1.3.7 java

Como exemplo do comando *java*, criando o arquivo *Teste.java* no mesmo diretório, no mesmo pacote:

```
package certificacao;
class Teste {
    public static void main(String[] args) {
        Prova p = new Prova();
        p.tempo = 210;
        System.out.println(p.tempo);
    }
}
```

```
$ javac certificacao/Teste.java
$ java certificacao.Teste
```

```
210.0
```

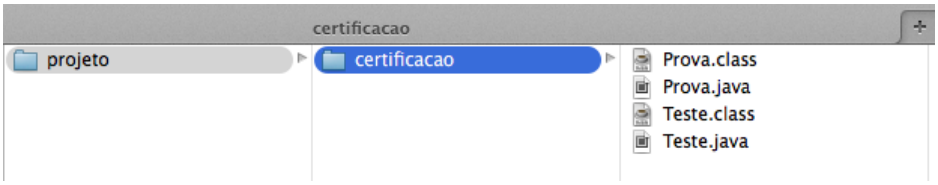

Figura 6: Pasta de trabalho com *package* e várias classes

Somente o arquivo *Teste.java* foi passado para o compilador. Nesse arquivo, a classe *Teste* utiliza a classe *Prova* que se encontra em outro arquivo, *Prova.java*. Dessa forma, o compilador vai compilar automaticamente o arquivo *Prova.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*.

1.3.8 Propriedades na linha de comando

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

```
java -Dchave1=abc -Dchave2=def Foo xpto bar
```

chave1=abc e ***chave2=def*** são parâmetros/propriedades e ***xpto*** e ***bar*** são argumentos recebidos pelo método ***main***.

1.3.9 Classpath

Para compilar ou para 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 (**.**).

1.3.10 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 /diretorio/biblioteca.jar Prova.java
$ java -cp /diretorio/biblioteca.jar Prova
```

E podemos passar tanto caminhos de outras pastas como de JARs ou zips. Para passar mais de um parâmetro 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 /diretorio/biblioteca.jar;/outrodir/ scjp/Prova.java
$ java -cp /diretorio/biblioteca.jar;/outrodir/ scjp.Prova
```

1.3.11 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 scjp:

```
jar -cf bib.jar scjp
```

Para utilizar o *jar*:

```
java -cp bib.jar scjp.Prova
```

1.3.12 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 ao nosso *jar*. E dentro dela, cria o arquivo *Manifest.mf*.

Esse arquivo pode ser usado para algumas configurações. Por exemplo, é possível dizer qual classe do nosso *jar* é a classe 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: scjp.Teste
```

Para gerar o *jar*:

```
jar -cmf bib.jar meumanifest scjp
```

Para rodar um *jar* com *Main-Class*, basta usar:

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

1.3.13 Perguntas

1) Como compilar e rodar o arquivo *A.java*, existente no diretório *b*?
R: *javac b/A.java* e *java b.A*

2) O que pode acontecer ao tentarmos executar o código a seguir?

```
1 class A {
2     public static void main(String[] args) {
3         System.out.println(args);
4         System.out.println(args.length);
5         System.out.println(args[0]);
6     }
7 }
```

R: Ao rodar sem argumentos, ocorre uma *ArrayIndexOutOfBoundsException* na linha 5.

3) Como rodar a classe *A.java* presente no diretório *b*, que foi compactado em um arquivo chamado *programa.jar*, sendo que não existe nenhum arquivo de manifesto?

```
1 package b;
2 class A {
3     public static void main(String[] args) {
4         System.out.println(args[0]);
5     }
6 }
```

R: 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 programa.jar b.A*.

4) Como compilar a classe A.java, definida como no pacote b presente no diretório b, e adicionar também o arquivo programa.jar na busca de classes durante a compilação? Lembre-se que . significa o diretório atual.
R: Durante a compilação, para adicionar o arquivo *programa.jar* ao *classpath*, devemos usar *-cp programa.jar* e, para especificar o arquivo adequado, usamos *b/A.java*.
Ficando assim: *javac -cp programa.jar:. b/A.java*.

1.4 Import other Java packages to make them accessible in your code

1.5 Compare and contrast the features and components of Java such as: platform independence, object orientation, encapsulation, etc.