Fundamental

1. Learning About the Java Environment

El entorno Java consiste en comprender una serie de tecnologías.

1.1. MAJOR COMPONENTS OF JAVA

El *Java Development Kit* (JDK) contiene el software mínimo que necesita para realizar el desarrollo de Java. Las piezas clave incluyen el compilador (`javac`), que convierte archivos .java en archivos `.class`, y el lanzador `java`, que crea la máquina virtual y ejecuta el programa. El JDK también contiene otras herramientas, incluido el comando `archiver` (`jar`), que puede empaquetar archivos juntos, y el comando de documentación API (`javadoc`) para generar documentación.

El programa `javac` genera instrucciones en un formato especial que el comando `java` puede ejecutar llamado `bytecode`. Luego, java lanza la máquina virtual Java (JVM) antes de ejecutar el código.

WHERE DID THE JRE GO?
En versiones anteriores de Java, podía descargar un entorno de ejecución de Java (JRE) en lugar del JDK completo.

En Java 11, el JRE ya no está disponible como descarga independiente o como subdirectorio del JDK. Las personas pueden usar el JDK completo cuando ejecutan un programa Java.

Al escribir un programa, existen algunas funciones y algoritmos comunes que los desarrolladores necesitan. Java viene con un gran conjunto de interfaces de programación de aplicaciones (API) que puede utilizar. Por ejemplo, hay una clase `StringBuilder` para crear un `String` grande y un método en Colecciones para ordenar una lista.

1.2. DOWNLOADING A JDK

Cada seis meses, se incrementa el número de versión de Java. Java 11 salió a la luz en septiembre de 2018.
Cada tres años, Oracle tiene una versión de soporte a largo plazo (LTS). A diferencia de las versiones que no son LTS y que son compatibles solo durante seis meses, las versiones LTS tienen parches y actualizaciones disponibles durante al menos tres años.

Recomendamos utilizar la distribución de Oracle de Java 11 para estudiar para este examen. Alternativamente, puede usar OpenJDK, que se basa en el mismo código fuente.

2. Identifying Benefits of Java

Java tiene algunos beneficios clave que deberá conocer para el examen.

Orientado a objetos: Java es un lenguaje orientado a objetos, lo que significa que todo el código se define en clases, y la mayoría de esas clases se pueden instanciar en objetos. Java permite la programación funcional dentro de una clase, pero la orientación a objetos sigue siendo la principal organización del código.

Encapsulación: Java admite modificadores de acceso para proteger los datos de accesos y modificaciones no intencionales.

Plataforma independiente: Java es un lenguaje interpretado que se compila en `bytecode`. Un beneficio clave es que el código Java se compila una vez en lugar de tener que volver a compilar para diferentes sistemas operativos. Esto se conoce como "escribir una vez, ejecutar en todas partes". La portabilidad le permite compartir fácilmente piezas de software precompiladas. Si le preguntan sobre cómo ejecutar Java en diferentes sistemas operativos, la respuesta es que los mismos archivos de clase se ejecutan en todas partes.

Robusto: Una de las principales ventajas de Java sobre C ++ es que evita pérdidas de memoria. Java administra la memoria por sí solo y realiza la recolección de basura automáticamente. La mala gestión de la memoria en C ++ es una gran fuente de errores en los programas.

Simple: Se pretendía que Java fuera más sencillo de entender que C ++. Además de eliminar los punteros, eliminó la sobrecarga de operadores.

Seguro: El código Java se ejecuta dentro de la JVM. Esto crea una caja de arena que dificulta que el código Java haga cosas malas en la computadora en la que se ejecuta.

Multiproceso: Java está diseñado para permitir que varios fragmentos de código se ejecuten al mismo tiempo. También hay muchas API para facilitar esta tarea.

Compatibilidad con versiones anteriores: Los arquitectos del lenguaje Java prestan especial atención a asegurarse de que los programas antiguos funcionen con versiones posteriores de Java. Si bien esto no siempre ocurre, los cambios que romperán la compatibilidad con versiones anteriores ocurren lentamente y con aviso. La obsolescencia (*deprecated*) es una técnica para lograr esto donde el código está marcado para indicar que no debe usarse.

3. Understanding the Java Class Structure

En los programas Java, las clases son los bloques de construcción básicos.

Un objeto es una instancia en tiempo de ejecución de una clase en la memoria. A menudo se hace referencia a un objeto como instancia, ya que representa una única representación de la clase. Todos los diversos objetos de todas las diferentes clases representan el estado de su programa. Una referencia es una variable que apunta a un objeto.

3.1. FIELDS AND METHODS

Las clases de Java tienen dos elementos principales: métodos, a menudo llamados funciones o procedimientos en otros lenguajes, y campos, más generalmente conocidos como variables. Juntos, estos se denominan miembros de la clase. Las variables mantienen el ***estado del programa*** y los métodos operan en ese estado.

Otros bloques de construcción incluyen interfaces, enumeraciones.

La clase Java más simple que puede escribir tiene este aspecto:

```
1: public class Animal {
2: }
```

Java llama a una palabra con un significado especial una palabra clave (*keyword*). Otras clases pueden usar esta clase ya que hay una palabra clave pública en la línea 1.

```
1: public class Animal {
2:    String name;
3: }
```

Definimos una variable llamada `name`. También definimos el tipo de esa variable para que sea una `String`.

```
1: public class Animal {
2:    String name;
3:    public String getName() {
4:       return name;
5:    }
6:    public void setName(String newName) {
7:       name = newName;
8:    }
9: }
```

Un método es una operación que se puede llamar. Nuevamente, `public` se usa para significar que este método se puede llamar desde otras clases. Luego viene el tipo de retorno. La palabra clave `void` significa que no se devuelve ningún valor. Este método requiere que se le proporcione información desde el método de llamada; esta información se llama parámetro.

Dos piezas del método son especiales. El nombre del método y los tipos de parámetros se denominan firma del método (*method signature*).

3.2. COMMENTS

Hay tres tipos de comentarios en Java. El primero se llama comentario de una sola línea:

```
// comment until end of line
```

Luego viene el comentario de varias líneas:

```
/* Multiple
 * line comment
 */
```

Un comentario de varias líneas (también conocido como comentario de varias líneas) incluye cualquier cosa que comience desde el símbolo `/**` hasta el símbolo `* /`.

```
/**
 * Javadoc multiple-line comment
 * @author Jeanne and Scott
 */
```

Este comentario es similar a un comentario de varias líneas, excepto que comienza con **`/**`**.

3.3. CLASSES VS. FILES

La mayoría de las veces, cada clase Java se define en su propio archivo `.java`. Suele ser público, lo que significa que cualquier código puede llamarlo. Por ejemplo, esta clase está bien:

```
1: class Animal {
2:    String name;
3: }
```

Incluso puede poner dos clases en el mismo archivo. Cuando lo hace, como máximo una de las clases en el archivo puede ser pública:

```
1: public class Animal {
2:    private String name;
3: }
4: class Animal2 {
5: }
```

Si tiene una clase pública, debe coincidir con el nombre del archivo.

4. Writing a main() Method

Un programa Java comienza a ejecutarse con su método `main()`. Un método `main()` es la puerta de enlace entre el inicio de un proceso Java, que es administrado por la Máquina Virtual Java (JVM).

4.1. CREATING A MAIN() METHOD

El método `main()` permite que la JVM llame a nuestro código:

```
1: public class Zoo {
2:    public static void main(String[] args) {
3:
4:    }
5: }
```

Para compilar y ejecutar este código, escríbalo en un archivo llamado `Zoo.java`:

```
javac Zoo.java
java Zoo
```

Para simplificar las cosas por ahora, seguiremos este subconjunto de reglas:

- Cada archivo puede contener solo una clase pública.
- El nombre del archivo debe coincidir con el nombre de la clase, incluido Mayúsculas y Minúsculas, y tener una extensión `.java`.

La palabra clave `public` es lo que se denomina modificador de acceso. Declara el nivel de exposición de este método a posibles llamantes en el programa. Naturalmente, público significa en cualquier lugar del programa.

La palabra clave static vincula un método a su clase para que pueda ser llamado solo por el nombre de la clase, como en, por ejemplo, `Zoo.main()`. Java no necesita crear un objeto para llamar al método `main()`.

Un método `main()` no estático también podría ser invisible desde el punto de vista de la JVM.

La palabra clave `void` representa el tipo de retorno. Un método que no devuelve datos devuelve el control a la persona que llama en silencio.

llegamos a la lista de parámetros del método `main()`, representada como una matriz de objetos `java.lang.String`. En la práctica, puede escribir cualquiera de los siguientes:

```
String[] args
String args[]
String... args;
```

4.2. PASSING PARAMETERS TO A JAVA PROGRAM

Todos los argumentos de la línea de comandos se tratan como objetos `String`, incluso si representan otro tipo de datos como un número:

```
javac Zoo.java
java Zoo Zoo 2
```

4.3. RUNNING A PROGRAM IN ONE LINE

A partir de Java 11, puede ejecutar un programa sin compilarlo primero, bueno, sin escribir el comando `javac`. Creemos una nueva clase:

```
public class SingleFileZoo {
   public static void main(String[] args) {
      System.out.println("Single file: " + args[0]);
   }
}
```

Podemos ejecutar nuestro ejemplo de `SingleFileZoo` sin tener que compilarlo:

```
java SingleFileZoo.java Cleveland
```

Observe cómo este comando pasa el nombre del archivo Java. Al ejecutarlo como una sola línea, escribimos java `SingleFileZoo.java`. Esta función se denomina iniciar programas de código fuente de un solo archivo.

¡Tener que escribir una línea para ejecutarlos en lugar de dos será un alivio! Sin embargo, compilar su código de antemano usando `javac` hará que el programa se ejecute más rápido, y definitivamente querrá hacerlo para programas reales.

Full command	**Single-file source-code command**
`javac HelloWorld.java` `java HelloWorld`	*java HelloWorld.java*
Produce un archivo `.class`	Todo en memoria
Para cualquier programa	Para programas de un solo archivo
Puede importar Código de cualquier librería Java.	Puede importar solo Código que viene con el JDK

5. Understanding Package Declarations and Imports

Java pone clases en paquetes. Estos son agrupamientos lógicos para clases.

Suponga que intenta compilar este código:

```
public class ImportExample {
   public static void main(String[] args) {
      Random r = new Random();   // DOES NOT COMPILE
      System.out.println(r.nextInt(10));
   }
}
```

El compilador de Java te da un error que se parece a esto:

```
Random cannot be resolved to a type
```

Intentar esto nuevamente con la importación le permite compilar:

```
import java.util.Random;  // import nos dice donde encontrar Random
public class ImportExample {
   public static void main(String[] args) {
      Random r = new Random();
      System.out.println(r.nextInt(10));  // imprime number 0-9
   }
}
```

Ahora se ejecuta el código; imprime un número aleatorio entre 0 y 9.

Si comienza con java o javax, significa que vino con el JDK. Si comienza con otra cosa, probablemente muestre de dónde vino usando el nombre del sitio web al revés. Por ejemplo, `com.amazon.javabook` nos dice que el código proviene de Amazon.com.

Verá nombres de paquetes en el examen que no siguen esta convención. No se sorprenda al ver nombres de paquetes como `a.b.c`. Técnicamente, se le permite un par de caracteres más entre los puntos (.).

5.1. WILDCARDS

Classes in the same package are often imported together. You can use a shortcut to import all the classes in a package:

```
import java.util.*;    // imports java.util.Random además de otras cosas
public class ImportExample {
   public static void main(String[] args) {
      Random r = new Random();
      System.out.println(r.nextInt(10));
   }
}
```

El * es un comodín que coincide con todas las clases del paquete. No importa paquetes, campos o métodos secundarios; solo importa clases. Podría pensar que incluir tantas clases ralentiza la ejecución de su programa, pero no es así. El compilador averigua qué se necesita realmente.

Podría pensar que incluir tantas clases ralentiza la ejecución de su programa, pero no es así. El compilador averigua qué se necesita realmente.

5.2. REDUNDANT IMPORTS

Hay un paquete especial en el mundo de Java llamado `java.lang`. Este paquete es especial porque se importa automáticamente. Por ejemplo:

```
1:  import java.lang.System;
2:  import java.lang.*;
3:  import java.util.Random;
4:  import java.util.*;
5:  public class ImportExample {
6:     public static void main(String[] args) {
7:        Random r = new Random();
8:        System.out.println(r.nextInt(10));
9:     }
10: }
```

La respuesta es que tres de las importaciones son redundantes. Líneas 1, 2 y 4.

¿Qué importaciones crees que funcionarían para compilar este código?

```
public class InputImports {
   public void read(Files files) {
      Paths.get("name");
   }
}
```

Hay dos respuestas posibles:

```
// Primera:
import java.nio.file.*;

//Segunda:
import java.nio.file.Files;
import java.nio.file.Paths;
```

Ahora consideremos algunas importaciones que no funcionan:

```
import java.nio.*;          // NO GOOD - wildcard sólo hace match con
                            // nombres de clases, no con "file.Files"

import java.nio.*.*;        // NO GOOD – Sólo se puede tener un wildcard
                            // y debe estar al final

import java.nio.file.Paths.*; // NO GOOD – No puede importar métodos
                              // sólo nombres de clases
```

5.3. NAMING CONFLICTS

Una de las razones para usar paquetes es que los nombres de las clases no tienen que ser únicos en todo Java. ¿Qué importación podríamos usar si queremos la versión `java.util.Date`?

```
public class Conflicts {
   Date date;
   // some more code
}
```

Puede escribir `import java.util.*;` o `import java.util.Date;`. Los casos complicados surgen cuando hay otras importaciones presentes:

```
import java.util.*;
import java.sql.*; // causa que la declaración Date no compile
```

Java te da un error de compilación.

```
error: reference to Date is ambiguous
   Date date;
   ^
   both class java.sql.Date in java.sql and class java.util.Date in java.util match
```

Pero, ¿qué hacemos si necesitamos un montón de otras clases en el paquete `java.sql`?

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

Ah, ahora funciona. Si importa explícitamente un nombre de clase, tiene prioridad sobre cualquier comodín presente.

…pero:

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

Debido a que no puede haber dos valores predeterminados, el compilador le dice lo siguiente:

```
error: reference to Date is ambiguous
   Date date;
   ^
   both class java.util.Date in java.util and class java.sql.Date in java.sql match
```

IF YOU REALLY NEED TO USE TWO CLASSES WITH THE SAME NAME
A veces, realmente desea usar Date de dos paquetes diferentes. Cuando esto sucede, puede elegir uno para usar en la importación y usar el nombre de clase completo del otro. Aquí tienes un ejemplo:

```
// Primera forma:
import java.util.Date;
public class Conflicts {
   Date date;
   java.sql.Date sqlDate;

}

//Segunda forma:
public class Conflicts {
   java.util.Date date;
   java.sql.Date sqlDate;

}
```

5.4. CREATING A NEW PACKAGE

Hasta ahora, todo el código que hemos escrito en este capítulo ha estado en el paquete predeterminado. Este es un paquete especial sin nombre que debe usar solo para código desechable. Supongamos que tenemos estas dos clases en el directorio `C:\temp`:

```
package packagea;
public class ClassA {
}

package packageb;
import packagea.ClassA;
public class ClassB {
   public static void main(String[] args) {
      ClassA a;
      System.out.println("Got it");
   }
}
```

En este caso, ejecutar desde `C:\temp` funciona porque tanto el paquete como el paquete están debajo.

5.5. COMPILING AND RUNNING CODE WITH PACKAGES

Para las clases anteriores, si ejecutamos:

Step	Windows	Mac/Linux
Creamos `ClassA`.	`C:\temp\packagea\ClassA.java`	`/tmp/packagea/ClassA.java`
Creamos `ClassB`.	`C:\temp\packageb\ClassB.java`	`/tmp/packageb/ClassB.java`
No ubicamos en el directorio superior	`cd C:\temp`	`cd /tmp`

Para compilar escriba el siguiente código:
`javac packagea/ClassA.java packageb/ClassB.java`

COMPILING WITH WILDCARDS
Puede utilizar un asterisco para especificar que le gustaría incluir todos los archivos Java en un directorio. Podemos reescribir el comando `javac` anterior así:

```
javac packagea/*.java packageb/*.java
```

Sin embargo, no puede utilizar un comodín para incluir subdirectorios.

Ahora que su código se ha compilado, puede ejecutarlo escribiendo el siguiente comando:

```
java packageb.ClassB
```

A continuación, se muestra dónde se crearon los archivos `.class` en la estructura de directorios.

```
+ packagea
+-- ClassA.java
+-- ClassA.class
+ packageb
+-- ClassB.java
+-- ClassB.class
```

5.6. USING AN ALTERNATE DIRECTORY

De forma predeterminada, el comando `javac` coloca las clases compiladas en el mismo directorio que el código fuente. También proporciona una opción para colocar los archivos de clase en un directorio diferente. La opción -d especifica este directorio de destino.

Por ejemplo, para el comando:

```
javac -d classes packagea/ClassA.java packageb/ClassB.java
```

```
+ packagea
+-- ClassA.java
+ packageb
+-- ClassB.java
+ classes
+---+ packagea
    +-- ClassA.class
    + packageb
    +-- ClassB.class
```

Para ejecutar el programa, especifica la ruta de clases para que Java sepa dónde encontrar las clases. Hay tres opciones que puede utilizar. Los tres hacen lo mismo:

```
java -cp classes packageb.ClassB
java -classpath classes packageb.ClassB
java --class-path classes packageb.ClassB
```

Observe que el último requiere dos guiones (--), mientras que los dos primeros requieren un guion (-).

THREE CLASSPATH OPTIONS
Observe que el último requiere dos guiones (--), mientras que los dos primeros requieren un guion (-).

Opciones que se deben saber para `javac`:

Opción	Descripción

`-cp <classpath>`
`-classpath <classpath>`
`--class-path <classpath>`
	Ubicación de las clases necesarias para compilar el programa.

`-d <dir>`	Directorio para colocar archivos de clases generados

Opciones que se deben saber para `java`:

Opción	Descripción

`-cp <classpath>`
`-classpath <classpath>`
`--class-path <classpath>`
	Ubicación de las clases necesarias para ejecutar el programa.

5.7. COMPILING WITH JAR FILES

También puede especificar la ubicación de los otros archivos explícitamente utilizando una ruta de clase. Esta técnica es útil cuando los archivos de clases se encuentran en otro lugar o en archivos JAR especiales. Un archivo Java Archive (JAR) es como un archivo zip de principalmente archivos de clase Java. Por ejemplo:

```
//Para windows
java -cp ".;C:\temp\someOtherLocation;c:\temp\myJar.jar" myPackage.MyClass
//Para Linux/Mac
java -cp ".:/tmp/someOtherLocation:/tmp/myJar.jar" myPackage.MyClass
```

El punto (.) Indica que desea incluir el directorio actual en la ruta de clases.

Al igual que cuando está compilando, puede usar un comodín (`*`) para hacer coincidir todos los archivos JAR en un directorio. Aquí tienes un ejemplo:

```
java -cp "C:\temp\directoryWithJars\*" myPackage.MyClass
```

No incluirá ningún JAR en la ruta de clase que esté en un subdirectorio de `directoryWithJars`.

5.8. CREATING A JAR FILE

Puede utilizar la forma corta o larga para cada opción:

```
jar -cvf myNewFile.jar .
jar --create --verbose --file myNewFile.jar .
```

Alternativamente, puede especificar un directorio en lugar de usar el directorio actual:

```
jar -cvf myNewFile.jar -C dir .
```

Opciones que se deben saber para `jar`:

Opción	Descripción
`-c` `--create`	Crea un nuevo archivo JAR
`-v` `--verbose`	Imprime detalles al trabajar con archivos JAR
`-f <fileName>` `--file <fileName>`	Nombre de archivo JAR
`-C <directory>`	Directorio que contiene archivos que se utilizarán para crear el JAR

5.9. RUNNING A PROGRAM IN ONE LINE WITH PACKAGES

Puede usar programas de código fuente de un solo archivo desde dentro de un paquete siempre que se basen solo en las clases proporcionadas por el JDK. Por ejemplo:

```
package singleFile;
import java.util.*;

public class Learning {
   private ArrayList list;
   public static void main(String[] args) {
      System.out.println("This works!");
   }
}
```

Puede ejecutar cualquiera de estos comandos:

```
java Learning.java            // desde dentro del directorio singleFile
java singleFile/Learning.java // desde el directorio de arriba singleFile
```

6. Ordering Elements in a Class

Echemos un vistazo al orden correcto para escribirlos en un archivo:

Elemento	Ejemplo	¿Requerido?	¿Dónde va?
`package`	`package abc;`	No	Primera línea del archivo
`Import`	`import java.util.*;`	No	Inmediatamente después del paquete (si está presente)
`class`	`public class C`	Yes	Inmediatamente después de la importación (si corresponde)
Campos	`int value;`	No	Cualquier elemento de nivel superior en una clase
Métodos	`void method()`	No	Cualquier elemento de nivel superior en una clase

7. Code Formatting on the Exam

No todas las preguntas incluirán declaraciones e importaciones de paquetes. No se preocupe si faltan estados de cuenta de paquetes o importaciones a menos que se le pregunte acerca de ellos.

- Código que comienza con un nombre de clase.
- Código que comienza con una declaración de método
- Código que comienza con un fragmento de código que normalmente estaría dentro de una clase o método

Ejemplo:

```
public class MissingImports {
   Date date;
   public void today() {}
}
```

¡Si! La pregunta no era sobre importaciones, por lo que debe asumir que `import java.util` está presente.

También tenga en cuenta que las importaciones no se eliminarán para ahorrar espacio si la declaración del paquete está presente. Esto se debe a que las importaciones van después de la declaración del paquete.

Verá un código que no tiene un método `main()`. Cuando esto suceda, asuma que cualquier código de plomería necesario como el método `main()` y la definición de clase se escribieron correctamente.

Otra cosa que hace el examen para ahorrar espacio es combinar código en la misma línea. Debería esperar ver un código como el siguiente y que se le pregunte si se compila. Por ejemplo:

```
6: public void getLetter(ArrayList list) {
7:    if (list.isEmpty()) { System.out.println("e");
8:    } else { System.out.println("n");
9: }  }
```

Además, aún puede asumir que la definición de clase necesaria y las importaciones están presentes. Ahora, ¿qué pasa con este? ¿Se compila?

```
1: public class LineNumbers {
2:    public void getLetter(ArrayList list) {
3:       if (list.isEmpty()) { System.out.println("e");
4:       } else { System.out.println("n");
5: }  } }
```

Para este, respondería "No compila". Dado que el código comienza con la línea 1, no puede suponer que antes se proporcionaron importaciones válidas.

Nota
Recuerde que los espacios en blanco adicionales no importan en la sintaxis de Java. El examen puede usar cantidades variables de espacios en blanco para engañarlo.

8. Review Questions

1. Which of the following are true? (Choose all that apply.)
   A. `javac` compiles a .class file into `a .java` file.
   B. `javac` compiles a .java file into `a .bytecode` file.
   C. `javac` compiles a .java file into `a .class` file.
   D. java accepts the name of the class as a parameter.
   E. java accepts the filename of the `.bytecode` file as a parameter.
   F. java accepts the filename of the `.class` file as a parameter.

2. Which of the following are true if this command completes successfully assuming the CLASSPATH is not set? (Choose all that apply.)
```
java MyProgram.java
```
   A. A `.class` file is created.
   B. `MyProgram` can reference classes in the package `com.sybex.book`.
   C. `MyProgram` can reference classes in the package `java.lang`.
   D. `MyProgram` can reference classes in the package `java.util`.
   E. None of the above. The program needs to be run as java `MyProgram`.

3. Given the following classes, which of the following can independently replace INSERT IMPORTS HERE to make the code compile? (Choose all that apply.)
```
// File 1
package aquarium;
public class Tank { }

// File 2
package aquarium.jellies;
public class Jelly { }

// File 3
package visitor;
INSERT IMPORTS HERE
public class AquariumVisitor {
  public void admire(Jelly jelly) { } }

A. import aquarium.*;
B. import aquarium.*.Jelly;
C. import aquarium.jellies.Jelly;
D. import aquarium.jellies.*;
E. import aquarium.jellies.Jelly.*;
F. None of these can make the code compile.
```

4. Given the following classes, what is the maximum number of imports that can be removed and have the code still compile?
```
//File 1
package aquarium;
public class Water { }

//File 2
package aquarium;
import java.lang.*;
import java.lang.System;
```

```
import aquarium.Water;
import aquarium.*;
public class Tank {
  public void print(Water water) {
     System.out.println(water); } }
```

A. 0
B. 1
C. 2
D. 3
E. 4
F. Does not compile

5. Suppose we have the following class in the file /my/directory/named/A/Bird.java. Which of the answer options replaces INSERT CODE HERE when added independently if we compile from /my/directory? (Choose all that apply.)

```
INSERT CODE HERE
```

```
A. public class Bird { }
B. package my.directory.named.a;
C. package my.directory.named.A;
D. package named.a;
E. package named.A;
F. package a;
G. package A;
```