1. Which statements about the final modifier are correct? (Choose all that apply.) A. Instance and static variables can be marked final. B. A variable is effectively final only if it is marked final. C. An object that is marked final cannot be modified. D. Local variables cannot be declared with type var and the final modifier. E. A primitive that is marked final cannot be modified.	A, E Las variables de instancia y estáticas pueden ser marcadas como ***final***, haciendo que la opción A sea correcta. Efectivamente final significa que una variable local no está marcada como ***final*** pero cuyo valor no cambia después de ser establecido, haciendo que la opción B sea incorrecta. La opción C es incorrecta, ya que final se refiere solo a la referencia a un objeto, no a su contenido. La opción D es incorrecta, ya que ***var*** y ***final*** se pueden usar juntos. Finalmente, la opción E es correcta: una vez que un primitivo está marcado como final, no se puede modificar.

2. Which of the following can fill in the blank in this code to make it compile? (Choose all that apply.)
 	
public class Ant {
_____ void method() {}
}
 	
A. default
 	
B. final
 	
C. private
 	
D. Public
 	
E. String
 	
F. zzz:
	B, C
	
	La palabra clave ***void*** es un tipo de retorno. Solo el modificador de acceso o los especificadores opcionales están permitidos antes del tipo de retorno. La opción C es correcta, creando un método con acceso privado. La opción B también es correcta, creando un método con acceso de paquete y el especificador opcional ***final***. Dado que el acceso de paquete no usa un modificador, podemos saltar directamente a final. La opción A es incorrecta porque el acceso de paquete omite el modificador de acceso en lugar de especificar ***default***. La opción D es incorrecta porque Java distingue entre mayúsculas y minúsculas. Hubiera sido correcto si ***public*** hubiera sido la elección. La opción E es incorrecta porque el método ya tiene un tipo de retorno ***void***. La opción F es incorrecta porque no se permiten etiquetas para los métodos.

3. Which of the following methods compile? (Choose all that apply.) A. final static void rain() {} B. public final int void snow() {} C. private void int hail() {} D. static final void sleet() {} E. void final ice() {} F. void public slush() {}	A, D Las opciones A y D son correctas porque los especificadores opcionales están permitidos en cualquier orden. Las opciones B y C son incorrectas porque cada una tiene dos tipos de retorno. Las opciones E y F son incorrectas porque el tipo de retorno está antes del especificador opcional y el modificador de acceso, respectivamente.
4. Which of the following can fill in the blank and allow the code to compile? (Choose all that apply.) final _____ song = 6; A. int B. Integer C. long D. Long E. double F. Double	A, B, C, E El valor 6 puede ser promovido implícitamente a cualquiera de los tipos primitivos, haciendo que las opciones A, C y E sean correctas. También puede ser autoboxeado a ***Integer***, haciendo que la opción B sea correcta. No puede ser promovido y autoboxeado a la vez, haciendo que las opciones D y F sean incorrectas.

5. Which of the following methods compile? (Choose all that apply.) A. public void january() { return; } B. public int february() { return null;} C. public void march() {} D. public int april() { return 9;} E. public int may() { return 9.0;} F. public int june() { return;}	A, C, D Las opciones A y C son correctas porque un método ***void*** opcionalmente puede tener una sentencia return siempre que no intente devolver un valor. La opción B no compila porque ***null*** requiere un objeto de referencia como tipo de retorno. Dado que ***int*** es primitivo, no es un objeto de referencia. La opción D es correcta porque devuelve un valor ***int***. La opción E no compila porque intenta devolver un ***double*** cuando el tipo de retorno es ***int***. Dado que un ***double*** no se puede asignar a un ***int***, tampoco se puede devolver como tal. La opción F no compila porque en realidad no se devuelve ningún valor.
6. Which of the following methods compile? (Choose all that apply.) A. public void violin(int… nums) {} B. public void viola(String values, int… nums) {} C. public void cello(int… nums, String values) {} D. public void bass(String… values, int… nums) {} E. public void flute(String[] values, …int nums) {} F. public void oboe(String[] values, int[] nums) {}	A, B, F Las opciones A y B son correctas porque el único parámetro varargs es el último parámetro declarado. La opción F es correcta porque no usa ningún parámetro varargs. La opción C es incorrecta porque el parámetro varargs no es el último. La opción D es incorrecta porque no se permiten dos parámetros varargs en el mismo método. La opción E es incorrecta porque los ... para un varargs debe estar después del tipo, no antes.

7. Given the following method, which of the method calls return 2? (Choose all that apply.)
 	
`public int juggle(boolean b, boolean... b2) {`
`  return b2.length;`
`}`
 	
A. `juggle();`
B. `juggle(true);`
C. `juggle(true, true);`
D. `juggle(true, true, true);`
E. `juggle(true, {true, true});`
F. `juggle(true, new boolean[2]);`
	D, F
	
	La opción D pasa el parámetro inicial más dos más para convertirlo en una matriz varargs de tamaño 2. La opción F pasa el parámetro inicial más una matriz de tamaño 2. La opción A no compila porque no pasa el parámetro inicial. La opción E no compila porque no declara una matriz correctamente. Debería ser ***new boolean[] {true, true}***. La opción B crea una matriz varargs de tamaño 0, y la opción C crea una matriz varargs de tamaño 1.

8. Which of the following statements is correct? A. Package access is more lenient than protected access. B. A public class that has private fields and package methods is not visible to classes outside the package. C. You can use access modifiers so only some of the classes in a package see a particular package class. D. You can use access modifiers to allow access to all methods and not any instance variables. E. You can use access modifiers to restrict access to all classes that begin with the word Test.	D La opción D es correcta. Una práctica común es establecer todos los campos como privados y todos los métodos como públicos. La opción A es incorrecta porque el acceso protegido permite todo lo que permite el acceso al paquete y, además, permite el acceso a las subclases. La opción B es incorrecta porque la clase es pública. Esto significa que otras clases pueden ver la clase. Sin embargo, no pueden llamar a ninguno de los métodos ni leer ninguno de los campos. Esencialmente es una clase inútil. La opción C es incorrecta porque el acceso al paquete se aplica a todo el paquete. La opción E es incorrecta porque Java no tiene esa capacidad de acceso comodín.

9. Given the following class definitions, which lines in the main() method generate a compiler error? (Choose all that apply.)

```
// Classroom.java
package my.school;
public class Classroom {
  private int roomNumber;
  protected static String teacherName;
  static int globalKey = 54321;
  public static int floor = 3;
  Classroom(int r, String t) {
    roomNumber = r;
    teacherName = t; } }
```

```
// School.java
1: package my.city;
2: import my.school.*;
3: public class School {
4:   public static void main(String[] args) {
5:     System.out.println(Classroom.globalKey);
6:     Classroom room = new Classroom(101, "Mrs. Anderson");
7:     System.out.println(room.roomNumber);
8:     System.out.println(Classroom.floor);
9:     System.out.println(Classroom.teacherName); } }
```

A. None: the code compiles fine.
B. Line 5
C. Line 6
D. Line 7
E. Line 8
F. Line 9

B, C, D, F

Las dos clases están en paquetes diferentes, lo que significa que el acceso privado y el acceso de paquete no compilarán. Esto causa errores de compilador en las líneas 5, 6 y 7, haciendo que las opciones B, C y D sean respuestas correctas. Además, el acceso protegido no compilará ya que ***School*** no hereda de ***Classroom***. Esto causa el error de compilador en la línea 9, lo que hace que la opción F también sea una respuesta correcta.

10. What is the output of executing the Chimp program?

```
// Rope.java
1: package rope;
2: public class Rope {
3:   public static int LENGTH = 5;
4:   static {
5:     LENGTH = 10;
6:   }
7:   public static void swing() {
8:     System.out.print("swing ");
9:   } }
```

```
// Chimp.java
1: import rope.*;
2: import static rope.Rope.*;
3: public class Chimp {
4:   public static void main(String[] args) {
5:     Rope.swing();
6:     new Rope().swing();
7:     System.out.println(LENGTH);
8:   } }
```

A. swing swing 5
B. swing swing 10
C. Compiler error on line 2 of Chimp
D. Compiler error on line 5 of Chimp
E. Compiler error on line 6 of Chimp
F. Compiler error on line 7 of Chimp

B

Rope ejecuta la línea 3, estableciendo LENGTH a 5, e inmediatamente después de eso ejecuta el inicializador estático, que lo establece a 10. La línea 5 en la clase Chimp llama al método estático normalmente e imprime swing y un espacio. La línea 6 también llama al método estático. Java permite llamar a un método estático a través de una variable de instancia, aunque no se recomienda. La línea 7 usa la importación estática en la línea 2 para hacer referencia a LENGTH. Por estas razones, la opción B es correcta.

11. Which statements are true of the following code? (Choose all that apply.)

```
1: public class Rope {
2:   public static void swing() {
3:     System.out.print("swing");
4:   }
5:   public void climb() {
6:     System.out.println("climb");
7:   }
8:   public static void play() {
9:     swing();
10:    climb();
11:  }
12:  public static void main(String[] args) {
13:    Rope rope = new Rope();
14:    rope.play();
15:    Rope rope2 = null;
16:    System.out.print("-");
17:    rope2.play();
18:  }}
```

A. The code compiles as is.

B. There is exactly one compiler error in the code.

C. There are exactly two compiler errors in the code.

D. If the line(s) with compiler errors are removed, the output is swing-climb.

E. If the line(s) with compiler errors are removed, the output is swing-swing.

F. If the line(s) with compile errors are removed, the code throws a NullPointerException.

B, E

La línea 10 no compila porque no se permite que los métodos estáticos llamen a métodos de instancia. Aunque estemos llamando a ***play()*** como si fuera un método de instancia y exista una instancia, Java sabe que ***play()*** es realmente un método estático y lo trata como tal. Dado que esta es la única línea que no compila, la opción B es correcta. Si se elimina la línea 10, el código imprime swing-swing, lo que hace que la opción E sea correcta. No lanza una NullPointerException en la línea 17 porque play() es un método estático. Java mira el tipo de referencia para ***rope2*** y traduce la llamada a ***Rope.play()***

12. How many variables in the following method are effectively final?

```
10: public void feed() {
11:   int monkey = 0;
12:   if(monkey > 0) {
13:     var giraffe = monkey++;
14:     String name;
15:     name = "geoffrey";
16:   }
17:   String name = "milly";
18:   var food = 10;
19:   while(monkey <= 10) {
20:     food = 0;
21:   }
22:   name = null;
23: }
```

A. 1

B. 2

C. 3

D. 4

E. 5

F. None of the above. The code does not compile.

B

La prueba para efectivamente ***final*** es si el modificador ***final*** se puede agregar a la variable local y el código aún compila. La variable ***monkey*** declarada en la línea 11 no es efectivamente ***final*** porque se modifica en la línea 13. Las variables ***giraffe*** y ***name*** declaradas en las líneas 13 y 14, respectivamente, son efectivamente finales y no se modifican después de que se establecen. La variable ***name*** declarada en la línea 17 no es efectivamente ***final*** ya que se modifica en la línea 22. Finalmente, la variable ***food*** en la línea 18 no es efectivamente ***final*** ya que se modifica en la línea 20.

13. What is the output of the following code?

```
// RopeSwing.java
import rope.*;
import static rope.Rope.*;
public class RopeSwing {
  private static Rope rope1 = new Rope();
  private static Rope rope2 = new Rope();
  {
    System.out.println(rope1.length);
  }
  public static void main(String[] args) {
    rope1.length = 2;
    rope2.length = 8;
    System.out.println(rope1.length);
  }
}
// Rope.java
package rope;
public class Rope {
  public static int length = 0;
}
```

A. 02

B. 08

C. 2

D. 8

E. The code does not compile.

F. An exception is thrown.

14. How many lines in the following code have compiler errors?

```
1: public class RopeSwing {
2:   private static final String leftRope;
3:   private static final String rightRope;
4:   private static final String bench;
5:   private static final String name = "name";
6:   static {
7:     leftRope = "left";
8:     rightRope = "right";
9:   }
10:  static {
11:    name = "name";
12:    rightRope = "right";
13:  }
14:  public static void main(String[] args) {
15:    bench = "bench";
16:  }
17: }
```

A. 0

B. 1

C. 2

D. 3

E. 4

F. 5

D

Hay dos detalles a tener en cuenta en este código. Primero, observe que ***RopeSwing*** tiene un inicializador de instancia y no un inicializador estático. Dado que ***RopeSwing*** nunca se construye, el inicializador de instancia no se ejecuta. El otro detalle es que **length** es estático. Los cambios de cualquier objeto actualizan esta variable estática común.

E

Si una variable es ***static final***, debe establecerse exactamente una vez, y debe ser en la línea de declaración o en un bloque de inicialización estático. La línea 4 no compila porque ***bench*** no se establece en ninguna de estas ubicaciones. La línea 15 no compila porque no se permite que las variables ***final*** se establezcan después de ese punto. La línea 11 no compila porque ***name*** se establece dos veces: una vez en la declaración y otra vez en el bloque estático. La línea 12 no compila porque rightRope también se establece dos veces. Ambos están en bloques de inicialización estática.

15. Which of the following can replace line 2 to make this code compile? (Choose all that apply.)

```
1: import java.util.*;
2: // INSERT CODE HERE
3: public class Imports {
4:   public void method(ArrayList<String> list) {
5:     sort(list);
6:   }
7: }
```

A. import static java.util.Collections;

B. import static java.util.Collections.*;

C. import static java.util.Collections.sort(ArrayList<String>);

D. static import java.util.Collections;

E. static import java.util.Collections.*;

F. static import java.util.Collections.sort(ArrayList<String>);

16. What is the result of the following statements?

```
1:  public class Test {
2:    public void print(byte x) {
3:      System.out.print("byte-");
4:    }
5:    public void print(int x) {
6:      System.out.print("int-");
7:    }
8:    public void print(float x) {
9:      System.out.print("float-");
10:   }
11:   public void print(Object x) {
12:     System.out.print("Object-");
13:   }
14:   public static void main(String[] args) {
15:     Test t = new Test();
16:     short s = 123;
17:     t.print(s);
18:     t.print(true);
19:     t.print(6.789);
20:   }
21: }
```

A. byte-float-Object-

B. int-float-Object-

C. byte-Object-float-

D. int-Object-float-

E. int-Object-Object-

F. byte-Object-Object-

B

Las dos formas válidas de hacer esto son ***import static java.util.Collections.*;*** e ***import static java.util.Collections.sort;***. La opción A es incorrecta porque solo se puede hacer una importación estática en miembros estáticos. Las clases como **Collections** requieren una importación regular. La opción C no tiene sentido, ya que los parámetros del método no tienen cabida en una importación. Las opciones D, E y F intentan engañarlo para que invierta la sintaxis de ***import static***.

E

El argumento en la línea 17 es un ***short***. Se puede promover a un int, por lo que se invoca ***print()*** en la línea 5. El argumento en la línea 18 es un booleano. Se puede convertir automáticamente a un Booleano, por lo que se invoca ***print()*** en la línea 11. El argumento en la línea 19 es un ***double***. Se puede convertir automáticamente a un ***Double***, por lo que se invoca ***print()*** en la línea 11. Por lo tanto, la salida es int-Object-Object-

17. What is the result of the following program?

```
1:  public class Squares {
2:    public static long square(int x) {
3:      var y = x * (long) x;
4:      x = -1;
5:      return y;
6:    }
7:    public static void main(String[] args) {
8:      var value = 9;
9:      var result = square(value);
10:      System.out.println(value);
11:   } }
```

A. -1

B. 9

C. 81

D. Compiler error on line 9

E. Compiler error on a different line

B

Dado que Java es de paso por valor y la variable en la línea 8 nunca se reasigna, permanece como 9. En el método ***square***, ***x*** comienza como 9. El valor de y se convierte en 81, y luego x se establece en -1. La línea 9 sí establece el resultado en 81. Sin embargo, estamos imprimiendo el valor, y eso sigue siendo 9, lo que hace que la opción B sea correcta.

18. Which of the following are output by the following code? (Choose all that apply.)

```
public class StringBuilders {
  public static StringBuilder work(StringBuilder a,
    StringBuilder b) {
    a = new StringBuilder("a");
    b.append("b");
    return a;
  }
  public static void main(String[] args) {
    var s1 = new StringBuilder("s1");
    var s2 = new StringBuilder("s2");
    var s3 = work(s1, s2);
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
  }
}
```

A. s1 = a

B. s1 = s1

C. s2 = s2

D. s2 = s2b

E. s3 = a

F. The code does not compile.

B, D, E

Dado que Java es de paso por valor, asignar un nuevo objeto a a no cambia la persona que llama. Llamar a ***append()*** sí afecta a la persona que llama porque tanto el parámetro del método como la persona que llama tienen una referencia al mismo objeto. Finalmente, devolver un valor sí pasa la referencia a la persona que llama para la asignación a s3. Por estas razones, las opciones B, D y E son correctas.

19. Which of the following will compile when independently inserted in the following code? (Choose all that apply.)

```
1: public class Order3 {
2:   final String value1 = "red";
3:   static String value2 = "blue";
4:   String value3 = "yellow";
5:   {
6:     // CODE SNIPPET 1
7:   }
8:   static {
9:     // CODE SNIPPET 2
10:  } }
```

A. Insert at line 6: value1 = "green";

B. Insert at line 6: value2 = "purple";

C. Insert at line 6: value3 = "orange";

D. Insert at line 9: value1 = "magenta";

E. Insert at line 9: value2 = "cyan";

F. Insert at line 9: value3 = "turquoise";

B, C, E

La variable ***value1*** es una variable de instancia ***final***. Se puede establecer solo una vez: en la declaración de la variable, un inicializador de instancia o un constructor. La opción A no compila porque la variable final ya se estableció en la declaración. La variable ***value2*** es una variable estática. Tanto los inicializadores de instancia como los estáticos pueden acceder a variables estáticas, lo que hace que las opciones B y E sean correctas. La variable ***value3*** es una variable de instancia. Las opciones D y F no compilan porque un inicializador estático no tiene acceso a las variables de instancia.

20. Which of the following are true about the following code? (Choose all that apply.)

```
public class Run {
  static void execute() {
    System.out.print("1-");
  }
  static void execute(int num) {
    System.out.print("2-");
  }
  static void execute(Integer num) {
    System.out.print("3-");
  }
  static void execute(Object num) {
    System.out.print("4-");
  }
  static void execute(int... nums) {
    System.out.print("5-");
  }
  public static void main(String[] args) {
    Run.execute(100);
    Run.execute(100L);
  }
}
```

A. The code prints out 2-4-.

B. The code prints out 3-4-.

C. The code prints out 4-2-.

D. The code prints out 4-4-.

E. The code prints 3-4- if you remove the method static void execute(int num).

F. The code prints 4-4- if you remove the method static void execute(int num).

A, E

El parámetro 100 es un ***int*** y por lo tanto llama al método ***int*** coincidente, lo que hace que la opción A sea correcta. Cuando se elimina este método, Java busca el siguiente constructor más específico. Java prefiere el **autoboxing** a **varargs**, por lo que elige el constructor ***Integer***. El parámetro 100L es un ***long***. Dado que no se puede convertir a un tipo más pequeño, se convierte automáticamente en un ***Long*** y luego se llama al método para ***Object***, lo que hace que la opción E sea correcta.

21. Which method signatures are valid overloads of the following method signature? (Choose all that apply.) public void moo(int m, int… n) A. public void moo(int a, int… b) B. public int moo(char ch) C. public void moooo(int… z) D. private void moo(int… x) E. public void moooo(int y) F. public void moo(int… c, int d) G. public void moo(int… i, int j…)	B, D La opción A es incorrecta porque tiene la misma lista de parámetros de tipos y, por lo tanto, la misma firma que el método original. Las opciones B y D son sobrecargas de métodos válidas porque los tipos de parámetros en la lista cambian. Al sobrecargar métodos, el tipo de retorno y los modificadores de acceso no necesitan ser los mismos. Las opciones C y E son incorrectas porque el nombre del método es diferente. Las opciones F y G no compilan. Puede haber como máximo un parámetro varargs, y debe ser el último elemento de la lista de parámetros.