**Constructores**

**Ejercicio 1:**

SALIDA:

*Pruebas 1*

*123*

*123*

¿Por qué? Porque los constructores por defecto (sin parámetros) llaman al super implicitamente, por ende, se llama al constructor del padre.

**Ejercicio 2:**

SALIDA:

*Nada*

¿Por qué? Porque hay un error de código. No se puede declarar dos veces la misma variable. Además, hay un constructor privado…

Los constructores por defecto se llaman siempre implícitamente. Si llamo a un constructor de una clase hijo con parámetros, este llamará al constructor de la clase padre por defecto (si es que lo tiene).

Si pongo un constructor explicito, se anula el por defecto.

SIEMPRE se ejecutan los constructores de los padres. El fin de un constructor es inicializar los atributos de una clase. No tendría sentido instanciar un objeto y que los atributos de su clase padre no estén inicializados.

**Ejercicio 3:**

SALIDA:

123

¿Por qué? Porque los constructores se llaman “en cascada”, partiendo desde la superclass más alta.

**Cuestionario**

1. En el constructor ¿En qué casos necesario invocar explícitamente al constructor de la clase base?  
     
   Cuando la clase base no posee un constructor por defecto y debamos enviar un parámetro a un constructor parametrizado.
2. ¿En toda clase es necesario definir un constructor?  
     
   No, no es necesario. Las clases definen constructores por defecto (vacíos, sin parámetros), los cuales inicializan los atributos de la misma con el valor neutro para el tipo de atributo.
3. ¿Una clase abstracta debe tener constructor?  
     
   Depende del caso. Puede tener un constructor en una clase abstracta si quiero inicializar algún atributo de la misma, de esa forma cambiando el valor que aparecerá en las clases hijas. Si bien la clase abstracta no puede instanciar objetos, esto no me limita que tenga un constructor.
4. ¿Una interfaz puede tener constructor?  
     
   No, las interfaces no pueden ser instanciadas. Las interfaces se usan a través de clases que las implementen, a través de los objetos de las mismas.
5. En una jerarquía de clases ¿En que orden se ejecutan los constructores?  
     
   Se ejecutan desde la super clase (la clase base) padre de mayor nivel, derivando en la clase hijo. Si mi jerarquía es Vehiculo -> Auto -> Rueda, e instancio una rueda, se ejecuta primero el constructor de Vehiculo, luego el de Auto, y luego el de Rueda. No tendría sentido que los atributos de las clases padres no sean inicializados.
6. ¿Qué ocurre si olvidamos poner super como primera línea de la subclase?  
     
   El super siempre se llama implícitamente, a menos que la superclase no posea un constructor por defecto (debido a que hay uno parametrizado escrito explícitamente). En este caso, se debería llamar a super y enviarle los atributos necesarios del constructor parametrizado.
7. Los constructores no se heredan entre jerarquías de clases. Verdadero o Falso. Justifique

Verdadero. Los constructores no se heredan entre jerarquías de clases. La jerarquía de clases hereda todos los miembros de una clase (atributos, métodos, otras clases heredadas). Los constructores no son miembros de la clase. Pero pueden invocarse utilizando super().  
  
Además, no tiene mucho sentido heredar un constructor de una clase padre. ¿Para qué iba a querer el constructor del padre si soy una clase hijo, que necesita tener un constructor propio?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Modificadores de los atributos | | | |
| Visibilidad | private | package | protected | public |
| Misma clase | Sí | Sí | Sí | Sí |
| Mismo paquete | No | Sí | Sí | Sí |
| Subclase en mismo paquete | No | Sí | Sí | Sí |
| Subclase en otro paquete | No | No | Sí | Sí |
| Cualquier clase que no sea subclase en otro paquete | No | No | No | Sí |

**Modificadores de Visibilidad**

**Colecciones**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | ArrayList | LinkedList | TreeSet | HashMap |
| Agregar | Sí | Sí | Sí | Sí |
| Eliminar | Sí | Sí | Sí | Sí |
| Recorrido | Se puede recorrer el ArrayList con un foreach, o con un Iterator. | Se puede recorrer una LinkedList con un foreach, o con un Iterator. | Se puede recorrer con un foreach o un Iterator. | No se puede iterar directamente, pero pueden usarse otras formas como entrySet(), Iterator, iterando sobre las Keys, o un forEach usando expresiones lambda. |
| Orden de los elementos | Orden de inserción. | Orden de inserción. | Orden en orden ascendente mantenido por su orden natural (haya o no un comparator). | Al azar, no es constante, no se garantiza que el orden de los elementos se mantenga a medida que pasa el tiempo. |
| Escenario de uso ideal |  |  |  |  |
| Admite duplicados | Sí | Sí | No | No |

**Excepciones**

Realice los siguientes pasos, completando con el código dentro de cada ítem:

1. Dentro de un método, lance una excepción del tipo ArithmeticException

2. Cambie la excepción por una IOException

3. ¿Qué diferencia encontró? ¿Por qué?  
  
ArithmeticException puede ser lanzada solo con escribir throw new ArithmeticException. Para lanzar una IOException, además de necesitar importar la biblioteca que la contenga, debe ser añadida en la declaración del método con la palabra clave **throws** seguida por **IOException**.  
  
Esto es debido a que existen dos tipos de excepciones: checked exceptions y unchecked exceptions.   
  
Las excepciones chequeadas son aquellas que necesitan ser manejadas de alguna forma. Son errores que se producen debido a algún problema externo no provocado por la inhabilidad del programador. Por ejemplo, que un archivo no sea encontrado y, por ende, no pueda ser leído. Este tipo de error necesita ser manejado de alguna forma, o en su defecto, debe ser aclarado que el método puede tirar una excepción de ese tipo. Esto se hace usando la keyword ***throws Exception***.  
  
Las excepciones no-chequeadas son aquellas que no se chequean al momento de compilar el programa. Son más que nada causadas por la falta de habilidad del programador. Es decir, por errores en el código causados por VOS. Intentar dividir por cero, intentar calcular el radio de un cuadrado, poner un string en un int de alguna forma media extraña… y demás.

Las excepciones no chequeadas no necesitan ser explícitamente indicadas, pero sin embargo esto no quiere decir que no puedan ser catch-eadas. Se pueden atrapar y manejar de la misma forma que las chequeadas.

4. Cree una excepción propia, heredando de RuntimeException.  
  
**public** **class** SuperException **extends** RuntimeException {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**public** SuperException(String msg) {

**super**("Super excepcion: " + msg);

}

}

5. Muestre cómo utilizarla, afectando de la menor manera posible al contexto en que se llama al método que la lanza  
  
**public** **class** Prueba {

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

Prueba p = **new** Prueba();

p.excepcion();

System.***out***.println(p);

}

**public** **void** excepcion() {

**try** {

**throw** **new** SuperException("XD");

} **catch** (Exception e) {

System.***out***.println("Excepcion: " + e.getMessage());

}

}

}  
  
6. Muestre cómo tratar una excepción registrando un mensaje de error  
  
Lo mismo que arriba.

7. Muestre cómo tratar una excepción relanzándola con un cambio de clase  
  
¿Esto? No entendí.  
  
**public** **class** Prueba {

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

Prueba p = **new** Prueba();

p.excepcion();

System.***out***.println(p);

}

**public** **void** excepcion() {

**try** {

**throw** **new** SuperException("XD");

} **catch** (SuperException e) {

System.***out***.println("Excepcion: " + e.getMessage());

}

}

}

8. ¿Para qué sirve el bloque finally?  
  
***The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. But finally is useful for more than just exception handling — it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break. Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.***Al ejecutarse siempre, incluso aunque haya ocurrido una excepción, es una buena forma de poder controlar las cosas. Por ejemplo, si intento escribir en un archivo con un FileWriter, puede ocurrir como no ocurrir una excepción. El bloque finally sería un buen lugar para asegurarse de que el archivo es cerrado, sin importar si ocurrió o no una excepción.

9. ¿Qué parte tiene más prioridad en un bloque try-catch-finally?  
  
¿Prioridad en qué sentido?   
  
Si es por orden de ejecución, los que tienen mayor prioridad son el try y el finally. ¿Por qué? Porque ambos se ejecutarán sí o sí, sin importar si ocurre o no una excepción. El catch solamente se va a ejecutar en caso de que ocurra una.

10. Si hubiera un return dentro del bloque catch, y existiera un finally… ¿se ejecutan ambos? ¿en qué orden?  
  
Se ejecutan ambos, primero se ejecuta el bloque catch, y la instrucción de retorno se pone en “pausa”, hasta la finalización de la ejecución del bloque finally. El bloque finally **siempre es ejecutado**, sin importar el código que haya en el try, o en el catch.

**Static**

Investigue, explique y ejemplifique los siguientes conceptos:

1. Importación static:   
     
   Me permite escribir los métodos de una clase estática sin necesidad de estar escribir el nombre de la clase. E.g.: en vez de usar Math.pow, puedo importar estáticamente Math y utilizar los métodos de la misma usando sus nombres directamente.
2. Atributos static:

Un atributo estático es común a todas las instancias (objetos) de una clase. Es un atributo al nivel de la clase. Una sola copia del atributo es creada y compartida por todas las instancias.

Un ejemplo: puede servir como contador de objetos de esa misma clase.

1. Métodos static:  
     
   Un método estático puede ser invocado sin necesidad de instanciar una clase.
2. Explique claramente qué significa declarar una clase como static y resuelva el ejemplo siguiente analizando en profundidad la respuesta escogida y explicando el porqué de su elección.   
     
   Java permite clases estáticas solamente dentro de una clase. Es decir, permite solamente la creación de clases estáticas nested (anidadas, la estática dentro de otra clase) (you can declare a static class as an inner class inside an outer class).   
     
   Clases top-level pueden simular ser estáticas si las declaramos como final y todos sus miembros son estáticos (métodos y atributos), y su constructor privado.  
     
   Las clases estáticas se usan cuando no tiene sentido instanciar la clase para utilizar sus métodos. Un buen ejemplo es la clase Math, que contiene constantes matématicas y resolución de algunos cálculos. Tener que instanciar un objeto de esa clase para usar sus métodos sería confuso.  
     
   La clases estáticas top-level no pueden ser instanciadas. Pero las static nested sí pueden serlo.   
     
   La respuesta del enunciado es el punto 2. (ver: <https://www.geeksforgeeks.org/nested-classes-java/> )

**Final**Pueden declararse atributos, métodos, y clases final.

Los atributos final son constantes que no pueden modificarse.

Los métodos final no pueden ser overriden (sobreescritos) por las clases heredadas.

Las clases final no pueden ser heredadas por otras. **Clases abstractas**

1. Cree una clase, y agregue un método abstracto (a través de la palabra clave abstract). ¿Qué error ocurrió? ¿Por qué?  
  
Eclipse muestra un error. Si una clase tiene métodos abstractos, la misma por definición debe ser abstracta. La documentación de Java es bastante clara en este punto.

2. Cree una clase como abstracta sin incluir ningún método abstracto, y verifique que no puede crear una instancia de esa clase.

Cannot instantiate the type Prueba.

3. Cree una clase como abstracta incluyendo sólamente un método abstracto, y verifique que tampoco puede crearse una instancia de esa clase.

Cannot instantiate the type Prueba.

4. Cree una nueva clase y haga que extienda la clase abstracta creada en (2). Instánciela. ¿Qué puede decir de lo que ha ocurrido?  
  
Se ha creado un objeto que hereda todos los miembros de la clase abstracta padre. Si bien las clases abstractas no pueden instanciarse, sí pueden ser instanciadas las clases heredadas de la misma (mientras estas tampoco sean abstractas).

5. Agregue un par de métodos abstractos a la clase abstracta utilizada en (4) y observe que errores ocurrieron. ¿Cuales son las dos maneras de solucionar estos errores (sin eliminar los métodos creados)?  
  
Primera forma: realizando un override de los métodos heredados. De esta forma, el método heredado puede ser sobrescrito con una funcionalidad propia, que será única para la clase en la que está sobrescrito.   
  
Segunda forma: convertir a la clase actual en una clase abstracta. De esta forma no hace falta que los métodos sean declarados en la misma (pero eventualmente tendrán que ser declarados en clases hijas subsiguientes).

## **Interfaces**

1. Todos los métodos en una interfaz son automáticamente públicos. ¿Es esto verdad? ¿Cuál es la utilidad?  
     
   Sí, es verdad. Las interfaces son un *contrato* que se realizan entre la misma y una clase. A través de este contacto, la clase se asegura de implementar todos aquellos métodos de la interfaz.   
     
   Las interfaces están hechas para ayudar a definir el comportamiento de una clase. Si por defecto las definiciones fueran en privado, las clases que implementen la interfaz no podrían declarar los métodos. No sería de ayuda para definir el comportamiento de la clase.
2. Una interfaz no debería tener atributos, pero puede tener constantes. ¿Por qué sería necesario?  
     
   En el caso de que las constantes sea necesarias para algunos de los métodos de la interfaz. Es una buena forma de evitar que las cosas estén desparramadas, y a falta de otro, ¿qué mejor lugar que poner una constante relevante para los métodos de una interfaz, que la interfaz misma?
3. En Java 1.8 se incorporó el concepto de Métodos por Defecto, dentro de las interfaces. ¿Por qué se hizo esto? Proporcione un ejemplo de su autoría.  
     
   Los métodos por defecto son una forma que tiene Java de declarar métodos en las interfaces sin que los mismos tengan que declararse en las clases que implementan la interfaz.   
     
   Si a mí me dan el código de una interfaz que es implementada por 500 clases diferentes, y tengo que declarar un nuevo método abstracto… significaría que ahora debo implementar obligatoriamente el mismo método en esas 500 clases. En sí, es una medida que garantiza la retrocompatibilidad.   
     
   Se pueden usar estos métodos si los mismos otorgan un comportamiento esencial para cada una de las clases que implementan la interfaz, de otra forma no se recomienda utilizarlos.

**Clases abstractas e interfaces**

1. ¿Cómo se puede extender de más de una clase en Java?  
  
La múltiple herencia puede ser simulada a través del uso de una herencia común y de la implementación de interfaces. Las interfaces ayudan a definir el comportamiento de una clase, y además extiende el polimorfismo presente en Java, ya que los objetos instanciados de esa clase además de ser del tipo Clase, también son del tipo de cada una de las interfaces que implemente.

*In Java, a class can inherit from only one class but it can implement more than one interface. Therefore, objects can have multiple types: the type of their own class and the types of all the interfaces that they implement. This means that if a variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface.*

2. ¿Cuántas interfaces se pueden implementar en una clase en Java?  
  
A fines prácticos, no hay un límite de interfaces que puedan ser implementadas.   
  
El límite en sí está impuesto por cosas como la cantidad de memoria disponible en la computadora, el máximo valor de ID que puede tomar una interfaz en Java, o el espacio en disco duro para guardar el código.

3. ¿Cuántas clases se pueden extender desde una clase en Java?  
  
Las clases pueden ser heredadas ilimitadas veces a fines prácticos. No hay límite en la cantidad de clases hijos, nietos, etc, que pueda tener una clase.  
  
Pero, una clase solo puede realizar un extend de otra. Java no permite la herencia múltiple, es decir, extenderse de dos clases diferentes.

4. ¿Qué diferencia hay entre implementación y extensión?  
  
La implementación hace referencia únicamente a las interfaces. Se implementa una interfaz cuando se quiere añadir comportamiento extra a una clase.   
  
La extensión hace referencia al hecho de heredar todos los miembros de una clase padre. Es decir, sus atributos y métodos.

**Herencia**

1. La herencia es una relación entre clases… ¿de qué tipo? ¿en qué sentido/s se puede leer? Proporcione un ejemplo de su autoría.  
  
La herencia representa una relación del tipo “ES UN”, también conocida como una relación padre-hijo.  
  
Por ejemplo, si Vehículo es una clase padre, y Auto una clase hijo heredada de Vehículo, se lee: un Auto ES UN Vehículo.

2. ¿Por qué es necesaria la herencia? Algunos dicen que para reutilizar código. Sin embargo, esto no es así.  
  
No se trata de reutilizar código, si no, de poder usar el código ya existente. Es decir, si yo tengo código que existe y funciona para la clase padre, cuando herede una clase hijo, ese mismo código va a funcionar para la misma también.

Además, la herencia extiende el polimorfismo dentro del código. Un Auto puede ser considerado un Vehículo sin recurrir a mayores problemas, lo cual es una gran herramienta que puede llevar a simplificar el código.

También la herencia sirve para encapsular funcionalidades compartidas entre nuestra clase y su superclase.

3. En Java todas las clases heredan implícitamente, salvo que se explicite. ¿De quien heredan? ¿Por qué? ¿Qué permite esto?  
  
Por defecto, todas las clases de Java son heredadas de la clase Object. ¿Por qué? Para que todas las clases compartan algunas funcionalidades esenciales en común, como la sincronización (wait, notify, notifyAll), recolección de basura (finalize), soporte de colecciones (hashCode, equals), clonación de objetos (clone), y métodos como getClass() y toString().  
  
Además, que la clase Object sea la super-clase de todas las clases, nos permite poder utilizarla para pasar objetos de forma genérica sin tener que preocuparnos de qué tipo es el objeto que estamos recibiendo.

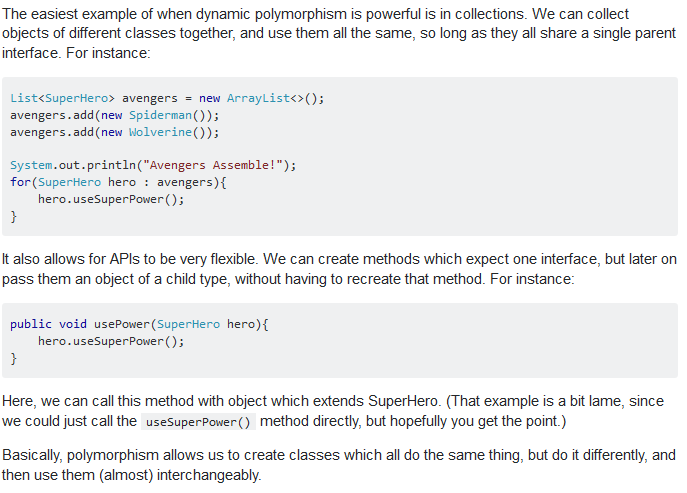
4. Diferencie claramente y brinde ejemplos de uso de sobrecarga y sobreescritura de métodos.  
  
La sobrecarga de métodos es la capacidad que tienen algunos lenguajes de programación de poder contar con métodos que se llamen de la misma forma, pero que sin embargo pueden ser diferenciados debido a una diferencia entre los parámetros que reciben. El compilador elige a qué método llamar basándose en qué parámetros se le están siendo enviados.   
  
La sobreescritura de métodos es algo totalmente distinto. La sobreescritura de un método cambia la funcionalidad de un método heradado en la clase hijo. No se sobrecarga el método, directamente es pisado y un nuevo código es puesto en su lugar.

**Polimorfismo**

1. ¿Qué se entiende por polimorfismo?  
  
***It’s the ability to refer to, pass, use objects, which all share some common characteristics, but when calling methods or acting on the specific (or “concrete”) implementation of that object you get the behavior of that specific specialized/more derived version of the class****.*  
  
Es la capacidad de un objeto de ser interpretado de varias formas. De poder tener un objeto que sea de un tipo, pero que no se “encuentre únicamente atado a ese tipo”. El polimorfismo está presente incluso en las cosas más básicas, como sumar un entero y un float.   
  
Cualquier objeto de una clase hijo puede tomar la forma de cualquiera de sus clases padre superiores.

Existen dos tipos de polimorfismo, polimorfismo dinámico y polimorfismo estático.

El **polimorfismo dinámico** está asociado con la sobreescritura de métodos y la capacidad de poder referenciar a un objeto de una clase hijo como si fuera un objeto de la clase padre. Esto está implícitamente dicho en las herencias de Java (un Auto ES UN Vehículo).



El **polimorfismo estático** hace referencia a la sobrecarga de métodos. Tener el mismo nombre de método, pero poder llamarlo de diferentes formas según los parámetros.

2. ¿Qué sucedería en la POO si no existiera el concepto de Polimorfismo?  
  
Se caería a pedazos. El paradigma orientado a objetos se basa fundamentalmente en el polimorfismo. Si no existiera el concepto de polimorfismo, la herencia no tendría tanto sentido y las interfaces no servirían de mucho. Es decir, sus funciones se mantendrían, pero se perdería la flexibilidad que nos permite escribir código de manera más eficiente.

3. Proporcione un ejemplo en el que asigne polimórficamente una referencia. Cambie la asignación, y muestre en qué afecta al código preexistente, si correspondiese.  
  
**public** **static** **void** main(String[] args) {

PruebaPadre prueba = **new** PruebaHijo();

prueba.saySomething();

}

@Override

**public** **void** saySomething() {

System.***out***.println("Somethng");   
}

Originalmente, se llama al método saySomething de la clase hijo, porque el objeto creado a través de la referencia es de esa clase.   
  
Si se cambiase la referencia por un new PruebaPadre(), el método llamado sería el saySomething implementado de la clase padre.

4. Defina parámetro formal y parámetro real.  
  
Un parámetro formal es aquel que es escrito en la firma/declaración de un método. Es el tipo de parámetro que se espera recibir.

El parámetro real es el parámetro que se pasa al momento de invocar a la función. Son los valores de estos parámetros los que se copian en los parámetros formales al momento de realizar la llamada.

5. Proporcione un ejemplo en el que se reciba un parámetro de un supertipo, y cómo se invocaría dicho método desde una clase cliente del mismo. ¿Cambiará el método original si el parámetro real fuera de una u otra subclase del parámetro formal?  
  
El método que es llamado es aquel perteneciente a la clase que instanció el objeto del parámetro real. Si bien la referencia lo toma como un objeto de la clase super, el método que se invoca es el que está implementado en la subclase.

6. Proporcione un ejemplo en el que se declare una lista (List) de un tipo polimórfico. ¿Qué ventajas encuentra contra declararla de un tipo específico?

Ver imagen puesta en el punto 1.