Preparando OCJP 1.6

Pablo Reyes Almagro

20 de enero de 2012

Resumen

Todo el contenido aquí publicado es un mero resumen y adaptación del libro SUN™ CERTIFIED PROGRAMMER FOR JAVA™ 6 STUDY GUIDE, Kathy Sierra & Bert Bates. La intención de este documento no es ser usado como sustituto del libro original, sino servir de complemento en los momentos previos al exámen, gracias a su brevedad con respecto al original y la falta de detalles innecesarios a mi juicio. Para compender muchos aspectos de lo aquí resumido, es imprescindible disponer de un libro completo como el original.

Índice

1. Declaraciones y control de acceso

En este capítulo se explica cómo deben declararse los distintos elementos del lenguaje (clases, interfaces, métodos, variables, etc) y qué modificadores se les puede aplicar a cada uno para limitar su visibilidad o su comportamiento.

1.1. Identificadores legales

Para que un identificador sea reconocido y aceptado por el compilador, debe seguir una serie de reglas.

- Debe estar compuesto por UNICODE, números, $ y _
- Debe comenzar por UNICODE, $ o _
- Sin limite de longitud
- No pueden ser palabras reservadas
- Case-sensitive

1.2. Convenciones de código de Java SUN

SUN creó un standard de codificación para facilitar la lectura de código, con la intención de que todo el mundo usara el mismo estilo de nombrar variables, funciones, clases, etc.

- Clases e interfaces
  - Primera letra mayuscula
  - camelCase
  - Las clases son sustantivos
  - Las interfaces son adjetivos
- Métodos
  - Primera letra minuscula
  - camelCase
  - Verbo+sustantivo
- Variables
  - Primera letra minuscula
  - camelCase
  - Nombres significativos, y a poder ser cortos
- Constantes
  - Todo mayúsculas
  - Palabras separadas por _

1.3. Estándares JavaBeans

Un estándar de nomenclatura que facilita la portabilidad de código entre desarrolles y aplicaciones. Los JavaBeans son un modelo de componentes usados para encapsular varios objetos en uno único para hacer uso de un objeto en lugar de varios más simples. Estas convenciones permiten que las herramientas puedan utilizar, reutilizar, sustituir y conectar JavaBeans.

Reglas para atributos

- getter y setters se conforman por *get* o *set* más el nombre del atributo. No tiene porqué existir un atributo con el mismo nombre exactamente
- Si el atributo es un booleano, puede usarse `isAtributo()`
- Los getters deben ser `public` y no recibir argumentos
- Los setters deben ser `public`, recibir un argumento del tipo del atributo y devolver *void*

Reglas para listeners

- Los métodos que registren un listener son `add`
- Los métodos que borren un listener comienzan por `remove`
- Después del prefijo sigue el nombre del listener, que siempre acaba en `Listener` (`removeLocationListener()`)
- El tipo de listener que se va a registrar o borrar se pasa como argumento

1.4. Archivos de código fuente

Existen una serie de reglas acerca de la estructura de los ficheros:

- Sólo una clase `public` por fichero, pero varias no publicas
- Los comentarios pueden aparecer en cualquier sitio
- Si hay una clase `public` en el fichero, el nombre del fichero debe ser el mismo que el de la clase
- Si la clase está en un `package`, el nombre de éste debe aparecer en la la primera linea del fichero, antes de los `import`
- Si hay `import`, deben aparecer entre el nombre del `package` (si lo hay) y la declaración de la clase
- El `package` y los `import` se aplican a todos las clases del fichero

1.5. Declaración de clases y modificadores de acceso

1.5.1. Modificadores de acceso a clases

El acceso significa visibilidad. Dependiendo de los modificadores de nuestros elementos, estos serán visibles o no por los demás.

- Default (`package`)
  - No lleva modificador, por lo que se aplica uno por defecto
  - Clase visible para todos las clases dentro del mismo paquete
- Public
  - Clase visible para todas las demás, incluso las de otros paquetes
  - Si la clase se usa desde otro paquete, es necesario importarla

1.5.2. Modificadores de no acceso

Estos modificadores no están relacionados con la visibilidad, y pueden ser combinados con los modificadores de acceso.

Cuadro 1: Exam tip 1

En el examen, algunas preguntas de lógica compleja son simples trampas con problemas de modificadores de acceso. Si se detecta alguna irregularidad, elegir siempre "Compilation Fails".

- Final
  - Una clase `final` no puede ser extendida, es decir, no puede tener subclases
- Abstract
  - No puede ser instanciada, su objetivo es ser extendida
  - Si algún método es `abstract`, la clase debe ser `abstract`
  - No todos los métodos deben ser `abstract`
  - Los métodos `abstract` no llevan cuerpo, no tienen {}
  - La primera subclase no abstracta de esta clase debe implementar todos los métodos `abstract` de esta
  - Una clase `MedioDeTransporte` podría ser abstracta, y de ella heredarían la clase `Coche`, `Tren`, o `Avion`.

Cuadro 2: Exam tip 2

Cuando en el examen aparezca un método acabado en `;`, es decir, sin cuerpo, y no esté en una interface, comprobar que dicho método es `abstract` y que, por tanto, la clase también lo es.
Una clase no puede ser `final` y `abstract`, pues tienen objetivos totalmente opuestos. Si vemos algo así en el examen, el código no compilará.

1.6. Declaración de interfaces

Una interface define una serie de coasas que una clase que la implemente deberá hacer, pero no cómo deberá hacerlo.

- Todos sus métodos son implicitamente `public abstract`
- Todos los atributos son implicitamente constantes (`public static final`)
- Los métodos no pueden ser `final`, `strictfp` ni `native`
- Una `interface` puede ser heredada por otras `interface`
- Una `interface` puede heredar de más de una `interface`, al contrario que las clases
- Una `interface` no puede implementar otra `interface`
- Una `interface` puede ser `public` o usar el acceso por defecto

Cuadro 3: Exam tip 3

En el examen podemos encontrar casos en los que una clase intente cambiar el valor de una variable de la interfaz que implementa. Debemos recordar que los atributos de una interfaz siempre son constantes y no pueden ser cambiados!

1.7. Declaración de métodos y atributos

Éstos pueden tener modificadores de acceso y de no acceso. Puesto que tanto métodos como atributos se declaran de la misma forma, se explicarán de forma conjunta.

1.7.1. Modificadores de acceso

Existen dos conceptos que diferenciar aquí: referencia y herencia. Llamamos acceso por referencia cuando una clase accede a un miembro de otra clase mediante el operador punto (`.`):

```
class Coche{
  int potencia = 140;
}
```

```
class Test{
  Coche c = new Coche();
  int x = c.potencia;
}
```

El acceso por herencia se da cuando una subclase accede a los miembros heredados de su superclase. Estos miembros pertenecen, desde que se hace la herencia, al hijo. Es decir, cuando modificamos un miembro heredado no estamos modificando el miembro en la superclase, sino en la propia clase en la que se modifica:

```
class Coche{
  int potencia = 140;
}
```

```
class CocheBmw extends Coche{
  int cv = potencia;
}
```

- Public
  - Cualquier clase puede referenciarlo y heredarlo, sea o no sea subclase y esté o no esté en el mismo paquete
- Private
  - Miembro sólo accesible por la propia clase, ni siquiera por sus subclases
  - Si una subclase define un miembro con el mismo nombre que algún miembro privado de su superclase, lo que hace realmente es crear uno nuevo e independiente, no acceder al de la superclase
- Default (package)
  - Puede ser referenciado y heredado desde cualquier clase dentro del mismo paquete
- Protected
  - Igual que el acceso default, pero además puede ser heredado desde clases fuera del package

1.7.2. Modificadores de no acceso

- Final
  - En un método, éste no puede ser redefinido por una subclase
  - En un atributo, no se puede cambiar su valor
- Abstract

- En un método, significa que se declara pero no se implementa (acaba con `;` en vez de `{}`)
- Es una forma de forzar a las subclases a que implementen dicho método de forma particular.
- No puede ser combinado con el modificador `static`

- Synchronized
  - En un método, significa que sólo puede ser accedido por un `thread` a la vez
  - Puede ser combinado con cualquiera de los modificadores de control de acceso

- Native
  - Es un método que será implementado dependiendo de la plataforma, normalmente en C
  - No se implementa. No tiene cuerpo, igual que los métodos `abstract`

- Strictfp
  - Igual que para las clases, fuerza a usar la norma IEEE754 para los numeros en coma flotante

1.8. Métodos con lista de argumentos variables (var-args)

Desde JAVA 5.0, se permite usar listas con una cantidad de argumentos variables. Se definen como `float avg(float... numeros)`. Puede haber más parámetros en el método, pero sólo puede haber uno *var-args* y debe ir al final. Cuando se llama a un método sobrecargado o sobreescrito, los métodos que no incluyen *var-args* tienen preferencia sobre el que lo incluye:

```
class Test{
  void showAvg(int a, int b){
    System.out.println("Sólo dos elementos. Media no fiable");
  }
  void showAvg(int... numeros){
    /* Calculo real de la media */
  }
  public static void main(String[] args){
    showAvg(1, 2); // Mostrara que la media no es fiable!
  }
}
```

1.9. Declaración de constructores

Cada vez que se instancia un objecto con `new`, se invoca un constructor para dicho objeto.

- Toda clase tiene al menos un constructor. Sino lo creamos nosotros, el compilador hará uno por defecto
- Si se especifica al menos un constructor, el constructor implícito deja de existir
- Puede haber más de un constructor con distinto tipo o número de argumentos
- Un constructor nunca devuelve nada
- El nombre del constructor debe ser el mismo que el de la clase
- Puede usar modificadores de acceso {`public`, `private`, `protected`, `package`}
- No puede ser `static` (ya que se ejecutan sobre la instancia), ni `final`, ni `abstract` (ya que no pueden ser sobrescritos)

1.10. Declaración de variables

Existen dos tipos de variables en Java:

- Primitivas
  - char, boolean, byte, short, int, long y float.
  - Una vez declarado no se puede cambiar su tipo, aunque generalmente se puede cambiar su valor.
- Referencias
  - Se usan para acceder a un objeto.
  - Se declara para ser de un tipo específico, y dicho tipo no puede cambiarse.
  - Pueden referenciar a cualquier objeto del tipo declarado o cualquier tipo compatible (subtipo) suyo.

1.10.1. Declaración de primitivas y sus rangos

Se pueden declarar como:

- Variables de clases (`static`)
- Variables de instancia

- Parámetros
- Variables locales

Tipo	Bits	Bytes	Mínimo	Máximo
`byte`	8	1	-2^7	2^7-1
`short`	16	2	-2^{15}	2-1
`int`	32	4	-2^{31}	$2^{31}-1$
`long`	64	8	-2^{63}	$2^{63}-1$
`float`	32	4	n/a	n/a
`double`	64	8	n/a	n/a
`char`	16	2	0	$2^{16}-1$
`boolean`	Depende de la maquina virtual			

El tipo char, que en otros lenguajes consta de 1 byte, en Java consta de 2, pues alberga un *Unicode* en vez de un *ISO Latin-1*, pudiendo representar muchos más simbolos.

1.10.2. Declaración de variables de referencia

Se pueden declarar como:

- Variables de clases (`static`)
- Variables de instancia
- Parámetros
- Variables locales

Variables de instancia Estas variables se definen dentro de la clase, fuera de cualquier método, y sólo se inicializan cuando se instancia la variable. Son los atributos de la clase, y suelen ser atributos privados. Pueden usar cualquier modificador de acceso: `public`, `private`, `protected` o `package`. Pueden ser `final`, `transient` o `volatile`. No pueden ser `abstract`, `synchronized`, `strictfp`, `native`, ni `static` (ya que entonces serían variables de clase y no de instancia)

Variables locales Son variables declaradas dentro de un método, y pueden tener el mismo nombre que una variable de instancia (*shadowing*). Para acceder a una variable de instancia con el mismo nombre que una variable local, se hace a través de la referencia al objeto `this`. El ciclo de vida de estas variables empieza y acaba en el método en el que se declara. Puede ser `final`. Siempre se almacenan en el *stack*, y no se inicializan por defecto.

Arrays Un array es un objeto que contiene variables del mismo tipo, o subclases del mismo tipo. Pueden almacenar primitivas o referencias a objetos. Como objeto que es, siempre se almacena en el *heap*. Los arrays pueden tener varias dimensiones. También pueden declararse con los corchetes después del nombre de la variable, pero no se recomienda (`Universe[][][] univ;` es mejor que `Universe univ[][][];`)

Cuadro 4: Exam tip 4

Aunque en otros lenguajes se pueda, en Java nunca se puede incluir el tamaño del array en su declaración:

```
int[5] numeros; // ilegal
```

El único momento en el que Java reserva memoria es en la instanciación del objeto:

```
int[] numeros = new int[5]; // legal
```

Variables final Cuando declaramos una variable como final, lo que hacemos es prohibir que su valor cambie una vez ha sido inicializada con un valor explícito (dado por nosotros, no de forma por defecto). En las primitivas, significa que una vez que le demos un valor, ese valor se mantendrá por siempre. Por ejemplo, si declaramos un `final int x = 10;`, ese valor será 10 para siempre y no lo podremos cambiar. En el caso de variables de referencia, significa que no podremos asignarle otro objeto distinto a dicha referencia una vez que le hemos asignado el primero, pero eso no significa que no podamos cambiar el contenido de dicho objeto referenciado. ¡Es la variable de referencia la que no puede cambiar! Es decir, no existen objetos `final`, sino referencias `final`.

Variables transient Una variable transient es una variable que será ignorada a la hora de serializar el objeto que la contiene. Sólo se aplica a las variables de instancia.

Variables volatile Una variable volatile es una variable preparada para ser accedida por varios *threads*. Su acceso será sincronizado, y los *threads* leerán/escribirán directamente en memoria en vez de en versiones locales de la variable. No es un modificador muy común, ya que se prefiere el uso de sincronización. No es necesario para el exámen. Sólo se aplica a las variables de instancia.

Variables static y metodos El modificador static se usa para crear variables y métodos independientes de la instancia. Estas variables y métodos existirán incluso antes de instanciar la clase, y sólo habrá una copia suya independientemente del número de objetos de la clase que haya. Es decir,

todos los objetos de una misma clase comparten el mismo valor para las variables y métodos `static`.

Pueden ser static:

- Métodos
- Variables
- Una clase dentro de otra clase (pero no dentro de un método)
- Bloques de inicialización

No pueden ser static:

- Constructores (no tiene sentido, puesto que operan sobre instancias)
- Clases (a no ser que esten dentro de otra clase)
- Interfaces
- Variables locales
- Métodos y variables de isntancia de clases dentro de clases

1.11. Declaración de enums

Java permire restringir variables para que sólo puedan tomar valores de una lista predefinida:

```
enum Color {ROJO, VERDE, AZUL};
Color colorCoche = Color.VERDE;
```

Los enumeradores pueden ser declarados en una clase aparte o miembros de una clase, pero nunca dentro de un método. Cada enumerador conoce su índice, por lo que el orden si importa.

Cada elemento dentro de `Color` es una instancia de tipo `Color`. Podemos pensar en un `enum` como en un tipo de clase de este tipo (aunque no exactamente):

```
class Color{
  public static final Color ROJO = new COLOR("ROJO", 0);
  public static final Color VERDE = new COLOR("VERDE", 0);
  public static final Color AZUL = new COLOR("AZUL", 0);
}
```

1.11.1. Constructores, métodos y variables en un enum

Los `enum` son un tipo de clase, y como tal, puedes hacer más que mostrar cada una de sus constantes. Imaginemos que queremos saber el código RGB de cada uno de los colores de nuestro `enum`. La mejor manera es tratar cada uno de los valores como objetos que pueden tener sus propios atributos:

```
enum Color{
  ROJO(0xF00),VERDE(0x0F0), AZUL(0X00F);
  private int rgbColor;   // variable de instancia
  Color(int rgb){       // constructor
    this.rgbColor = rgb;
  }
  public int getRGB(){
    return this.rgbColor;
  }
}
```

Debemos recordar que:

- No se puede invocar un constructor de `enum` directamente. Este constructor se llama automáticamente con los argumentos que se definen después de las constantes
- Se puede definir más de un argumento para el constructor, y éste se puede sobrecargar al igual que en las clases
- No se puede declarar nada antes de las constantes dentro de un `enum`

2. Orientación a objetos

La orientación a objetos es un paradigma de programación en el que las entidades se definen como elementos que realizan sus propias acciones y que se relacionan con los demás. Este paradigma conlleva el uso de *herencia*, *polimorfismo*, *abstracción* y *encapsulamiento*.

2.1. Encapsulamiento

La orientación a objetos ofrece dos ventajas claras, que son sus principales objetivos: flexibilidad y mantenimiento. Para conseguir dichas cualidades no basta con usar un lenguaje orientado a objetos, sino usarlo debídamente. La clave es encapsular el estado y la lógica de la clase para que no pueda ser accedida de forma directa, y dejar definidas una serie de funciones y métodos encargados de operar sobre ella.

- Mantener atributos de la instancia protegidos (`private/protected`)
- Hacer métodos public para acceder a los atributos de instancia
- Usar la convención de JavaBean para los anteriores

Cuadro 5: Exam tip 5

En el exámen se pueden encontrar preguntas aparentemente relacionadas con el comportamiento de un método, pero donde el problema es falta de encapsulación

2.2. Herencia

La herencia nos permite realizar estructuras de clases más complejas y mas simples a la vez. Por un lado, ofrece la capacidad de visualizar nuestro sistema como un árbol, facilitando el esquema mental que debemos hacer de la relación de las clases. Por otro lado, nos ahorra el escribir métodos repetidos que son comunes a más de una clase.

Java no sopota la multi-herencia, es decir, una clase no puede heredar de más de una clase. En *C++*, por ejemplo, si se permite que una clase extienda de dos clases. Para solventar dicha "deficiencia" en Java, se usan las interfaces.

Sus dos características más importantes son la reutilización de código y el polimorfismo. En el siguiente ejemplo, `Trabajador` hereda de `Persona`, lo que significa que todo trabajador es una persona. Usando la herencia, definimos esta relación y nos ahorramos el tener que escribir un nuevo `setNombre(String)` para el trabajador, pues todas las personas tienen un nombre y un trabajador ya es una persona.

```
class Persona{
  protected String nombre;
  public void setNombre(String n){
    nombre = n;
  }
  public String getNombre(){
    return nombre;
  }
}
class Trabajador extends Persona{
  private float salario;
  public void setSalario(float s){
    salario = s;
  }
  public float getSalario(){
    return salario;
  }
}
public TestClass{
  public static void main(String[] args){
    Trabajador trab = new Trabajador();
    trab.setNombre("Manolo");
    trab.setSalario(2400);
  }
}
```

El operador `instanceof` nos devolverá `true` si el objeto es de la clase que comparamos. Ojo: Toda clase también es una instancia del tipo de su superclase!

```
System.out.println("Manolo instancia de Trabajador?: " + (trab
    instanceof Trabajador)); // true
System.out.println("Manolo instancia de Persona?: " + (trab instanceof
    Persona));    // true
System.out.println("Manolo instancia de Object?: " + (trab instanceof
    Object));   // SIEMPRE true
```

En el exámen se pueden encontrar referencias a *IS-A* y a *HAS-A*:

- *X IS-A Y* significa que *X* deriva de *Y* a través de `extends` o `implements`
- *IS-A* es igual que *derived*, *subclass*, *inherits* o *subtype*
- La clase de la que deriva otra clase se llama *superclase*
- *X HAS-A Y* significa que la clase *X* contiene una referencia a una instancia de la clase *Y*

2.3. Polimorfismo

Cualquier objeto que cumpla la condición *IS-A* más de una vez (todos derivan de `Object`) puede considerarse polimórfico. La única manera de acceder a un objeto es mediante una variable de referencia, de la que debemos recordar:

- Una variable de referencia puede ser de un sólo tipo, y una vez declarada no puede cambiarse dicho tipo

- Puede ser reasignada a otro objeto del mismo tipo (sino es `final`)
- El tipo de esta variable determina los métodos que se pueden invocar en el objeto que está referenciando
- Puede referenciar cualquier objeto del tipo de la variable, o cualquiera de sus subtipos
- Puede tener como tipo el de una interface, y puede referenciar a cualquier clase que implemente dicha interface

2.3.1. Sobreescritura

Imaginemos el siguiente ejemplo, en el que la clase Coche ha sobrescrito un método de la clase de la que hereda:

```
public class TestClass{
  public static void main(String[] args){
    Coche coche = new Coche();
    Vehiculo veh = coche;
    veh.abrirMaletero();   //Error de compilación
    veh.describe();   // Soy un coche
  }
}

class Vehiculo{
  public void describe(){
    System.out.println("Soy un vehiculo");
  }
}

class Coche extends Vehiculo{
  public void describe(){
    System.out.println("Soy un coche");
  }
  public void abrirMaletero(){
    System.out.println("Abriendo maletero");
  }
}
```

A la hora de compilar, el compilador nos dirá que no existe el método `abrirMaletero()` para la clase `Vehiculo` y no compilará. Es decir, el compilador ha buscado el método en la clase de la variable que le referencia: él cree que lo está llamando un `Vehiculo`. Con la llamada a `describe()` hará lo mismo, y al encontrarla, no dará error ninguno.

Pero... y en tiempo de ejecución? La *JVM* si sabrá de qué tipo es realmente el objeto `coche`, por lo que al llamar a `describe()` lo hará desde la clase `Coche` y no desde la clase `Vehiculo`, por lo que mostrará por pantalla "*Soy un coche*".

Es decir, en la compilación se comprueba que exista dicho método para el tipo de objeto del que se está llamando (en el caso anterior, `Vehiculo`), pero es en tiempo de ejecución cuando se decide a qué método llamar (en el caso anterior, `Coche`, ya que `veh` hacía referencia a un `Coche`). ¿Y qué pasaría si la clase `Vehiculo` declara una excepción verificada en `describe()` pero

`Coche` no? Pues que el compilador seguirá creyendo que estamos lanzando `Vehiculo.describe()` y nos saltará con un `unreported exception`, a pesar de ser realmente un `Coche` que no lanza excepciones.

```
public class TestClass{
  public static void main(String[] args){
    Coche coche = new Coche();
    Vehiculo veh = coche;
    veh.describe();     // Unreported Exception
  }
}
class Vehiculo{
  public void describe() throws Exception{
    System.out.println("Soy un vehiculo");
    throw new Exception();
  }
}
class Coche extends Vehiculo{
  public void describe(){
    System.out.println("Soy un coche");
  }
  public void abrirMaletero(){
    System.out.println("Abriendo maletero");
  }
}
```

Para sobrescribir un método se deben seguir una serie de reglas:

- La lista de argumentos debe ser exactamente la misma. De lo contrario, se está creando un método distinto (sobrecarga)
- El tipo devuelto debe ser del mismo tipo o un subtipo del que devolvía la clase ancestra
- El nivel de acceso no puede ser más restrictivo (aunque sí menos) que el que tenía el método que está sobrescribiendo
- Sólo se pueden sobrescribir los métodos heredados (restringido por modificadores de acceso)
- El método puede lanzar excepciones sin verificación sin importar las excepciones declaradas en la superclase
- El método no puede lanzar excepciones verificadas que son nuevas o amplían las declaradas por el método sobrescrito
- El método puede lanzar menos excepciones que las declaradas por el método sobrescrito
- No se puede sobrescribir un método `final` ni `static`

Se puede llamar al método de la superclase en vez de al método propio con `super.metodo()`:

```
class Animal{
  public void describe(){
    System.out.println("Soy un animal");
  }
}
class Dog extends Animal{
  public void describe(){
    super.describe();
    System.out.println("Pero tambien soy un Dog!");
  }
}
```

Cuando se ejecute el método `describe()` de `Dog`, también se mostrará *"Soy un animal"*.

2.3.2. Sobrecarga

La sobrecarga de métodos permite reusar el mismo nombre de método en una clase pero con distintos argumentos (y puede con distinto tipo devuelto). Esto generalmente facilita al usuario la llamada a funciones más o menos genéricas que deberían actuar sobre más de un tipo de dato. Las reglas para la sobrecarga de métodos son sencillas:

- Deben tener argumentos distintos
- Pueden devolver un tipo de dato distinto
- Pueden cambiar el modificador de acceso
- Pueden declarar nuevas o más amplias excepciones verificadas
- Pueden ser sobrecargados en la misma clase o en una subclase

Un simple ejemplo de sobrecarga:

```
public class Test{
  public static void main(String[] args){
    System.out.println(getMax(1, 6));
    System.out.println(getMax(0.9, 0.04));
  }
  static int getMax(int a, int b){
    return (a>b)?a:b;
  }
  static double getMax(double a, double b){
    return (a>b)?a:b;
  }
}
```

Cabe destacar que el método que se va a usar se decide en tiempo de compilación en base a los argumentos usados. Cuando declaramos clases propias y conllevan herencia, eso puede llevar a confusiones:

```
public class Acelerador{
  public void acelerar(Vehiculo v){
    System.out.println("Acelerando vehiculo");
```

```
  }
  public void acelerar(Coche c){
    System.out.println("Acelerando coche");
  }
  public static void main(String[] args){
    Coche coche = new Coche();
    Vehiculo veh = coche;
    Acelerador ac = new Acelerador();
    ac.acelerar(veh); //Acelerando vehiculo!
  }
}
class Vehiculo{  }
class Coche extends Vehiculo{  }
```

A primera vista podría parecer que se mostrará "*Acelerando coche*" puesto que, en realidad, `veh` es un `Coche`. Sin embargo, como hemos dicho antes, el método a usar se decide en tiempo de compilación y por tanto se usa el tipo de la variable que referencia: `Vehiculo`.

A continuación se muestra una tabla con las diferencias entre sobrecarga y sobrescritura:

	Sobrecarga	Sobrescritura
Argumentos	Deben cambiar	No pueden cambiar
Tipo devuelto	Puede cambiar	No puede cambiar (salvo por subclases)
Excepciones	Pueden cambiar	Pueden reducirse o eliminarse
Modificadores de acceso	Pueden cambiar	No pueden ser mas restrictivos
Invocación	El tipo de referencia determina qué versión se llama, en tiempo de compilación	El tipo de la instancia determina que versión se llama, en tiempo de ejecución

2.4. Casteo de variables de referencia

En ocasiones es necesario usar un método de una subclase específica, pero la referencia que tenemos es su superclase. En estos casos se usa un tipo de casteo llamado *downcast*. El problema de este *downcast* es que no genera errores en tiempo de compilación, pero puede hacerlo en tiempo de ejecución.

```
class Animal{
  void makeNoise(){
    System.out.println("Ruido generico");
  }
}
class Dog extends Animal{
  void makeNoise(){
```

Cuadro 6: Exam tip 6

Aunque los métodos de instancia se resuelven en tiempo de ejecución, las variables de instancia se resuelven en tiempo de compilación.

```
class Mammal{
  String name = "furry ";
  String makeNoise(){ return "generic noise"); }
}
class Zebra extends Mammal{
  String name = "stripes ";
  String makeNoise(){ return "bray"; }
}
public class ZooKeeper{
  public static void main(String[] args){
    new ZooKeper().go();
  }
  void go(){
    Mammal m = new Zebra();
    System.out.println(m.name + m.makeNoise());
  }
}
```

El anterior código mostrará *furry bray*, ya que la variable de instancia se ha resuelto en tiempo de compilación para la variable de referencia de tipo `Mammal` mientras que el método se ha resuelto en tiempo de ejecución para la instancia de tipo `Zebra`.

```
    System.out.println("Guau!");
  }
  void playDead(){
    System.out.println("Boca arriba!");
  }
}
class Test{
  public static void main(String[] args){
    Animal[] a = {new Animal(), new Dog(), new Animal()};
    for(Animal animal : a){
      if(animal instanceof Dog){
        animal.playDead();
      }
    }
  }
}
```

Puede parecer, en un principio, que el código funcionaría correctamente, puesto que nunca se llamará a `tumbarse` desde un animal genérico gracias a la comprobación de `if(animal instanceof Dog)`, pero debemos recordar que en tiempo de compilación se comprueba que existe un método `tumbarse` para la variable de referencia, que en este caso es un `Animal`, por lo que obtendríamos un error de compilación: *cannot find symbol*.

Para solventar este problema, usamos el antes llamado *downcast*:

```
for(Animal animal : a){
```

```
  if(animal instanceof Dog){
    Dog d = (Dog) animal;
    d.playDead();
  }
}
```

Así conseguimos que la variable de referencia sea el propio `Dog` y el compilador no de problemas. Estamos haciendo que el compilador confie en nosotros, diciendole: "*Tranquilo, se que lo que viene ahora es un Dog, trátalo como si fuera uno*". Debemos tener cuidado con esto, ya que podemos hacer creer al compilador que un animal se trata de un Dog cuando realmente no lo es, lo que llevará a un error de ejecución más adelante:

```
Animal animal = new Animal();
Dog d = (Dog) animal;
```

El código compilará correctamente, pero cuando lo ejecutemos encontraremos un error *java.lang.ClassCastException*. El compilador sólo comprueba que los dos tipos pertenecen al mismo árbol de referencia, por lo que no podemos confiar en él para que detecte errores como este.

Tambien existe lo que se llama *upcasting*, es decir, castear una subclase a su superclase. Esto ni siquiera es necesario indicarlo, ya que se hace de forma implícita. Cuando se hace un *upcast*, sólo se está restringiendo el abanico de métodos que se pueden invocar.

```
Dog d = new New Dog();
Animal a1 = p;       // Upcasting implicito
Animal a2 = (Animal) p; // Upcasting explicito
```

Cuadro 7: Exam tip 6

En el exámen encontraremos código escrito de forma poco común tanto por la falta de espacio como por las ganas de ofuscar. El siguiente código:

```
Animal a = new Dog();
Dog d = (Dog) a;
d.makeNoise();
```

puede ser reemplazado por esto:

```
Animal a = new Dog();
((Dog)a).makeNoise();
```

2.5. Implementación de interfaces

Cuando se implementa una interfaz, lo que se hace es aceptar el contrato provisto por esta, e implementarlo en la clase. Para el compilador, lo único que requiere es que se le dé una implementación legal al método, pudiendo ser esta un cuerpo vacío. Para que la implementación de una interfaz sea legal, la

primer clase no abstracta del árbol de herencia debe cumplir con las siguientes reglas:

- Especificar una implementación concreta para todos los métodos definidos en el contrato de la interfaz
- Seguir todas las reglas para la sobre escritura de métodos
- Declarar sin excepciones controles sobre los métodos de ejecución distintos de los declarados por el método de interfaz, o subclases de los declarados por el método de interfaz
- Mantener la misma firma del método, y devolver un valor del mismo tipo (o subtipo)

Además, una clase que implemente una `interface` puede ser `abstract`. ¿¿!!Cómo!!?? Pues sí, una clase puede `implements` una interface, y además ser abstracta, dejándo a su primera subclase no abstracta que implemente los métodos:

```
abstract class Ball implements Bounceable{}

class BeachBall extends Ball{
  public void bounce(){ ... }
  public void setBounceFactor(int bf) { ... }
}
```

Como vemos, existe una `interface Bounceable` que tiene dos métodos, `bounce()` y `setBounceFactor()`. Estos métodos han sido implementados por la primera subclase no abstracta de la clase que hizo el `implements`.

Recordemos que una clase puede implementar más de una `interface`, y que una `interface` puede extender otras `interface`:

```
interface Moveable{
  void moveIt();
}

interface Spherical(){
  void doSphericalThing();
}

interface Bounceable extends Moveable, Spherical{
  void Bounce();
  void setBounceFactor(int bf);
}

abstract class Ball implements Bounceable{
  public void Bounce(){ ... }
  public void setBounceFactor(int bf){ ... }
}

class SoccerBall extends Ball{
  public void moveIt(){ ... }
  public void doSphericalThing(){ ... }
}
```

En este ejemplo, tenemos una `interface Bounceable` que extiende otras dos interfaces, `Moveable` y `Spherical`. Luego, una clase abstracta `Ball` ha decidido implementar los métodos `Bounce()` y `setBounceFactor(int bf)`, dejando los otros dos métodos a una subclase `SoccerBall`. Si `SoccerBall` implementara también `Bounce()` y `setBounceFactor()` estaría sobrescribiendo los métodos que ya ha implementado su superclase.

Cuadro 8: Exam tip 7

Muchas preguntas en el exámen están relacionadas con el mal uso de la implementación de interfaces. Estos ejemplos deben quedar claros!

```
class Foo{}             // OK
class Bar implements Foo {}     // No! No se puede implementar una
    clase
interface Baz{}           // OK
interface Fi{}            // OK
interface Fee implements Baz{}    // No! Una interface no puede
    implementar nada
interface Zoo extends Foo{}     // No! Una interface no puede extender
    una clase
interface Boo extends Fi{}      // Ok. Una interface puede extender
    otra interface
class Toon extends Foo, Button{}    // No! Una clase no puede extender
    varias clases
class Zoom implements Fi, Baz{}   // Ok. Una clase puede implementar
    varias interfaces
interface Vroom extends Fi, Baz{}   // Ok. Una interface puede extender
     varias interfaces
class Yow extends Foo implements Fi{}  // Ok. Una clase puede extender
    e implementar (extender primero)
```

2.6. Tipos devueltos válidos

Vamos a ver qué podemos devolver en un método, diferenciando si es un método nuevo, sobreescrito o sobrecargado.

2.6.1. Tipos devueltos en métodos sobrecargados

Cuando se sobrecarga un método, realmente se está creando un método nuevo que usa el mismo nombre que otro ya existente, aunque con distinta lista de argumentos. Por tanto, el valor devuelto del método sobrecargado no tiene restricciones de ningún tipo.

2.6.2. Tipos devueltos en métodos sobrescritos

Desde JAVA 5 en adelante se puede devolver el mismo tipo que el método original, y además cualquier subclase de este.

```
class Alpha{
  Alpha doStuff(char c){
    return new Alpha();
  }
}
class Beta extends Alpha{
  Beta doStuff(char c){
    return new Beta();
  }
}
```

2.6.3. Reglas a la hora de devolver valores

Existen seis reglas que deben tenerse en cuenta a la hora de devolver un valor:

- Aunque se pida en la declaración devolver un objeto, se puede devolver `null`

  ```
  public Button doStuff(){
    return null;
  }
  ```

- Se pueden devolver arrays

  ```
  public String[] foo(){
    return new String[] {"Pablo", "Juan", "Pepe"};
  }
  ```

- Si se pide devolver un tipo primitivo, se puede devolver cualquier valor que se pueda convertir implicitamente a dicho tipo

  ```
  public int foo(){
    char c = 'c';
    return c;
  }
  ```

- Si se pide devolver un tipo primitivo, se puede devolver cualquier valor que se pueda convertir explicitamente a dicho tipo

  ```
  public int foo(){
    float f = 32.5f;
    return (int) f;
  }
  ```

- No se puede devolver nada de un método `void`

- Si se pide devolver un objeto, se puede devolver cualquier objeto que se pueda convertir implicitamente a dicho objeto

  ```
  public Animal getAnimal(){
    return new Horse();
  }
  ```

```
  public Object getObject(){
    int[] nums = {1, 2, 3};
    return nums;
  }

  public interface Chewable {}
  public class Gum implements Chewable {}
  public class TestChewable{
    public Chewable getChewable(){
      return new Gum();
    }
  }
```

2.7. Constructores e instanciación (INCOMPLETO)

Los objetos deben ser contruidos, y son los *constructores* los encargados de ello. Toda clase, aunque sea abstracta, debe tener al menos un constructor. Si el programador no lo crea, el compilador creará uno por defecto.

2.7.1. Conceptos básicos

Un constructor no devuelve nada (ni siquiera `void`) y tiene siempre el mismo nombre que la clase. Su función es, normalmente, inicializar las variables de instancia del objeto a los valores deseados.

```
class Foo{
  int size;
  String name;
  Foo(String name, int size){
    this.name = name;
    this.size = size;
  }
}
```

La clase `Foo` tiene un constructor que recibe dos argumentos, por lo que podría ser instanciada de la siguiente forma:

```
Foo f = new Foo("Pablo", 10);
```

Hemos dicho que si no creamos un constructor, el compilador creará uno por nosotros por defecto. Bien, el constructor que crea el compilador es un constructor sin argumentos, y sólo lo crea si no hemos definido ningun otro constructor. Es decir, el siguiente código no compilará:

```
Foo f = new Foo();
```

Ya que no existe ningún constructor sin argumentos, y al haber definido uno con argumentos, el compilador no ha creado ninguno.

2.7.2. Llamadas en cadena del constructor

Cuando un constructor se llama, éste llama antes que nada al constructor de su superclase con una llamada a `super()`. Esto quiere decir que si tenemos

una clase `Horse` que hereda de `Animal`, y ésta última hereda de `Object`, una llamada al constructor de `Horse` como esta:

```
Horse h = new Horse();
```

implicará a lo siguiente:

1. Se invoca el constructor de `Horse`
2. Se invoca el constructor de `Animal`
3. Se invoca el constructor de `Object`
4. Se asignan los valores correspondientes a las variables de instancia de `Object`
5. Termina el constructor de `Object`
6. Se asignan los valores correspondientes a las variables de instancia de `Animal`
7. Termina el constructor de `Animal`
8. Se asignan los valores correspondientes a las variables de instancia de `Horse`
9. Termina el constructor de `Horse`

2.7.3. Reglas para constructores

Las siguientes reglas resumen todo lo que se debe saber sobre constructores a la hora del exámen.

- Los constructores pueden utilizar cualquier modificador de acceso
- El nombre del constructor debe ser igual al de la clase
- Los constructores no pueden tener un valor de retorno
- Es legal tener un método que tenga el mismo nombre que la clase, pero eso no lo convierte en constructor
- Si no se especifica explícitamente un constructor en la clase, implícitamente se genera un constructor que no recibe parámetros
- Si se especifica al menos un constructor, el constructor implícito no será generado
- Todos los constructores tienen en la primer línea una llamada a un constructor sobrecargado `this()` o una llamada a un constructor de la superclase `super()`
- La llamada a un constructor de la superclase puede ser con o sin argumentos

- Un constructor sin argumentos no es necesariamente un constructor por defecto
- No es posible acceder a una variable o método de instancia hasta que se ejecute el constructor de la superclase
- Solo variables o métodos estáticos pueden ser llamados al invocar un constructor mediante `this()` o `super()`
- Las clases abstractas también tienen constructores, y estos son ejecutados cuando se instancia una subclase
- Las interfaces no tienen constructores, ya que no forman parte del árbol de herencia
- La única manera de invocar un constructor es desde otro constructor

2.8. Variables y métodos estáticos

El modificador de acceso `static` hace que una variable o un método sea propio de la clase en sí y no de la instancia. Es decir, son accesibles sin necesidad de instanciar el objeto que lo tiene, y su valor será común en todas las instancias de la clase. Puede usarse, por ejemplo, para llevar la cuenta de cuantas instancias se han creado de una clase:

```
class Frog{
  static int frogCount = 0;

  public Frog(){
    frogCount += 1;
  }

  public static void main(String[] args){
    new Frog();
    new Frog();
    new Frog();
    System.out.println("Frogs: " + frogCount); //Mostrará Frogs: 3
  }
}
```

Si la variable `frogCount` no hubiera sido `static`, hubieramos obtenido... ¡un error de compilación!

nonstatic variable frogCount cannot be referenced from a static context

La JVM no puede saber a qué `Frog` nos estamos refiriendo cuando llamamos a `frogCount` desde `main()`, ya que es un método estático que no pertenece a ninguna instancia. ¿Qué `frogCount` iba a elegir? ¿De qué instancia? `main()` está corriendo desde un contexto estático no perteneciente a ninguna instancia.

Y bien, ¿cómo podemos acceder entonces a miembros de instancia desde un método estático? Pues teniendo una instancia y accediendo con el operador `.`

Cuadro 9: Exam tip 8

Un error muy común es intentar acceder a miembros no estáticos desde un contexto estático:

```
class Foo{
  int x = 3;
  public static void main(String[] args){
    System.out.println("x is " + x);
  }
}
```

Debemos entender que no podemos acceder a la variable `x` desde el método `main()`, ya que éste pertenece a la clase mientras que la variable `x` es un miembro de instancia, ¡que ni siquiera existe!
En el exámen podemos encontrar errores de este tipo muy enmascarados que pueden despistarnos, como el siguiente ejemplo:

```
class Foo{
  int x = 3;
  float y = 4.3f;
  public static void main(String[] args){
    for(int z=x; z<++x; z--, y=y+z){
      ...
    }
  }
}
```

Si nos preguntan por el comportamiento del anterior código, podemos pasar un buen rato siguiendo a las variables. Sin embargo, la realidad es que nos encontramos con un error de compilación ya que se está intentando acceder a `x e y` desde el método estático `main()`.

```
class Frog{
  int x = 3;

  int getX(){
    return x;
  }

  public static void main(String[] args){
    Frog f = new Frog();
    System.out.println("x is " + f.getX());
  }
}
```

2.8.1. Acceso a variables y métodos estáticos

Para acceder a una variable o método estático, lo hacemos con el operador `.` tambien, pero usando como prefijo el nombre de la clase en vez de la referencia a una instancia:

```
class Frog{
```

```
  static int frogCount = 0;

  public Frog(){
    frogCount += 1;
  }
}

class TestFrog{
  public static void main(String[] args){
    new Frog();
    new Frog();
    new Frog();
    System.out.println("frogCount: " + Frog.frogCount); //Mostrará
        frogCount = 3
  }
}
```

Pero además, JAVA permite usar una referencia a objeto para acceder a un método estático, por lo que los dos siguientes acceso son idénticos:

```
int frogs1 = Frog.frogCount;
Frog f = new Frog();
int frogs2 = f.frogCount;
```

2.9. Coupling and cohesion (INCOMPLETO)

3. Literales, asignaciones y variables

3.1. Literales

Un literal es simplemente la representación del contenido de un tipo de dato primitivo. Por decirlo de otra forma, es lo que escribimos para representar un entero, un numero decimal, un booleano o un caracter.

Enteros Se pueden representar en JAVA de tres formas distintas: decimal (`int x = 198`), octal (`int x = 015`) o hexadecimal (`int x = 0xdead`)

Coma flotante Pueden ser de tipo `float` (32 bits) o `double` (64 bits). Un literal flotante como `163.9658` es por defecto un `double`, por lo que si queremos asignarlo a un `float` tendremos que especificar que sabemos que se puede perder precisión añadiendo al final del literal una *f* (`float g = 163.9658f`)

Booleano Sólo puede tener el valor `true` o `false`, no como en C++ donde un valor distinto de 0 ya implica que el booleano es `true`

Carácter Se representa por un sólo carácter entre comillas simples (`char a = 'a'`). Tambien puede especificarse un código UNICODE (`char letterN = '\u004E'`). Son realmente enteros de 16 bits sin signo, lo que significa que podemos asignarle un valor numérico entre 0 y 65535. También podemos asignar un valor mayor, pero es necesario realizar el casteo (`char c = (char) 70000;`). Para representar un carácter que no se pueda escribir como un literal, usamos la barra invertida (`char newLine = '\n',char tab = '\t'`, etc)

String Aunque no es un tipo primitivo, tambien tiene su literal. Se representa entre comillas dobles (`String nombre = "Jose Miguel"`);

3.2. Asignaciones

Las variables son únicamente contenedores de bits. Sueña extraño, así que lo aclararemos un poco. En el caso de variables que sean asignados a un tipo primitivo, dicha variable alberga una ristra de bits representando su valor de forma numérica. Una variable de tipo `byte` con el valor `6` por ejemplo, significa que la variable contiene el valor `00000110`.

```
byte b = 6;
```

Pero, ¿y cuando la variable es de un tipo no primitivo?

```
Button b = new Button();
```

En ese caso, lo que alberga la variable es una referencia al objeto creado. Lo que almacena dicha variable es una ristra de bits con una dirección para acceder a dicho objeto. Ni siquiera sabemos el formato que tiene, ya que la forma en que

se almacenan las referencias a objetos es dependiente de la máquina virtual. Lo que podemos asegurar es que esa variable no es un objeto, sino un valor que *apunta* a un objeto específico del *heap*. Una variable tambien puede tener asignado el valor `null`:

```
Button b = null;
```

Lo que significa "*La variable de tipo Button b no está referenciando a ningún objeto*"

Ahora que sabemos lo que albergan las variables, veamos cómo podemos cambiarlas.

3.2.1. Asignaciones de primitivas

Se puede asignar un valor primitivo a una variable usando un literal o el resultado de una expresión:

```
int x = 7;
bool b = True;
byte a = 3; // cast automático
byte b = 8; // cast automático
byte c = a + b; // error de compilación
```

Recordemos que un literal entero es siempre de tipo `int`, pero el compilador nos permite asignarlo a una variable tipo `char` o tipo `byte` aún siendo de capacidad menor. El compilador nos añade el casting de forma automática. Tambien sabemos que el resultado de una operación que involucre un `int` (o menor) es siempre un `int`. En la última linea, el resultado de `(a+b)` es suficientemente pequeño como para caber en un contenedor tipo `byte`, pero el resultado de dicha operación es siempre un `int`, por lo que debemos añadir el cast de forma explícita:

```
byte c = (byte) (a + b)
```

RELLENAR, FALTAN COSAS (AUNQUE ELEMENTALES)

3.2.2. Asignaciones de referencias

Podemos hacer que una variable referencie a un objeto recien creado de la siguiente forma:

```
Button b = new Button();
```

Esta linea hace lo siguiente:

1. Crea una referencia de tipo `Button` con nombre `b`
2. Crea un objeto de tipo `Button` en el *heap*
3. Asigna el objeto recien creado a la variable de referencia `b`

Recordemos que se puede asignar una variable de referencia a un objeto que sea subclase del tipo de dicha variable:

```
public class Foo{
  public void doFooStuff();
}
public class Bar{
  public void doBarStuff();
}
class Test{
  public static void main(String[] args){
    Foo reallyABar = new Bar(); // Legal
    Bar reallyAFoo = new Foo(); // Ilegal!
  }
}
```

3.2.3. Ámbito de las variables

Todas las variables no existen eternamente, sería ilógico, ineficiente y problemático. Existen cuatro tipos distintos de variables que se distinguen en el ámbito en el que actuan:

- Variables estáticas, que se crean cuando se carga la clase y son accesibles mientras la clase esté en la JVM
- Variables de instancia, que se crean cuando se instancia la clase y son accesibles mientras la instancia exista
- Variables locales, que son accesibles mientras el método al que pertenencen esté en el *stack*
- Variables de bloque, que son accesibles mientras se ejecuta el bloque de código

```
class Layout{
  static int s = 343; // Variable estática
  int x;          // Variable de instancia
  {int x2 = 5}        // Variable de bloque
  void doStuff(){
    int y = 0;      // Variable local
    for(int z = 0; z<4; z++){ // Variable de bloque
      y += z + z;
    }
  }
}
```

3.2.4. Uso de variables o arrays no inicializados

Java nos permite dejar variables sin inicializar. El comportamiento de dicha variable dependerá de su tipo y de su ámbito.

Variables de instancia

Las variables de instancia de tipo primitivo u objetos son inicializadas a un valor por defecto, que se muestra en la siguiente tabla:

Tipo de variable	Valor por defecto
Referencia a objeto	`null`
`byte, short, int, long`	`0`
`float, double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'`

Debemos tener cuidado a la hora de no inicializar las variables. Por lo general, es conveniente asignarles un valor. Esto hace que sea más facil de leer y menos propenso a errores.

```
public class Book{
  private String title;
  public String getTitle(){
    return title;
  }
  public static void main(String[] args){
    Book b = new Book();
    String s = b.getTitle();
    String t = s.toLowerCase(); // Runtime error!
  }
}
```

En el código anterior, obtenemos un error como este:

Exception in thread "main" java.lang.NullPointerException at Book.main(Book.java:9)

Dicho error se debe a que la variable de instancia `title` se ha inicializado a su valor por defecto `null`, que luego se ha asignado a la variable local `s`, de la que luego se ha querido invocar un método `toLowerCase()`.

En el caso de los arrays, tenemos que tener en cuenta que son objetos y, como tales, toman su valor por defecto `null` si no se especifica lo contrario. Y en el caso de que lo inicialicemos... ¿que pásа con los elementos del array? Que se inicializan a su valor por defecto.

```
public class BirthDays{
  static int[] year = new int[100];
  public static void main(String[] args){
    for(int i=0; i<100; i++){
      System.out.println(year[i]);
    }
  }
}
```

El código anterior mostrará que los cien elementos del array valen `0`.

Variables locales

Las variables locales, incluso las primitivas, siempre, siempre, siempre deben ser inicializadas antes de ser usadas. Java no asigna ningún valor por defecto a las variables locales, por lo que somos nosotros los responsables de

asignarles un valor antes de usarlas. Por supuesto, podemos dejarla declaradas pero sin inicializar siempre y cuando no la usemos (aunque carece de sentido).

```
public class TimeTravel{
  public static void main(String[] args){
    int year;
    int month;
    System.out.println("The year is " + year); // Compiler error!
  }
}
```

En el código anterior, tenemos un error de compilación a la hora de intentar mostrar el año, ya que no tiene ningún valor asignado (ni siquiera `null`). Con el mes no tenemos problemas, ya que no se ha usado.

Cuadro 10: Exam tip 10

El compilador no siempre puede saber si una variable local ha sido inicializada antes de su uso, ya que ello puede depender de ciertas condiciones. En el caso de que el conpilador no esté seguro, nos lo hará saber con un error, por si acaso:

```
public class TestLocal{
  public static void main(String[] args){
    int x;
    if(args[0] != null){
      x = 7;  // El compilador no sabe si esto se ejecutará
    }
    int y = x;  // Compiler error!
  }
}
```

El compilador no mostrará un error:

TestLocal.java:9: variable x might not have been initializated

Las referencias a objetos tambien se comportan de forma distinta dependiendo de si son de instancia o son locales. Al igual que las primitivas, estas variables no se inicializan a nada, ni siquiera a `null`.

```
import java.util.Date;
public class TimeTravel{
  public static void main(String[] args){
    Date date;
    if(date == null){ // Compiler error!
      System.out.println("date is null");
    }
  }
}
```

La variable `date` no se ha inicializado, ni siquiera al valor por defecto `null`, por lo que en el código anterior obtendremos un error de compilación al hacer la comparación `date == null`. Siempre conviene inicializar las variables, especialmente las locales. En este case, habría valido con inicializarla a null (`Date date = null;`).

Pasa lo mismo con los arrays. Una vez construido, sus elementos toman sus valores por defecto de forma automática, pero no el array en sí.

3.2.5. Asignando una variable de referencia a otra

En las variables de tipo primitivo, asignar una variable a otra significa copiar su patrón de bits en la otra variable. Con las variables de referencia a objetos pasa exactamente igual, se copia su ristra de bits, sólo que en este caso esa ristra es una *dirección*.

```
class ReferenceTest{
  public static void main(String[] args){
    Dimension a = new Dimension(5, 10);
    System.out.println(a.height);
    Dimension b = a;
    b.height = 30;
    System.out.println(a.height);
  }
}
```

La salida del código anterior será:

```
10
30
```

Eso es debido a que ambas variables de referencia apuntaban al mismo objeto, que es el que ha sido alterado. Existe una excepción en Java en este sentido, y es con la clase `String`. Un `String` es inmutable, es decir, no pueden cambiar su valor.

```
class StringTest{
  public static void main(String[] args){
    String x = "Java";
    String y = x;   // x e y apuntan al mismo objeto

    System.out.println("y string = " + y);
    x = x + " Bean";
    System.out.println("y string = " + y);
  }
}
```

En este ejemplo, la salida es:

```
y string = Java
y string = Java
```

Lo que ha ocurrido es que, en la linea `x = x + " Bean"` se ha creado un nuevo objeto String `con` contenido "*Java Bean*", y se ha hecho que la variable de referencia `x` apunte a dicho nuevo objeto. Por tanto, a partir de dicha linea, `x` e `y` no apuntan al mismo objeto.

3.3. Pasando variables a métodos

Los métodos pueden recibir primitivas o referencias a objetos en su lista de argumentos. Debemos tener claro cuando estas variables se pueden ver afectadas por operaciones dentro del método.

3.3.1. Pasando referencias a objetos

A la hora de recibir referencias a objetos, lo que queda en el método una vez en ejecución es una copia de la variable de referencia, es decir, otra variable que alberga la misma ristra de bits que la original. Esto significa que los cambios que realicemos en la variable dentro del método no afectarán a la variable, pero los cambios que hagamos sobre el objeto se verán reflejados cuando se salga del método.

```
class ReferenceTest{
  public static void main(String[] args){
    Dimension d = new Dimension(5, 10);
    ReferenceTest rt = new ReferenceTest();
    System.out.println("Before modify() = " + d.height);
    rt.modify(d);
    System.out.println("After modify() = " + d.height);
  }
  void modify(Dimension dim){
    dim.height = dim.height + 1;
    System.out.println("In modify() = " + dim.height);
  }
}
```

La salida del anterior código es:

Before modify() = 10

In modify() = 11

After modify = 11

Vemos como hemos modificado el único objeto de tipo `Dimension` que hemos creado a través de la variable `dim`, y después ese cambio se ha visto reflejado a través de la variable `d`. Alguien puede pensar que Java usa *paso por referencia* cuando ve lo que ha pasado con el objeto anterior, pero nada más lejos de la realidad. Java usa el mismo sistema con las variables de referencia que con las primitivas, y es el *paso por valor*, entiendolo como *paso por copia de variable*. Lo que se copia no es el objeto, sino la variable que dirige a él.

```
void bar(){
  Foo f = new Foo();
  doStuff(f);
}
void doStuff(Foo g){
  g.setName("Boo");
  g = new Foo();
}
```

Reasignar el valor de `g` a un nuevo objeto `Foo` no afecta a la variable `f`, que no ha cambiado su valor y sigue referenciando el objeto `Foo` original.

3.3.2. Pasando primitivas

Como ya se ha dicho, Java hace el *paso por valor* en el sentido de *paso por copia de variable*, por lo que el siguiente resultado no debería asustarnos:

```
class ReferenceTest{
  public static void main(String[] args){
    int a = 1;
    ReferenceTest rt = new ReferenceTest();
    System.out.println("Before modify() = " + a);
    rt.modify(a);
    System.out.println("After modify() = " + a);
  }
  void modify(int number){
    number = number + 1;
    System.out.println("In modify() = " + number);
  }
}
```

Salida:

Before modify() = 1

In modify() = 2

After modify = 1

Recordemos que `number` era una copia de la variable `a`, que albergaba el número `1` como entero. Al cambiar el valor de `number`, lo único que hacemos es cambiar su patrón de bits, por lo que la variable original `a` no se ve afectada.

3.4. Declaración, construcción e inicialización de arrays

Los arrays son objetos que albergan varias variables del mismo tipo. Debe quedar claro que albergan primitivas o referencias a objetos, pero no los objetos en sí. El array en sí se encuentra en el *heap* como objeto que es, pero su referencia puede estar en la pila.

3.4.1. Declaración de arrays

Los arrays se declaran igualmente ya sean para albergar primitivas o referencias a objetos, y existen dos formas:

```
int[] key;
int key[];

Thread[] threads;
Thread threads[];
```

La primera forma es la recomendada, con los corchetes justo después del tipo de la variable.

Los arrays, además, pueden ser de varias dimensiones. Podemos verlos como matrices o incluso como cubos, pero realmente lo que hacemos son arrays de arrays.

```
String[][] names;
String[] otherNames[];
```

La segunda forma debemos evitarla como sea. Es fea y lleva a confusión!

Por otro lado, Java no permite especificar el tamaño del array en la declaración. El tamaño se especificará en la construcción, como indicaremos más adelante. La JVM no reserva espacio hasta que se instancia el array. El siguiente código es ilegal:

```
int[] scores; // Compiler error!
```

3.4.2. Construcción de arrays

Construir un array (o cualquier objeto) significa reservarle espacio en el *heap*, y para reservar dicho espacio tenemos que decirle a la JVM cuando queremos reservar. Esto lo hacemos con el operador `new` aportando el número de elementos entre corchetes:

```
int[] scores;
testScores = new int[4];

Thread[] threads = new Thread[3];
```

Como se puede apreciar, podemos declarar y construir en la misma linea. Recordemos que en ningun caso se llama al constructor de `Thread`. Tras crear el array `threads`, sólo hemos creado un objeto de tipo "array para `Thread`", pero ningún objeto `Thread` ha sido creado!

Cuando construimos un array debemos especificar siempre su tamaño, y no hacerlo nos llevará a un error de compilación:

```
String[] names = new String[];  // Compilation error!
```

A la hora de crear arrays de varias dimensiones, el único tamaño que debemos especificar es el del primer nivel, ya que será el que se instancie y se reservará espacio únicamente para él en el *heap*. Los demás niveles se irán creando a medida que lo vayamos ordenando:

```
int[][] myArray = new int[3][];
myArray[0] = new int[2];
myArray[0][1] = 6;
myArray[0][2] = 7;
myArray[1] = new int[1];
myArray[1][0] = 8;
```

3.4.3. Inicialización de arrays

Con *inicializar* nos referimos a rellenar. No tiene mucho misterio. Existen tres formas de rellenar un array: la trabajosa (uno a uno), la cómoda (en bucle) y la corta (todo en una linea).

```
int[] x = new int[3];
x[0] = 10;
x[1] = 20;
x[2] = 30;

int[] y = new int[3];
for(int i=0; i<y.length; i++){
  y[i] = (i+1)*10;
}

int[] z = {10, 20, 30};
```

Como podemos intuir, el resultado de los tres arrays es el mismo. El objeto de tipo array contiene una única variable pública que nos devuelve el número de elementos que este tiene, por lo que podemos aprovecharlo para recorrerlo y rellenarlo en un bucle. Java nos ofrece, además, una forma corta para declarar, construir e inicializar en una sola linea.

Este último método no siempre podremos usarlo, ya que, por lo general, no sabemos el contenido con el que se rellenará el array de antemano, o siemplemente porque son muchos elementos y es más fácil y rápido que lo haga un bucle. Nótese que no debe especificarse el tamaño del array, ya que el tamaño vendrá dado por el número de elementos. En caso de especificarlo, el código no compilará:

```
int[3] z = {10, 20, 30};  // Compilation error!
```

Para los arrays de varias dimensiones funciona de la misma forma:

```
int[][] z = {{1,2,3}, {4,5,6,7,8}, {9}}
```

3.4.4. ¿Qué puede albergar un array?

Evidentemente, un array puede albergar elementos del tipo del que está declarado, pero existen detalles:

En el caso de un array de primitivas, dicho array puede contener elementos que sean implicitamente convertidos al tipo en el que está definido. Es decir, si declaramos un array de tipo `int`, éste podrá contener cualquier tipo primitivo menor de 32 bits:

```
int[] data = new int[5];
byte b = 4;
data[0] = 125;
data[1] = b;
data[2] = 'a';
```

Esto no implica que podamos hacer que un array de tipo `int`, por ejemplo, pase a referenciar a otro array de tipo `char`:

```
int[] data;
char[] letters = {'a', 'b', 'c'};
data = letter;  // Compilation error!
```

En el caso de las referencias a objetos, el array podrá contener cualquier referencia a objeto que cumpla la condición IS-A con el tipo del array. Así, podrá albergar referencias a subclases u objetos que implementen alguna interfaz:

```
class Car{}
class Subaru extends Car{}
class Ferrari extends Car{}
...
Car[] myCars = {new Subaru(), new Ferrari(), new Car()};
```

En las referencias a objetos, sin embargo, si podemos hacer que un array de tipo `Coche` pase a referenciar a una subclase suya. Podremos hacerlo tambien con interfaces, siempre y cuando se cumpla la condición IS-A:

```
Car[] cars;
Honda[] hondas = {new Honda(), new Honda()};
cars = hondas;
```

3.5. Bloques de inicialización

Ya debería estar claro el orden de ejecución a la hora de instanciar una clase, pero queda algo por ver. Los bloques de inicialización son secciones de código que se ejecutan después de la llamada a los constructores de las superclases, es decir, justo después de la llamada a `super()`. El orden en el que estos bloques aparezcan es importante, pues se ejecutarán *de arriba a abajo*. Pueden ser ejecutados cuando se carga la clase, en caso de ser estáticos, o cuando se instancia el objeto en caso de no serlo.

Tienen su utilidad si la clase tiene muchos constructores que repiten una serie de operaciones comunes. Estos bloques de inicialización se ejecutarán antes de que el constructor llamado realice sus operaciones específicas.

```
class Init{
  Init(int x){
    System.out.println("1-arg constructor");
  }
  Init(){
    System.out.println("no-arg constructor");
  }
  static{
    System.out.println("1st static init");
  }
  { System.out.println("1st instance init"); }
  { System.out.println("2nd instance init"); }
  static{
    System.out.println("2nd static init");
  }

  public static void main(String[] args){
    new Init();
    new Init(7);
  }
}
```

La salida del código anterior es:

1st static Init

2nd static Init

1st instance Init
2nd instance Init
no-arg constructor
1st instance Init
2nd instance Init
1-arg constructor

4. Operadores

Poco hay que saber de los operadores que no sepamos ya, así que resumiremos bastante.

Cuadro 11: Exam tip 11

Aunque en exámenes anteriores era necesario, a partir del exámen de Java 5 ya no es necesario aprender las operaciones a nivel de bits con operadores. Ni *bit shifting*, ni *bitwise*, ni complemento a dos ni divisiones por cero.

4.1. Operadores de asignación

Esto ya ha sido cubierto antes, pero para resumir:

- Cuando asignamos un valor a una primitiva, el tamaño importa. Debemos tener claro cuando ocurrirá el casting impolícito, cuando necesitamos especificarlo, y cuando puede existir truncamiento.
- Una variable de referencia no es un objeto, sino una forma de llegar a dicho objeto.
- Cuando asignamos un valor a una variable de referencia, el tipo importa. Hay que tener claras las reglas para superclases, subclases y arrays.

4.2. Operador de asignación compuestos

Estos operadores nos ahorran algún que otro golpe de tecla. Aunque existen 11 realmente, sólo nos interesan 4 (`+=`, `-=`, `*=`, `/=`). El funcionamiento, ya lo conocemos.

```
y = y - 6;
x = x + 2 * 5;
hello = hello + " mike";  // hello es un String ("hello")
```

Ahora obtenemos el mismo resultado pero con operadores compuestos:

```
y -= 6;
x += 2 * 5;
hello += " mike";
```

4.3. Operadores relacionales

Los valores relaciones nos sirven para comparar dos elementos, y siempre resultan en un boolean (`true` o `false`). De estos operadores relaciones, cuatro de ellos se pueden usar para comparar cualquier combinación de enteros, flotantes o caracteres. Cuando comparamos un caracter con otro carácter o un número, se compara realmente el valor UNICODE de dicho caracter:

```
String animal = "unknown";
int weight = 700;
char sex = 'm';
double colorWaveLenght = 1.630;
if(weight >= 500) { animal = "elephant"; }
if(colorWaveLenght > 1.621) { animal = "gray " + animal; }
if(sex <= 'f') { animal = "female " + animal; }
```

Al final de este código, `animal` es *gray elephant*.

Después tenemos los operadores de igualdad, que deben operar sobre dos elementos del mismo tipo o compatibles. Estos tipos pueden ser números, caracteres, booleanos o referencias a objetos.

Dos cosas tenemos que tener en cuenta:

- Cuando un flotante se compara con un entero y su valor es el mismo, su comparación es `true`.

  ```
  if(5.0 == 5L) { System.out.println("They are equal"); }
  ```

- Cuando se comparan referencias a objetos se compara su patrón de bits, es decir, el objeto al que están referenciando

  ```
  JButton a = new JButton("Exit");
  Jbutton b = a;
  if(a == b) { System.out.println("They are equal"); }
  ```

Cuadro 12: Exam tip 12

Cuidado con confundir los opeadores = y == en comparaciones. La siguiente expresión es legal:

```
boolean b = false;
if(b = true) { System.out.println("b is true"); }
else{ System.out.println("b is false"); }
```

Se mostrará *b is true*. En la comparación hemos usado el operador de asignación, y el valor de una asignación es siempre el valor asignado, por lo que `b = true` se evalua a `true`. Esto es válido sólo con booleanos, puesto que, a diferencia de lenguajes como C++, los únicos valores que evaluan a `true` en una comparación son los booleanos. El siguiente no compilará:

```
int x = 1;
if(x = 0) {}
```

Al ser `x` un entero, la expresión `x = 0` se evalua a `0`, y dicho no valor no es válido para el `if`.