JAVA

Curso práctico de formación

para la preparación del examen de certificación
Java SE Programmer I: IZ0-808

Antonio Martín Sierra

Alfaomega

libros RC

JAVA

Curso práctico de formación

para la preparación del examen de certificación
Java SE Programmer I: IZ0-808

JAVA

Curso práctico de formación

para la preparación del examen de certificación
Java SE Programmer I: IZ0-808

Antonio Martín Sierra

Alfaomega

libros

Diseño de colección y pre-impresión: Grupo RC
Diseño de cubierta: Cuadratín

Datos catalográficos
Martín, Antonio
Java. Curso práctico de formación;
para la preparación del examen de certificación Java SE
Programmer I: IZ0-808
Primera Edición
Alfaomega Grupo Editor, S.A. de C.V., México
ISBN: 978-607-538-267-8
Formato: 17 x 23 cm Páginas: 284

Java. Curso práctico de formación; para la preparación del examen de certificación Java SE Programmer I: IZ0-808
Antonio Martín Sierra
ISBN: 978-84-947170-6-2 edición original publicada por RC Libros, Madrid, España.

Primera edición: Alfaomega Grupo Editor, México, junio 2018

Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, 06720, Ciudad de México.

Miembro de la Cámara Nacional de la Industria Editorial Mexicana
Registro No. 2317

Pág. Web: **http://www.alfaomega.com.mx**
E-mail: **atencionalcliente@alfaomega.com.mx**

ISBN: 978-607-538-267-8

Nota importante:
La información contenida en esta obra tiene un fin exclusivamente didáctico y, por lo tanto, no está previsto su aprovechamiento a nivel profesional o industrial. Las indicaciones técnicas y programas incluidos, han sido elaborados con gran cuidado por el autor y reproducidos bajo estrictas normas de control. ALFAOMEGA GRUPO EDITOR, S.A. de C.V. no será jurídicamente responsable por: errores u omisiones; daños y perjuicios que se pudieran atribuir al uso de la información comprendida en este libro, ni por la utilización indebida que pudiera dársele.

Edición autorizada para venta en México y todo el continente americano.

Impreso en México. Printed in Mexico.

Empresas del grupo:

México: Alfaomega Grupo Editor, S.A. de C.V. – Dr. Isidoro Olvera (Eje 2 sur) No. 74, Col. Doctores, C.P. 06720, Del. Cuauhtémoc, Ciudad de México. Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396 E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A. – Calle 62 No. 20-46, Barrio San Luis, Bogotá, Colombia, Tels.: (57-1) 746 0102 / 210 0415 – E-mail: cliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor, S.A. – Av. Providencia 1443. Oficina 24, Santiago, Chile
Tel.: (56-2) 2235-4248 – Fax: (56-2) 2235-5786 – E-mail: agechile@alfaomega.cl

Argentina: Alfaomega Grupo Editor Argentino, S.A. – Av. Córdoba 1215 piso 10, C.P. 1055, Buenos Aires, Argentina. – Tel./Fax: (54-11) 4811-0887 y 4811 7183 – E-mail: ventas@alfaomegaeditor.com.ar

Índice

Prólogo

INTRODUCCIÓN

En un mundo como el actual, en el que existen un gran número de profesionales en las diferentes áreas de las TI, la posesión de las certificaciones de los fabricantes constituye un hecho diferenciador que, ante igualdad de conocimientos y experiencia, permitirá a las personas que dispongan de alguna certificación, situarse en una mejor posición a la hora de conseguir un puesto de trabajo o una mejora profesional, frente a aquellos que no la posean.

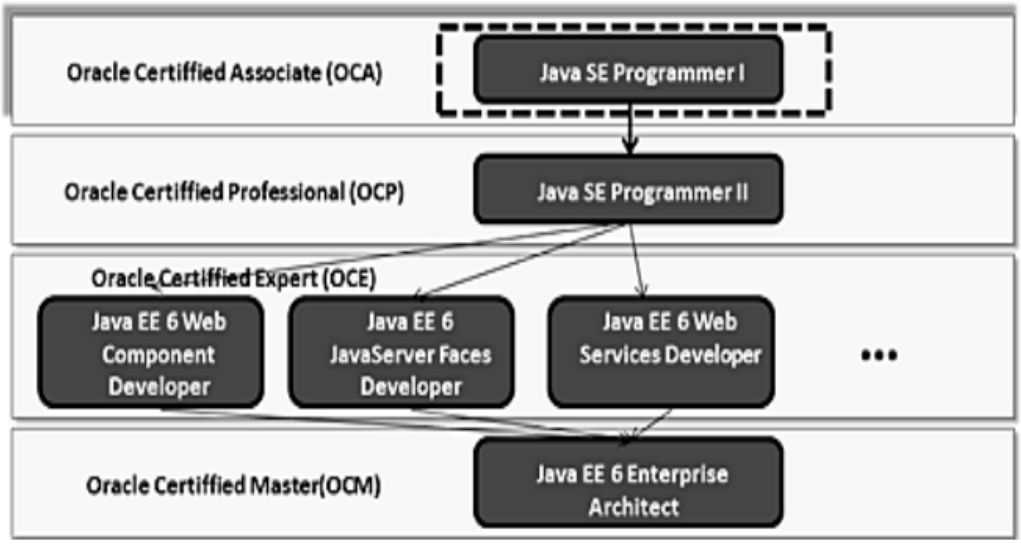
En este libro nos adentramos en el mundo de las certificaciones Java, centrándonos en la preparación del primer examen dentro del stack de certificaciones existentes sobre este lenguaje-tecnología.

OBJETIVOS

El objetivo de este libro es preparar al lector para superar el examen de certificación **Java SE Programmer I (código iz0-808),** que nos capacitaría como Programador Java Oracle Asociado.

Como se ha indicado, este examen corresponde al primer nivel dentro del conjunto de certificaciones Java y nos abriría la puerta a adquirir las certificaciones de especialización en las diferentes tecnologías que forman la plataforma Java.

En la siguiente figura podemos ver el conocido como Certification Path o pila de certificaciones Java:



Los objetivos de este examen se centran en el conocimiento detallado del lenguaje Java, así como de las clases de uso general, como las clases para el manejo de cadenas, fechas o colecciones. Concretamente, los objetivos a cubrir por el examen están organizados en nueve bloques:

- Conceptos básicos de Java
- Tipos de datos de Java
- Operadores y estructuras de decisión
- Arrays
- Estructuras repetitivas
- Métodos, constructores y encapsulación
- Estudio de la herencia
- Excepciones
- APIs de uso general

CARACTERÍSTICAS DEL EXAMEN DE CERTIFICACIÓN

El examen de certificación se realiza en centros autorizados por Oracle, repartidos a lo largo del mundo.

En la siguiente dirección, podemos encontrar información sobre los pasos que debemos seguir para realizar el examen:

https://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=5001&get_params=p_exam_id:1Z0-808

Como datos más interesantes, comentar que el examen tiene un precio aproximado de 212 euros y consta de 70 preguntas tipo test (algunas con respuesta única y otras con respuesta múltiple) que deberán resolverse en un tiempo máximo de 150 minutos. Aunque las respuestas incorrectas no restan, para superar el examen es necesario obtener un 65% de preguntas acertadas.

A QUIÉN VA DIRIGIDO EL LIBRO

Este libro está dirigido a programadores en cualquier lenguaje de programación que quieran introducirse en Java.

Pero en general, tanto si se trata de programadores iniciados como programadores con experiencia, el libro resultará ser una herramienta muy valiosa para la preparación del examen de certificación, puesto que las preguntas del examen se centran en detalles de uso poco habitual que requieren de una preparación especial.

ESTRUCTURA Y METODOLOGÍA DEL LIBRO

El libro está organizado en nueve capítulos, que se corresponden con cada uno de los bloques que forman los objetivos de examen. En cada capítulo, además de explicarse cada concepto con su correspondiente ejemplo de código, se hace especial hincapié en aquellos aspectos relevantes que se deben de tener en cuenta de cara a las posibles preguntas de examen.

Al final de cada capítulo se incluyen un juego de preguntas tipo de examen, muy parecidas a las que podemos encontrar en los exámenes reales, incluso están enunciadas en inglés para ir acostumbrando al lector al escenario real. Después de cada batería de preguntas, se incluyen las soluciones a cada una de ellas con la explicación correspondiente.

Este no es un libro para aprender a programar en Java, está orientado a proporcionar un conocimiento profundo del lenguaje y de todos aquellos detalles más desconocidos o de uso menos habitual en el día a día de la programación, pero que pueden ser objeto de alguna pregunta de examen. Siguiendo el orden de estudio de los capítulos, prestando atención a las diferentes indicaciones que se dan en los mismos, y realizando y razonando las preguntas tipo propuestas al final de cada capítulo, las posibilidades de superar el examen de certificación Java Programmer I son muy elevadas.

EL AUTOR

Antonio Martín Sierra nació en Madrid en 1966. Es diplomado en Ingeniería Técnica de Telecomunicación por la Universidad Politécnica de Madrid y es Programador Java Certificado.

Su trayectoria profesional ha estado ligada desde el principio de su carrera a la formación. Inicialmente, impartía cursos de Electrónica para diversas Escuelas de formación de Madrid. Desde mediados de los 90, orientó su carrera al mundo del desarrollo software, especializándose en la tecnología Java. Ha impartido numerosos cursos de formación sobre Java, Java EE y frameworks en importantes compañías como Telefónica, Indra, IBM y Core Networks, y ha publicado varios libros sobre tecnologías relacionadas con estas áreas.

Además de su experiencia como formador, ha colaborado con diferentes empresas en el diseño, gestión e implementación de soluciones formativas e-learning. Ha sido uno de los responsables del programa Empleo Digital de Telefónica Educación y desarrolla contenidos formativos en las áreas de su especialidad para empresas y centros de formación.

1 Fundamentos de Java

CARACTERÍSTICAS DE JAVA

El lenguaje de programación Java es, sin lugar a dudas, el más utilizado hoy en día para la creación de aplicaciones informáticas. Ya sean aplicaciones de escritorio, para dispositivos móviles y, muy especialmente, aplicaciones para la Web, Java suele ser la opción preferida por los programadores y empresas de desarrollo.

Origen y evolución

La primera versión del lenguaje Java fue lanzada por Sun Microsystems en mayo de 1995, se trataba de la versión 1. Después vendrían la 1.1, 1.2, 1.3 y 1.4. En el año 2004 se lanzó la versión 1.5, que trajo notables mejoras respecto a su predecesora, hasta tal punto que a partir de ese momento dejó de llamarse Java 1.5 y pasó a ser Java 5. Desde entonces, han ido apareciendo Java 6, Java 7, Java 8 y la última hasta el momento, Java 9.

Desde su aparición a mediados de los 90, Java no ha hecho más que crecer y extenderse. Ya en su primera versión, incorporó una característica que comentaremos seguidamente y que hizo que tuviera gran aceptación por parte de la mayoría de las empresas de software del momento, se trata de la posibilidad de compilar una vez y ejecutar en cualquier parte, algo que sin lugar a dudas fue una auténtica novedad en aquella época.

Principales características de Java

El conocimiento de las características del lenguaje Java es uno de los objetivos del examen de certificación y, por tanto, puedes encontrar alguna pregunta sobre ello. Así pues, vamos a comentar las principales características de la tecnología Java en general y el lenguaje en particular. Pasamos a enumerarlas a continuación:

- **Lenguaje orientado a objetos**. Posiblemente, hoy en día esto no resulte una novedad, pero en la época en la que apareció Java había muy pocos lenguajes que fueran orientados a objetos. En los lenguajes orientados a objetos, el código se escribe dentro de clases, organizado en funciones (métodos), que son invocados a través de objetos o instancias de la clase. Estos lenguajes exponen una serie de características de orientación a objetos como herencia, sobrecarga y sobrescritura, polimorfismo y encapsulación, que dotan al lenguaje de una gran potencia.
- **Portabilidad.** Quizá esta sea la principal característica de Java, y es que un programa escritor en Java se **compila una vez y se ejecuta en cualquier parte**. El resultado de la compilación (bytecodes) es independiente de la plataforma, puede ejecutarse igual en un sistema operativo Windows, en Linux, etc. Esto es gracias a un elemento clave de la tecnología Java, conocido como Máquina Virtual o Java Virtual Machine (JVM), que estudiaremos a continuación.
- **Encapsulación**. Es una propiedad de la orientación a objetos y a la que se le da especial importancia en el examen de certificación. La encapsulación consiste en proteger a los atributos de la clase con modificador privado para que no sean accesibles directamente desde el exterior, permitiendo el acceso a los mismos a través de métodos. Hablaremos de la encapsulación con más detenimiento en un capítulo posterior.
- **Robusto.** La memoria es gestionada de forma automática por la JVM, de forma que no se permite el acceso a ella desde código y se evitan problemas de violación de acceso a partes de la memoria.
- **Seguro**. El código Java se ejecuta en un entorno controlado por la JVM, impidiendo operaciones dañinas sobre el equipo.

Compilación de un programa Java

Los programas Java se escriben en archivos .java, que es lo que se conoce como **código fuente del programa**. Como se ha dicho antes, este código fuente se estructura en clases, de modo que al compilarlo se generará un archivo .class, conocido como **bytecodes**, por cada clase definida en el archivo.

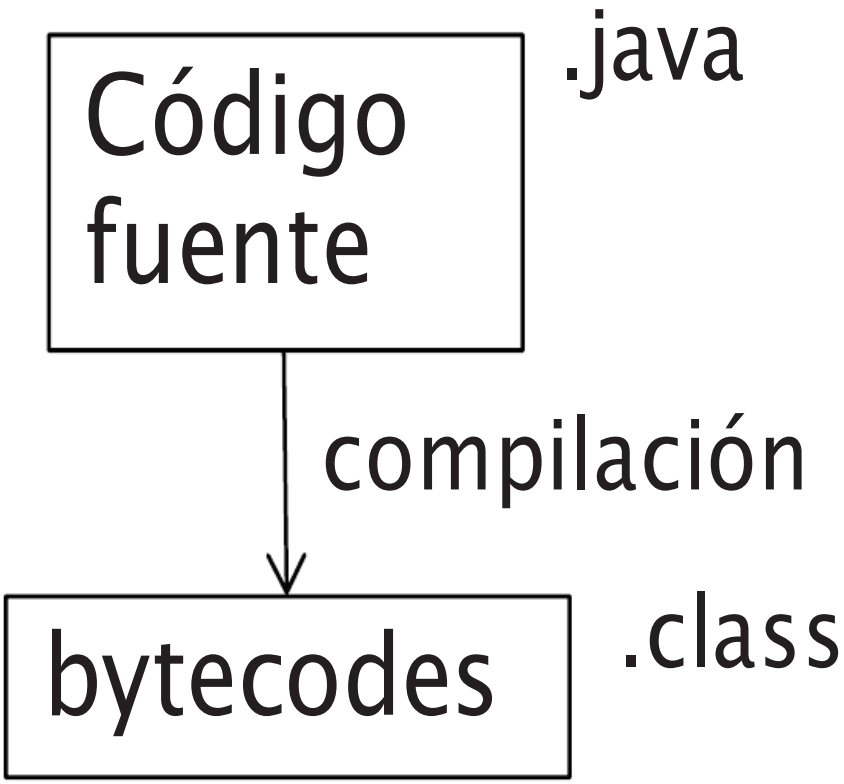


Fig. 1-1. Compilación en Java

Los bytecodes son independientes de la plataforma y se podrán ejecutar en cualquier sistema operativo que cuente con la Máquina Virtual Java (JVM).

Java Virtual Machine (JVM)

La Máquina Virtual Java, o JVM, es el software que se encarga de traducir en tiempo de ejecución los bytecodes a instrucciones comprensibles por el sistema operativo.

Existen versiones de JVM para cada todos los sistemas operativos modernos, lo que permite que un programa compilado (conjunto de archivos .class) pueda ser ejecutado en cualquier sistema operativo.

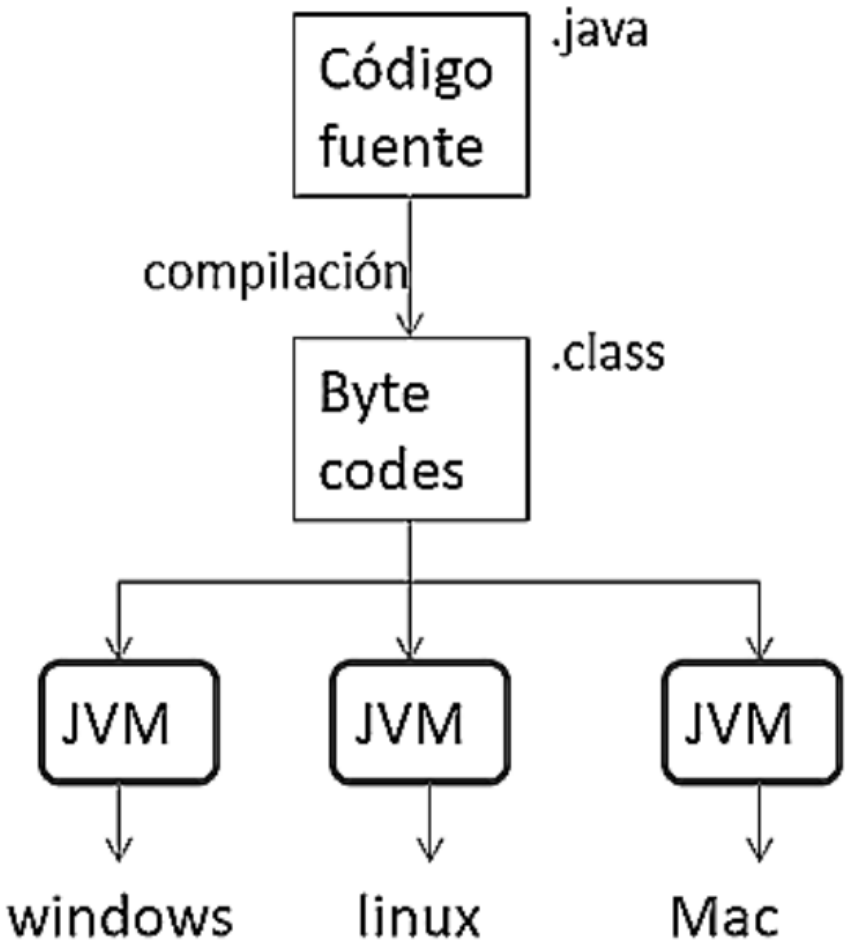


Fig. 1-2. La Máquina Virtual Java

Las primeras máquinas virtuales eran algo lentas a la hora de interpretar las instrucciones de bytecodes, lo que ya desde el principio le valió algunos enemigos a esta tecnología. Sin embargo, en las posteriores versiones se fue mejorando el rendimiento con la incorporación de los compiladores JIT, que compilan ciertas partes de código que tienen que interpretar para no repetir el proceso de interpretación cuando tienen que volver a ejecutar estos bloques.

Además del intérprete, una JVM incluye otros componentes software, como el **Gestor de multitarea**, el **Recolector de basura o Garbarge Collector** que se encarga de la gestión de memoria, el **Cargador de clases** o Class loader y el ya mencionado **Compilador JIT**.

CREANDO UN PROGRAMA JAVA: LAS CLASES

Según hemos indicado en el apartado anterior, todo programa Java se organiza en clases, pero vamos a ver cómo define una clase y cuál es su estructura.

Clase Java

Todo el código de un programa Java se escribe dentro de una clase. El objetivo de la clase es definir el comportamiento de los objetos de la misma. Este comportamiento se implementa en forma de **atributos**, que sirven para fijar las características de los objetos de la clase, y de **métodos** o funciones que definen las operaciones que se podrán realizar sobre dichos objetos.

Una vez definida la clase, se podrán crear objetos de la misma para poder hacer uso de los atributos y métodos. Dicho de otra manera, la clase es el molde y los objetos son el elemento "físico", creado a partir del molde, sobre el que se aplican las características y los comportamientos definidos.

Como ya hemos indicado, las clases se definen en archivos .java y al compilarlos, se generarán tantos .class como clases estén definidas en el código fuente.

Estructura de una clase

Una clase se define con la palabra reservada *class*, seguida del nombre de la clase. Entre llaves ({ y }) se define el contenido de la misma.

```
class NombreClase{



}
```

Un archivo .java puede contener varias clases, aunque **solo una con modificador public**, cuyo nombre debe coincidir con el del archivo .java.

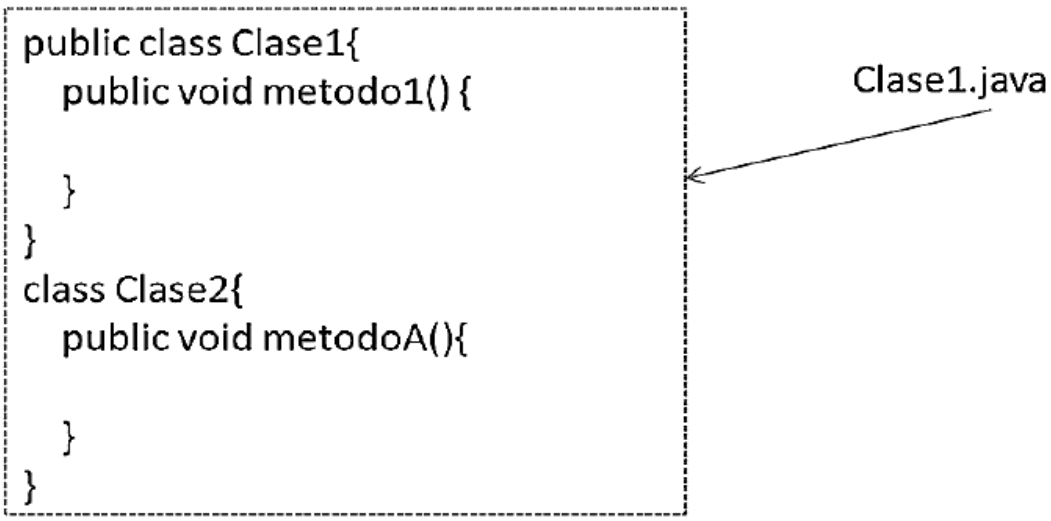


Fig. 1-3. Contenido archivo .java

Debes de tener cuidado ante una pregunta de examen en la que aparezcan dos clases públicas en un mismo archivo .java, esto supondría directamente un error de compilación, como también lo sería el hecho de que el nombre de la clase pública no coincida con el del archivo.

Como ya se ha indicado, una clase puede contener:

- **atributos**. Variables que almacenan propiedades de los objetos de la clase.
- **constructores**. Bloques de código que se ejecutan al crear objetos de la clase.
- **métodos**. Funciones que realizan tareas sobre los objetos de la clase.

Aquí tenemos un ejemplo completo de una definición de clase:

```
public class Clase1{

    int k; //atributo

    public Clase1(){ //constructor

    }


   public void metodo1() { //método

    }

}
```

A partir de una clase, se pueden crear objetos de la misma para poder hacer uso de los métodos. Aunque ya lo veremos en capítulos posteriores, la creación de un objeto se realiza utilizando el operador **new**:

```
Clase1 c=new Clase1();
```

Una vez que se tiene el objeto referenciado por la variable c, se puede utilizar el operador punto (.) para llamar a los métodos sobre el objeto:

```
c.metodo1(); //ejecuta el método
```

Empaquetado de una clase

Las clases se organizan en **paquetes** (directorios). Un paquete puede contener varios archivos .class y a su vez otros subpaquetes.

Los paquetes se definen con la sentencia ***package*** *nombrepaquete* al principio del archivo .java, de modo que todas las clases definidas en dicho archivo estarán en el mismo paquete. En el siguiente ejemplo, las clases Clase1 y Clase2 estarán en el paquete mypack:

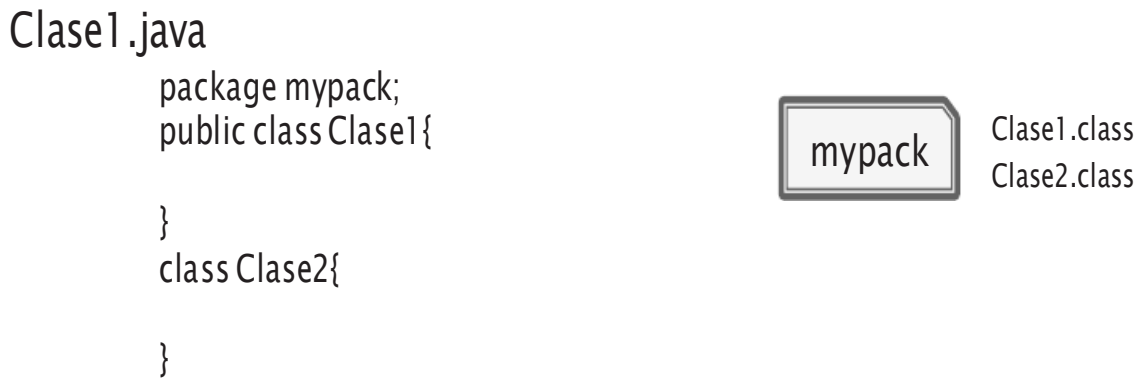


Fig. 1-4. Empaquetado de clases

Aunque insistiremos en ello más adelante, hay que destacar que la sentencia package debe ser **la primera del archivo .java**.

El método main

El método *main()* representa el punto de entrada de un programa Java. Es el método que lanza la JVM cuando se le indique que tiene que ejecutar una clase. Entre todas las clases de un programa Java, deberá haber una que incluya este método.

Es importante conocer el formato exacto del método *main()*, ya que alguna pregunta de examen puede ir en esa dirección. El formato exacto es el que se indica a continuación:

```
public class Clase1{
    public static void main(String [] args) {
        :
    }
}
```

La única variación admitida sería que, en lugar de definir un array de String como parámetro, se puede indicar un número variable de argumentos:

```
public class Clase1{
    public static void main(String ... args) {
        :
    }
}
```

Las siguientes definiciones del método main **no** serían correctas:

```
static void main(String[] args) //falta public
public void main(String[] args) //falta static
public static int main(String[] args) //tipo de devolución
                                      //incorrecto
public static void Main(String[] args) //nombre incorrecto
```

En los casos anteriores, estos métodos **no provocarían error de compilación** pues son sintácticamente correctos, lo que sucedería es que al intentar ejecutar la clase se produciría un error de ejecución ya que la JVM no encontraría el método main con el formato adecuado.

COMPILACIÓN Y EJECUCIÓN DE PROGRAMAS JAVA

Anteriormente, hemos indicado que al compilar un archivo .java se generan uno o varios .class, y que al ejecutar una clase la JVM busca el método *main()*. Pero ¿cómo se compila un código fuente Java? ¿Cómo se ejecuta una clase? Uno de los objetivos del examen de programador Java asociado es, precisamente, la utilización de los comandos para realizar estas operaciones. Vamos a ver cuáles son y cómo se utilizan.

Herramientas JDK

El comando para la compilación de un archivo .java se llama javac.exe, mientras que el que se emplea para la ejecución de una clase es el java.exe. Ambos se encuentran en el directorio de instalación del JDK, dentro de la subcarpeta bin:

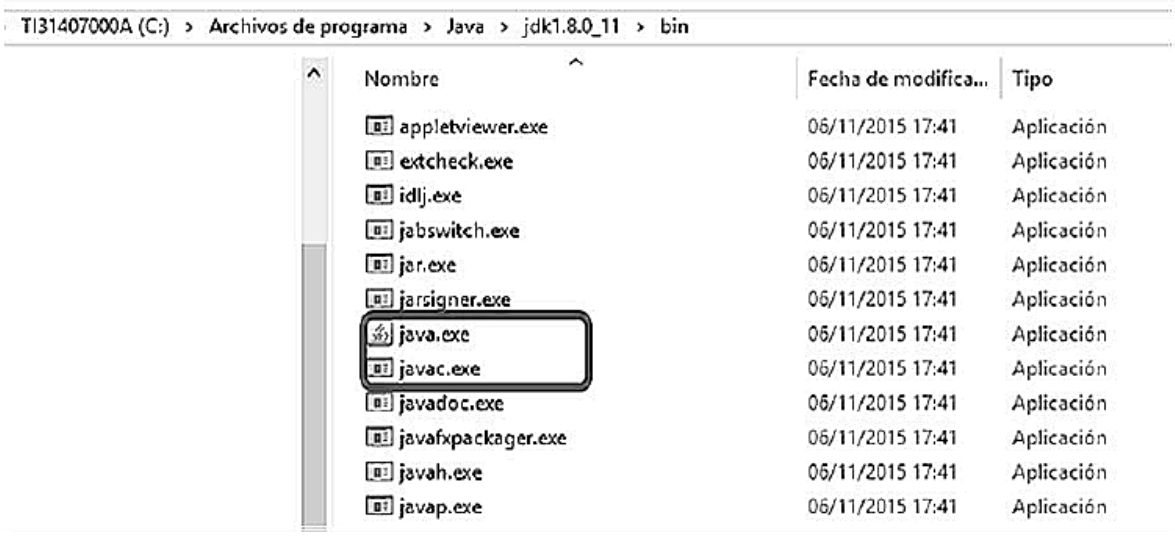


Fig. 1-5. Comandos del JDK

El JDK, o Java Development Kit, proporciona las herramientas básicas para poder compilar y ejecutar programas Java, así como las clases que forman el Java Standard Edition (Java SE).

Para descargar el JDK accederemos a la siguiente dirección donde encontramos el pack de descarga para última versión de Java(Java 9):

http://www.oracle.com/technetwork/java/javase/downloads/jdk9-downloads-3848520.html

Compilación de un archivo

Para compilar un archivo de código fuente .java, nos situaremos desde la línea de comandos en la carpeta donde se encuentra dicho archivo y allí escribiremos:

```
>javac NombreArchivo.java
```

Si las clases del archivo están definidas en un paquete y queremos que, además de los .class, se generen los directorios correspondientes a los paquetes, entonces deberíamos utilizar el siguiente comando:

```
>javac -d . NombreArchivo.java
```

El . indica que la generación de los directorios de los paquetes se realice en el directorio actual.

Si el archivo de código fuente contiene errores de compilación, se mostrarán los mensajes correspondientes en la propia línea de comandos. Si no hay errores, se generará un archivo .class por cada clase contenida en el código.

Ejecución de un programa

La ejecución de un programa Java consiste en la ejecución de la clase que contiene el método main, así que desde el directorio en el que realizamos la compilación, escribiremos el siguiente comando para ejecutar dicha clase:

```
>java NombreClase
```

Fíjate en una cosa importante, y es que a la hora de ejecutar una clase se **indica el nombre de la misma, no el archivo .class que la contiene**. Esto debemos de tenerlo muy presente de cara a alguna posible pregunta de examen.

Si la clase se encuentra dentro de algún paquete, se debería indicar el nombre cualificado de la clase, es decir, el nombre del paquete seguido de un punto y el nombre de la clase:

>java paquete.NombreClase

Por ejemplo, supongamos que tenemos el siguiente archivo Prueba.java:

```
package principal;

public class Prueba{

    public static void main(String[] args){

          System.out.println("hello");

    }

}
```

Para compilar el archivo escribiríamos:

```
>javac -d . Prueba.java
```

Después, para ejecutar la clase:

```
>java principal.Prueba
```

Tras el comando anterior se mostrará en pantalla:

hello

Argumentos de línea de comandos

Durante la ejecución de la clase principal se le pueden pasar al método main una serie de argumentos por línea de comandos, estos valores el método los recibe en el parámetro array de cadenas de caracteres.

Los argumentos se pasarían a continuación del nombre de la clase, separados por espacios:

>java NombreClase arg1 arg2

Por ejemplo, si queremos ejecutar la clase Prueba enviando los tres primeros días de la semana sería:

>java Prueba lunes martes miercoles

Si algún argumento debe contener espacios, entonces la cadena completa del argumento se escribe entre comillas:

>java principal.Prueba "argumento primero" segundo

En el caso de que la clase Prueba esté definida de la siguiente manera:

```
package principal;

public class Prueba{

    public static void main(String[] args){

          System.out.println(args[0]);

          System.out.println(args[1]);

    }

}
```

Al ejecutar el comando anterior se imprimirá por pantalla:

argumento primero

segundo

Debemos prestar atención a los índices del array cuando veamos alguna pregunta de examen en la que se accede a alguna de las posiciones del mismo, ya que podría aparecer algún acceso a posiciones que están fuera del array recibido, lo que provocaría directamente una excepción de tipo:

ArrayLindexOutOfBoundsException

Por ejemplo, si tuviéramos la clase:

```
public class Print{

    public static void main(String[] args){

          System.out.println(arg[0]+args[1]);

    }

}
```

Al ejecutarla con el siguiente comando se produciría la citada excepción, puesto que solo se le pasa un argumento y, por tanto, **la posición args[1] no existe**:

>java Print Java

IMPORTACIONES DE CLASES

Cuando queremos utilizar una clase que se encuentra en un paquete diferente, es necesario importarla dentro del archivo .java donde la queremos utilizar. Para ello, utilizaremos la sentencia **import**.

Sintaxis

Con import podemos importar:

- Una clase:

  import java.util.ArrayList;

- Todas las clases de un determinado paquete:

  import java.util.*;

- Los elementos estáticos de una clase:

  import static java.lang.Math.*;

Presta atención a los siguientes errores de compilación en el uso de import:

import java.util; //se debe indicar la clase dentro del paquete

import static java.util; //se debe indicar el elemento estático a importar

import static java.util.ArrayList; //igual que el anterior

Colocación de la sentencia import

Las sentencias import se deben situar después de *package*, que es la primera sentencia del archivo .java, y antes de la definición de la clase:

```
package mipaquete;

import  java.util.*;

public class  Clase{

:

}
```

Debes tener cuidado ante una posible pregunta de examen en la que aparezcan involucradas estas sentencias import y package, ya que, por ejemplo, un código como el siguiente provocaría un **error de compilación**:

```
import  java.util.*; //debe aparecer después de package

package mipaquete;

public class  Clase{

}
```

PREGUNTAS TIPO EXAMEN

Pregunta 1

Which of the following are legal entry point methods that can be run from the command line? (Choose 2)

A. private static void main(String[] args)

B. public static int main(String[] args)

C. public void main(String[] args)

D. public static void test(String[] args)

E. public static void main(String[] args)

F. public static void main(String data[])

G. None of the above.

Pregunta 2

Which three statements describe the object-oriented features of the Java language?

A. Objects cannot be reused.

B. A subclass can inherit from a superclass.

C. Objects can share behaviors with other objects.

D. A package must contain more than one class.

E. Object is the root class of all other objects.

F. A main method must be declared in every class.

Pregunta 3

Which statement is true about Java byte code?

A. It can run on any platform.

B. It can run on any platform only if it was compiled for that platform.

C. It can run on any platform that has the Java Runtime Environment.

D. It can run on any platform that has a Java compiler.

E. It can run on any platform only if that platform has both the Java Runtime Environment and a Java compiler.

Pregunta 4

What is the name of the Java concept that uses access modifiers to protect variables and hide them within a class?

A. Encapsulation

B. Inheritance

C. Abstraction

D. Instantiation

E. Polymorphism

Pregunta 5

Which of the following files compiles without errors? (Choose 2)

A. F1.java:

```
import java.util.*;

package mypackage;

public class F1{}
```

B. F2.java:

```
package mypackage;

import java.util.*;

public class MyClass{}
```

C. F3.java:

```
package mypackage;

import java.util.*;

public class F3{}
```

D. F4.java:

```
class MyClass{}

public class F4{}
```

E. F5.java:

```
public class MyClass{}

public class F5{}
```

Pregunta 6

Given the following files:

Company.java

```
package comp;

public class Company{}
```

Employ.java

```
package comp.emps;

publoc class Employ{}
```

Using.java

```
1. paclage using;

2. //insert code here

3. public class Using{

4.        Company cp;

5.        Employ em;

6.}
```

Which code fragment, when inserted at line 2, enables the code to compile?

A.
```
import comp.*;
import java.comp.emps.*;
```
B.
```
import comp.Company;
import comp.emps;
```
C.
```
import comp.*;
import emps.*;
```
D.
```
import comp.*;
import comp.emps.*;
```

Pregunta 7

Given the following class:

```
public class Test{

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

```
                System.out.println("Hello "+args[0]);
            }
}
```

Which set of commands prints **Hello Student** in the console?

A. javac Test.java

   java Test.class Student

B. javac Test

   java Test Student

C. javac Test.java Student

   java Test

D. javac Test.java

   java Test Student

Pregunta 8

Given:

```
public class Principal {
   public static void main(String[] args) {
      System.out.println("String "+args[0]);
   }
   public static void main(int[] args) {
      System.out.println("int "+args[0]);
   }
```

```
    public static void main(Object[] args) {
      System.out.println("Object "+args[0]);
    }
}
```

And commands:

```
javac Principal.java
java Principal 1 2 3
```

Which is the result?

A. int 1

B. String 1

C. Object 1

D. Compilations fails

E. An exception is thrown

Pregunta 9

Given the following class, which of the following calls print out My Car?

```
public class Test {
public static void main(String[] name) {
System.out.println(name[1]);
} }
```

A. java Test Blue My Car

B. java Test Blue "My Car"

C. java Test My Car Blue

D. java Test "My Car" Blue

E. java Test.class Blue "My Car"

F. Does not compile.

Pregunta 10

Which of the following are true? (Choose 2)

A. javac compiles a .class file into a .java file.

B. javac compiles a .java file into a .bytecode file.

C. javac compiles a .java file into a .class file.

D. Java takes the name of the class as a parameter.

E. Java takes the name of the .bytecode file as a parameter.

F. Java takes the name of the .class file as a parameter

SOLUCIONES

1. E y F. Ambas corresponden con la definición exacta del método main. De hecho, son prácticamente idénticas, solo se diferencian en la posición de los corchetes en la declaración del parámetro de tipo array, pero es que los corchetes pueden colocarse delante o detrás de la variable durante la declaración de un array.

2. B, D y E. La respuesta B describe una de las características de la herencia. D y E son ciertas, A no es cierta porque los objetos pueden ser reutilizados dentro de un programa. C no es cierta porque no hay ningún elemento que permita compartir datos entre objetos y F no es cierta porque solo tiene que haber un punto de inicio dentro de un programa, es decir, una única clase con método main.

3. C. Para que los bytecodes puedan ser ejecutados en cualquier máquina, es necesario que cuente con el entorno de ejecución Java, que es la máquina virtual y las clases de Java estándar.

4. A. El enunciado corresponde a uno de los principios de la encapsulación.

5. C y D. La A es incorrecta porque la instrucción import debe ir después de package. La B tampoco compila porque el nombre de la clase pública no coincide con el nombre del archivo .java y la E tampoco es correcta por la existencia de dos clases públicas en un mismo archivo.

6. D. En la respuesta A, el paquete java.comp.emps no existe. La B no compila porque la importación es incorrecta, no se pueden importar paquetes, se deben importar clases de paquetes. La C tampoco es correcta porque con el * importamos todas las clases que se encuentren directamente en un paquete, pero no las de los subpaquetes.

7. D. La compilación de un archivo Java se realiza indicando a continuación del comando javac el nombre del archivo con su extensión, por lo que las respuestas B y C quedan descartadas. En la ejecución se indica el nombre de la clase, no el archivo, seguido de la lista de parámetros, por lo que la A también es incorrecta.

8. B. El programa compila perfectamente porque es posible tener varios métodos con el mismo nombre, siempre que se diferencien en los parámetros. El único método main que ejecutará Java será el que recibe un array de cadenas de caracteres, por lo que la respuesta es la B.

9. B. La E es incorrecta porque se debe indicar el nombre de la clase, no el nombre del archivo. Con la A solo imprimiría My, con la C Car y con la D Blue. Con la B, el argumento que ocupa la posición 1 es la cadena "My Car", por lo que esa es la respuesta.

10. C y D. A es incorrecta porque es justo al revés, la B es incorrecta porque la extensión .bytecode no existe, por el mismo motivo, la E tampoco es correcta. F es incorrecta porque no es el nombre del archivo, sino el de la clase el que se emplea en la llamada a java.

2 Tipos de datos

VARIABLES

En Java, los datos, independientemente de su tipo, se manejan a través de **variables**. Una variable es una sección de memoria, a la que se le asigna un nombre o identificador, en la que se almacenan los datos manejados por el programa. Para operar con un dato, se utilizará la variable que lo contiene.

Así pues, antes de entrar a analizar en detalle los tipos de datos que existen en Java, vamos a abordar el estudio de las variables, su declaración, inicialización y ámbito.

Declaración de una variable

En Java es obligatorio declarar una variable antes de ser utilizada. Declarar la variable consiste en indicarle al compilador que vamos a utilizar una variable de un determinado tipo, asignándole un identificador:

Fig. 2-1. Declaración de una variable

En el ejemplo de la figura anterior, hemos declarado una variable con el identificador "mivar", que va a almacenar números enteros.

Para asignar posteriormente un valor a la variable previamente inicializada, utilizamos la expresión:

identificador=valor;

En el caso anterior, para asignar un valor a la variable *mivar* sería:

```
mivar=10;
```

Es posible declarar e inicializar una variable en una única instrucción:

```
int v=2;
```

También se pueden declarar varias variables en una misma línea, e incluso inicializar algunas de ellas:

```
int p, s, a=5;
```

Identificadores de variables

Como hemos visto, a la hora de declarar una variable se le asigna un identificador que luego utilizamos a lo largo del programa para referirnos a la variable. Pero a la hora de definir un identificador no vale cualquier combinación de caracteres, y es posible encontrar alguna pregunta de examen sobre este aspecto, por lo que vamos a indicar las reglas que deben seguirse a la hora de definir un identificador para una variable:

- Se puede utilizar cualquier combinación de letras, números y los símbolos $ y _
- La regla anterior tiene dos restricciones a tener en cuenta:
    - No se pueden utilizar palabras reservadas de Java como identificador (incluido *goto*).
    - El identificador no puede comenzar por un carácter numérico.

Como ves, se trata de unas sencillas reglas que durante al examen debemos de tener presentes de cara a determinar si un identificador es o no válido. Si se incumple

alguna de estas reglas, se producirá un error de compilación en la instrucción de declaración de la variable. Veamos algunos ejemplos:

```
int _1=10; //correcto

char break; //error, se utiliza una palabra reservada

          //como identificador

int 3aj; // error, no puede comenzar por un número

float car.t; //error, el punto no es carácter válido

char main; //correcto, un nombre de método NO es una

                //palabra reservada
```

Ámbito de una variable

El ámbito o visibilidad de una variable se refiere a la visibilidad de una variable, viene determinado por el lugar en el que la variable está declarada. En este sentido, tenemos dos ámbitos posibles de una variable:

- **Variable atributo**. Se declara a nivel de clase, fuera de los métodos. Estas variables son compartidas por todos los métodos de la clase
- **Variable local**. Se declara dentro de un método y solo son visibles dentro de ese método.

```
class MiClase{

    int n; //variable atributo

    public void metodo1(){

          int c; //variable local

          n=3;//acceso a variable atributo

    }

    public void metodo2(){
```

```
            c=c+2;//error de compilación, sin acceso a variable
                  //c por ser local a método1
    }
}
```

En Java puede darse el caso de que una variable local y otra atributo tengan el mismo nombre, en cuyo caso, para referirnos a la variable atributo habrá que utilizar la palabra *this*. Observa el siguiente ejemplo:

```
class MiClase{
    int n; //variable atributo
    public void metodo(){
            int c; //variable local
            int n; //local con mismo nombre que atributo
            n=10; //acceso a variable local
            this.n=3;//acceso a variable atributo
    }
}
```

Si dentro de un método se declara una variable en el interior de un bloque de código, la variable será visible solo dentro de ese bloque. En este ejemplo, la variable p solo puede ser utilizada dentro del bloque if:

```
public void metodo(){
    int n=2;
    if(n>3){
            int p=4;
            p++;
```

```
    }

    p=p+n;//error de compilación, p no es visible

}
```

Inicialización por defecto

¿Qué ocurre si dentro de un programa utilizamos una variable, por ejemplo para mostrar su contenido, sin haberla asignado previamente ningún valor? Eso dependerá del ámbito de la variable.

VARIABLES LOCALES

Las variables locales **no se inicializan implícitamente**, por tanto, NO pueden utilizarse sin haberles asignado explícitamente un valor. Observemos el siguiente código:

```
public void metodo(){

    int c;

    c=c+3; //error de compilación

}
```

Como se indica en el comentario, la instrucción c=c+3; provoca un error de compilación, ya que la variable c no ha sido inicializada y por tanto no puede utilizarse en una operación aritmética sin haberle asignado un valor previo.

VARIABLES ATRIBUTO

Las variables atributo son inicializadas implícitamente en el momento de su declaración. El valor de inicialización depende del tipo de la variable, aquí te indicamos los valores a los que se inicializaría una variable local según su tipo:

- Enteras: 0
- Decimales: 0.0

- boolean: false
- char: '\u0000' (carácter nulo)
- Objeto: null

En el siguiente ejemplo podríamos ver el resultado de operar con algunos tipos de variables atributo:

```
class Test{
    int c;
    boolean x;
    char car;
    public void metodo(){
         c=c+3;
         System.out.println(c); //3
         System.out.println(x); //false
         System.out.println(car); //carácter en blanco
    }
}
```

Variables objeto y variables de tipos primitivos

En Java distinguimos dos grandes grupos de datos. Los llamados **tipos primitivos**, que engloban a todos los tipos de datos básicos, como números, caracteres o datos de tipo lógico, y los **tipos objeto**, que representan cualquier clase o tipo de objeto Java.

Ambos tipos son tratados internamente de forma diferente por Java. Resulta esencial que conozcamos estas diferencias, no solo por las posibles preguntas de

examen al respecto, sino también por las implicaciones que puede tener en el comportamiento de los programas.

TIPOS PRIMITIVOS

Cuando definimos una variable de un tipo primitivo y asignamos un valor a la misma, la variable almacena el dato en sí:

Fig. 2-2. Almacenamiento de dato primitivo en variable

Si, posteriormente, asignamos la variable a otra, tendríamos dos copias del dato, una en cada variable:

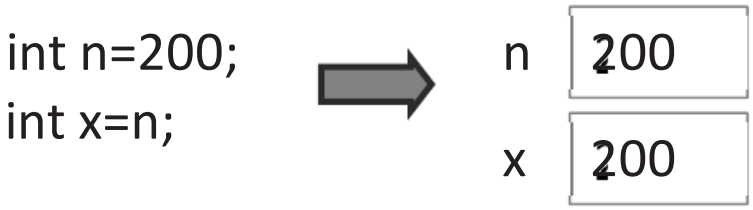


Fig. 2-3. Asignación de una variable a otra

TIPOS OBJETO

En el caso de las variables de tipo objeto, estas **no contienen el dato, sino una referencia al mismo**. Por ejemplo, supongamos que queremos trabajar con un dato de tipo Object, para ello, declaramos una variable de ese tipo y con el operador new creamos el objeto y lo asignamos a la variable:

Fig. 2-4. Referencia a un tipo objeto

Como observamos en la imagen anterior, el objeto se encuentra ocupando una zona de memoria y lo que tenemos en la variable es una **referencia** a dicha zona de memoria. El valor de esta referencia no es relevante, de hecho ni siquiera debemos preocuparnos por ello, lo importante es que a través de esta variable podemos acceder con el operador punto a los métodos de dicho objeto:

```
ob.toString();
```

Una implicación importante del hecho de que las variables objeto no contengan el objeto como tal, sino una referencia al mismo, es que si asignamos una variable objeto a otra, no tendremos dos copias de dicho objeto, sino **dos variables apuntando al mismo objeto**:

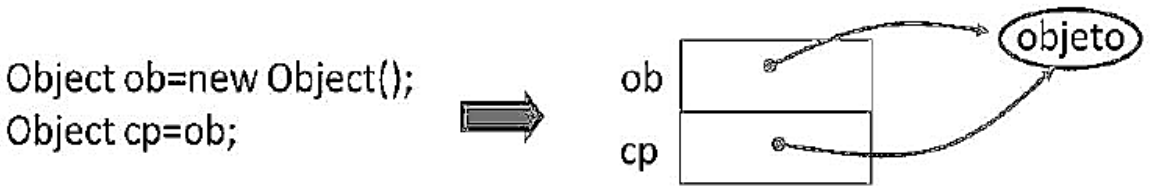


Fig. 2-5. Copia de referencia a objeto

TIPOS DE DATOS EN JAVA

Después de presentar los grandes grupos de datos y las implicaciones de manejar uno u otro a través de variables, en este apartado vamos a profundizar en el estudio de los diferentes tipos de datos existentes en Java, centrándonos especialmente en los datos de tipo primitivo.

Clasificación

Como ya indicamos en el apartado anterior, los tipos de datos se pueden clasificar en dos grandes grupos:

- **Tipos primitivos**. Representan a los tipos básicos del lenguaje, como los tipos numéricos, lógicos o caracteres. A continuación te indicamos algunos ejemplos de datos de tipo primitivo:

```
350 -> dato primitivo de tipo numérico

false-> dato primitivo de tipo lógico

'x'-> dato primitivo de tipo carácter
```

- **Tipos objeto.** Un tipo objeto es cualquier clase de Java. Los objetos de esas clases serán los valores. Algunos ejemplos de datos de tipo objeto serían:

```
new Object() -> objeto de la clase Object
"hello" -> Objeto de tipo texto (clase String)
```

TIPOS PRIMITIVOS

Java cuenta con ocho tipos primitivos de datos. En la siguiente tabla te indicamos los nombres de cada uno de los tipos y el dato que representan:

Tipo	Valores	Ejemplo
boolean	true o false	true
byte	Entero de 8 bits	39
short	Entero de 16 bits	780
int	Entero de 32 bits	59400
long	Entero de 64 bits	200000
float	Decimal 32 bits	45.6f
double	Decimal 64 bits	80.4
char	código unicode de 16 bits	'@'

Fig. 2-6. Tabla de tipos primitivos

Como vemos, hay cuatro tipos diferentes para la representación de números enteros, en función del rango numérico con el que vamos a trabajar.

Los tipos decimales, *float* y *double*, se les conoce como tipos en coma flotante, ya que el número de decimales para su representación es indefinido. El separador decimal es el punto.

Tenemos además el tipo *char*, que almacena el valor entero unicode correspondiente a un carácter, y el boolean para los tipos lógicos.

LITERALES

A un dato especifico de un tipo primitivo, como 4, 6.7 o *true*, se le conoce también como **literal**. Los literales se pueden asignar directamente a la variable del tipo que representan. Un ejemplo representativo de literal de cada tipo sería:

```
int n =34 //literal de tipo int. Todos los

          //literales enteros son considerados int

double d=3.6 //literal de tipo double. Todos los literales

          // decimales son considerados double

float g=4.7f; //para que un literal decimal

    //sea tratado como float se indica la f al final

char k='@'; //literal char entre comilla simple

boolean s=false; //los únicos literales boolean

                 //son true y false
```

Existen diversas consideraciones que debemos de tener en cuenta a la hora de tratar con literales y que debemos de tener muy presentes de cara a las preguntas de examen. Concretamente, en cuanto a la forma en la que los literales pueden ser representados y la utilización del símbolo de subrayado.

Representación

En Java 8, un literal de tipo entero puede ser representado en cuatro sistemas de numeración:

- **decimal**. Es el más utilizado, los caracteres numéricos se indican tal cual:

  ```
  int s=459;
  ```

- **octal**. Para representar un literal entero en octal indicaremos un 0 delante del número:

  ```
  int p=0431;
  ```

- **hexadecimal**. Se representan con 0x delante del número:

```
int s=0xef34;
```

- **binario**. Se representa con 0b delante del número:

```
int h=0b100110;
```

Hay que tener cuidado con algunas preguntas de examen donde aparezcan representaciones de literales enteros y fijarnos siempre en qué tipo de sistema está representado para, a partir de ahí, determinar de qué valor se trata y si es o no válido. Veamos algunos ejemplos:

```
int r=028;// error de compilación, el carácter 8 no
          //es válido en representación octal
System.out.println(32==040);//true, es el mismo valor
int bi=0b1040; //error de compilación, el carácter 4
                //no es válido en binario
```

El símbolo de subrayado "_"

Desde la versión Java 7 es posible utilizar el símbolo de subrayado "_" para representar un literal de tipo numérico, tanto entero como decimal:

```
1_000_000
37.30_49
```

La utilización de este símbolo facilita la interpretación visual por parte del programador de ciertos datos numéricos. Sin embargo, debemos tener en cuenta una importante restricción a la hora de utilizar este símbolo, y es que **debe aparecer siempre entre dígitos**, no se puede utilizar ni al principio ni al final de la cifra numérica, ni tampoco junto al punto decimal. Aquí tienes algunos ejemplos de instrucciones que no compilarían:

```
int n=_345; //no puede ser el primer carácter de la cifra
int s=0x_B2; //mismo problema anterior
```

```
double d= 45._9; //no puede aparecer junto
                 //al separador decimal
long ln=234_; //no puede ser el último carácter de la cifra
```

CONVERSIONES DE TIPO

Cualquier dato de tipo primitivo puede ser convertido a otro tipo, a **excepción de boolean**, que es incompatible con los demás y no puede haber conversiones entre este tipo y el resto.

Las conversiones pueden ser de dos tipos:

- **Implícitas**. Al realizar la asignación, Java convierte automáticamente el dato de origen en el tipo de la variable destino:

  ```
  int x=45;
  double n=x; //el entero de x se convierte a double
  char c='@';
  int s=c; //el dato char, que se almacena internamente
           //como valor entero, es convertido a int
  ```

  Siempre se realizará una conversión implícita cuando el tipo del dato origen sea de menor o igual tamaño que el destino, aunque esta regla tiene un par de restricciones. No se podrá realizar una conversión de forma implícita cuando:

  - El tipo de origen sea numérico (cualquier tipo) y el de destino char:
    ```
    byte b=5;
    char n=b;//error
    ```

  - El tipo de destino es entero y el de origen es decimal:
    ```
    float r=3.4f;
    long l=r; //error
    ```

- **Explícitas**. Cuando una conversión no pueda hacer implícitamente, siempre se podrá realizar de forma explícita indicando entre paréntesis, a la izquierda de dato origen, el tipo al que se tiene que convertir:

```
char n=(char)b;
long l=(long)r
```

En el siguiente bloque de código te mostramos algunos ejemplos de conversiones válidas, tanto implícitas como explícitas:

```
char c='@';

int p=c; //correcto, se almacena el código

         //unicode del carácter

int num=3450;

byte r=(byte)num;  //correcto, se realiza un

                   //truncado del número

c=(char)34.5; //igual que en el caso anterior
```


Sin embargo, las siguientes conversiones no serían válidas y provocarían error de compilación:

```
byte s=1000; //el literal entero no cabe en byte, requiere

             //conversión explícita

boolean b=false;

int n=(int)b; //no son compatibles
```

TIPOS OBJETO

Como indicamos anteriormente, cualquier clase Java es un tipo objeto. Al igual que los tipos primitivos, se manejan a través de variables de su tipo (clase), la cual contendrá una referencia al objeto.

Un ejemplo dato de tipo objeto es la cadena de caracteres. Una cadena de caracteres es un objeto de tipo String. Aunque más adelante analizaremos el uso de la clase String, la siguiente instrucción nos muestra cómo crear un objeto de este tipo y almacenar su referencia en una variable:

```
String s=new String("hello");
```

Dado que String es un tipo de dato muy habitual, Java permite hacer la operación anterior de forma abreviada, asignando directamente el literal de cadena (texto entre comillas) a la variable:

```
String s="hello";
```

Es importante indicar que **NO se puede hacer conversión**, ni implícita ni explícita, **entre tipos primitivos y objetos**. Por ejemplo, el siguiente intento de conversión de texto a número provocaría un error de compilación:

```
String num="24";

int n=(int)num; //error de compilación
```

Los datos de tipo objeto no admiten las operaciones clásicas que se realizan con tipos primitivos, como las aritméticas, lógicas, etc. El objetivo de los datos de tipo objeto es proporcionar métodos (funciones) que el programador puede utilizar dentro del programa para realizar alguna tarea.

El uso de estos métodos o funciones se realiza utilizando la referencia al objeto, seguido del operador punto y el nombre del método:

```
s.length(); //llamada a método length() del objeto String
```

En el último capítulo del libro estudiaremos los principales métodos de los tipos de objetos que serán objetivo de examen, como String, StringBuilder, ArrayList, etc. Aunque antes se analizarán algunas de las características y comportamientos generales de los objetos. Comenzamos en el próximo apartado hablando del ciclo de vida de los objetos.

CICLO DE VIDA DE UN OBJETO

El conocimiento del ciclo de vida de un objeto, y más concretamente, saber determinar en qué momento un objeto es elegido para su destrucción, es otro de los objetivos del examen de certificación y debemos esperarnos alguna pregunta de examen sobre este aspecto.

Así pues, a lo largo de este breve apartado, vamos a conocer el proceso de creación y destrucción de un objeto, proceso que es independiente del tipo de objeto que se esté utilizando.

Creación de un objeto

Un objeto o instancia de clase se crea en Java utilizando el operador **new**, seguido del nombre de la clase. La llamada a new crea un objeto en memoria y devuelve una referencia a dicho objeto que será almacenada en una variable de su tipo:

```
Clase1 c=new Clase1();

String s=new String("hola");

Object ob=new Object();
```

CONSTRUCTORES

Los constructores son bloques de código que se ejecutan durante la creación de un objeto. Se definen dentro de la clase con el nombre de la misma:

```
class MiClase{

    public MiClase(){

          //código constructor

    }

}
```

El código definido dentro del constructor se ejecutará cada vez que se crea un objeto de la clase:

```
MiClase mc=new MiClase(); //se ejecuta el constructor
```

La misión del constructor es realizar algún tipo de inicialización de atributos. De hecho, los constructores pueden recibir parámetros cuyos valores pueden utilizarse para inicializar los atributos del objeto.

Una clase puede incluir varios constructores, pero deberán diferenciarse en el número y/o tipo de parámetros:

```
class MiClase{

    public MiClase(){
```

```
    }

    public MiClase(int p){

    }

    public MiClase(boolean t){

    }

}
```

Dependiendo de los argumentos que se pasen durante la creación del objeto se llamará a unos u otros constructores:

```
new MiClase();//llamada primer constructor

new MiClase(6);//llamada segundo constructor

new MiClase(true);//llamada tercer constructor
```

Destrucción de un objeto

Cuando se crea un objeto en Java, dicho objeto se sitúa dentro de la memoria gestionada por la Máquina Virtual Java, ocupando un determinado espacio.

La destrucción del objeto y la consiguiente liberación del espacio ocupado por el mismo son llevadas a cabo por un proceso perteneciente a la propia máquina virtual, y que es conocido como el **recolector de basura** o **Garbage Collector** (GC).

El momento en que el GC se pone en funcionamiento para realizar la liberación de la memoria no podemos determinarlo, pues eso depende de la implementación de la JVM y de la ocupación de la memoria en cada momento. Lo que sí tenemos que saber es que, cuando el GC se ejecuta, realizará la liberación de la memoria de todos los objetos que hayan sido marcados para la recolección y debemos ser capaces de determinar en qué momento ocurre dicho marcado.

MARCADO DE OBJETOS PARA RECOLECCIÓN

Y, nos preguntaremos, ¿cuándo se marca un objeto para recolección? Un objeto es marcado para recolección cuando deja de haber referencias al mismo dentro de

un programa. Dicho de otra manera, en el momento en que no hay ninguna variable apuntando al objeto, este es marcado como basura.

Veamos la definición del siguiente método:

```
1. public void metodo(){
2.  Object ob1=new Object(); //objeto1
3.  Object ob2=new Object(); //objeto2
4.  ob1=ob2; //objeto1 a recolección
5.  ob2=null; //objeto2 sigue apuntado por ob1
6. } //objeto2 a recolección
```

En las líneas 2 y 3 se crean dos objetos de la clase Object, referenciados por las variables ob1 y ob2. Podríamos representar de forma gráfica esta situación de la siguiente manera:

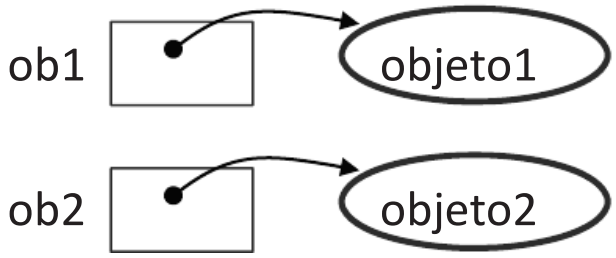


Fig. 2-7. Creación de dos objetos

En la línea 4, la variable ob2 se asigna a ob1, lo que significa que ambas variables contienen el mismo valor, que no es otro que la referencia al objeto2. Por tanto, el objeto1 deja de estar referenciado, por lo que es en este momento cuando es marcado para la recolección:

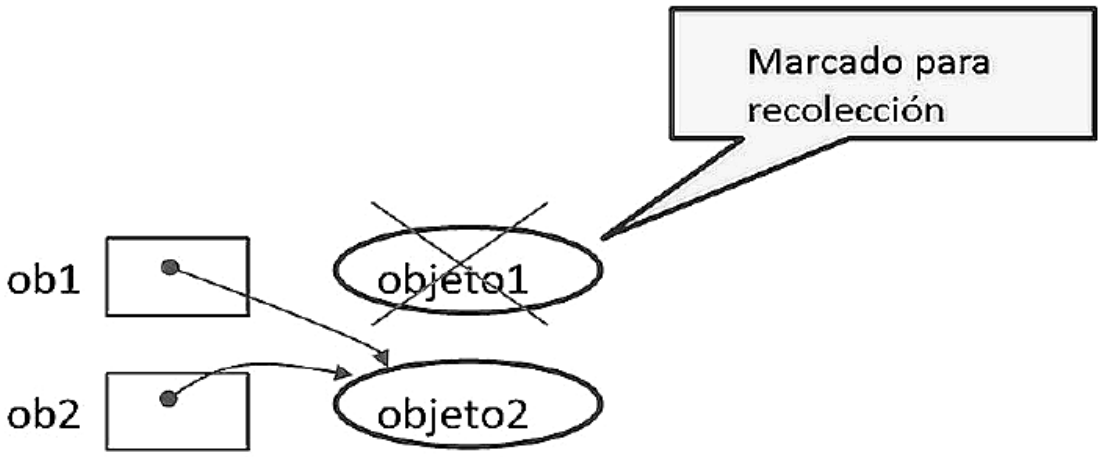


Fig. 2-8. Marcado de objeto para recolección

En la línea 5 se asigna el valor null a la variable ob2, lo que significa que esta variable ya no apunta a ningún objeto, pero como objeto2 sigue apuntado por ob1, aún no es marcado para recolección:

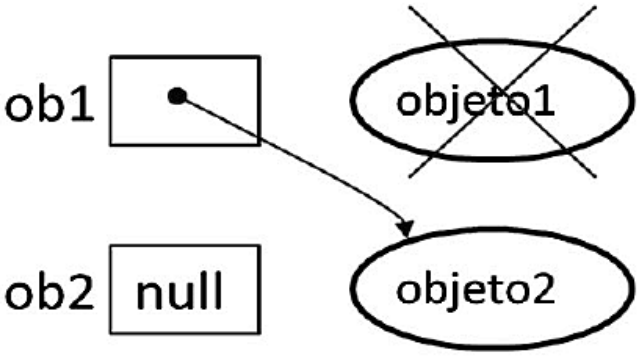


Fig. 2-9. Eliminación de referencia a objeto

Al finalizar el método en la línea 6, todas las variables locales son destruidas, es en ese momento cuando objeto2 también pasa a ser marcado para recolección.

Es importante destacar que cuando un objeto es marcado para recolección, **ya no hay forma de recuperarlo**.

MÉTODO FINALIZE()

Todas las clases Java disponen de un método llamado *finalize()*. Este método forma parte del ciclo de vida de un objeto, ya que es llamado por la JVM y no explícitamente desde código. La JVM llama al método *finalize()* de un objeto **justo antes de que el recolector lo elimine de memoria**. Pudiera ser que el objeto nunca sea eliminado, bien porque no sea marcado para recolección, o bien porque el GC no se haya activado desde que el objeto pasó a recolección, en este caso el método *finalize()* no será ejecutado.

De cara a posibles preguntas de examen es importante saber que *finalize()* puede ser llamado una o ninguna vez, pero **nunca más de una**.

CLASES DE ENVOLTORIO

El paquete java.lang cuenta con una serie de clases que permiten encapsular datos primitivos en forma de objetos. Esto permitirá realizar ciertas operaciones con números, boolean o caracteres, que solo pueden llevarse a cabo con objetos.

Creación de objetos de envoltorio

Para cada tipo primitivo existe una clase de envoltorio, con lo que tenemos ocho clases en total: Byte, Short, Integer, Long, Character, Float, Double y Boolean.

Todas las clases disponen de un constructor que permite crear un objeto de envoltorio a partir del tipo primitivo:

```
Integer ent=new Integer(39); //objeto Integer a
                             //partir del int 39
Character car=new Character('@');
Boolean b=new Boolean(false);
Double db=new Double(4.6);
```

A excepción de Character, todas disponen también de un constructor que permite crear un objeto a partir de la representación en forma de cadena del tipo primitivo:

```
Integer integ=new Integer("800");
Boolean bol=new Boolean("true");
```

Si, posteriormente, queremos recuperar el tipo primitivo a partir del objeto, todas estas clases disponen de un método xxxValue(), donde xxx sería el nombre del tipo primitivo. Por ejemplo, la clase Integer dispone del método intValue(), mientras que Character tiene charValue():

```
char carac=car.charValue();

int num=integ.intValue();
```

Autoboxing/unboxing

Desde la versión Java 5, la creación de objetos de envoltorio a partir de tipos primitivos puede realizarse directamente asignando el literal primitivo a la variable objeto, sin necesidad de utilizar el operador new:

```
Integer integ=49;

Character car='@';

Boolean bol=true;
```

Esto es lo que se conoce como **autoboxing**. Por otro lado, se puede recuperar el tipo primitivo asignando directamente la variable objeto a la variable primitiva, operación que es conocida como **unboxing**:

```
int n=integ; //unboxing

Integer k=30;//autoboxing

k++; //unboxing más autoboxing
```

Como vemos, gracias al autoboxing/unboxing, podemos trabajar indistintamente con tipos objetos y tipos primitivos. No obstante, debemos saber que si vamos a realizar operaciones aritméticas con números, es más eficiente utilizar tipos primitivos en lugar de objetos, pues el estar continuamente aplicando autoboxing/unboxing supone un coste en términos de rendimiento de la aplicación.

Inmutabilidad de objetos de envoltorio

Los objetos de las clases de envoltorio son inmutables, lo que significa que no se pueden modificar. Veamos el siguiente ejemplo:

```
Integer ent=200; //autoboxing

ent=ent+100; //genera un nuevo objeto, unboxing + autoboxing
```

Como se indica en el comentario, no se ha modificado el objeto numérico apuntado por la variable ent, Java aplica unboxing para recuperar el valor del objeto y calcular la suma, para después utilizar autoboxing y **generar un nuevo objeto** con el valor resultante, objeto que pasa a ser apuntado por la variable ent.

La siguiente figura nos muestra la situación de forma gráfica:

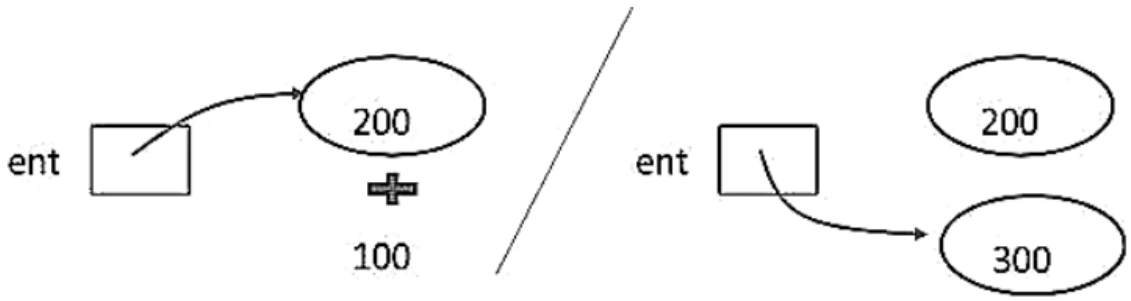


Fig. 2-10. Inmutabilidad de objetos de envoltorio

PREGUNTAS TIPO EXAMEN

Pregunta 1

Given the code fragment:

```
3. public static void main(String [] args){
4.      int ivar=100;
5.      float fvar=23.4f;
6.      double dvar=20;
7.      ivar=fvar;
8.      fvar=ivar;
9.      dvar=fvar;
10.     fvar=dvar;
11.     dvar=ivar;
12.     ivar=dvar;
13. }
```

Which three lines fails to compile?

A. Line 7

B. Line 8

C. Line 9

D. Line 10

E. Line 11

F. Line 12

Pregunta 2

Which of the following compiles with no errors? (choose 4)

A. int a = 2.7;

B. byte n=(byte) 4546.9;

C. int c=0b110_100;

E. double t =_6.4;

F. float g = 23;

G. short k =(short)45L;

H. boolean n =(boolean)1;

I. char vv=300000;

Pregunta 3

Given:

```
public class Marca{
    int num;
    public static void save(Marca obj4){
            obj4.num+=10;
    }
    public static void main(String[] args){
            Marca obj1=new Marca();
            Marca obj2=obj1;
            Marca obj3=null
```

```
            obj2.num=60;
            save(obj2);
        }
    }
```

How many Marca instances are created at runtime?

A. 1

B. 2

C. 3

D. 4

Pregunta 4

Given:

```
1. public class Test{
2.     public static void main(String[] args){
3.         Integer uno, dos;
4.         uno=new Integer (10);
5.         dos=new Integer(20);
6.         uno=dos;
7.         uno=null;
8.     }
9. }
```

When the objects created in lines 10 and 20 will be elegible for Garbage collection ?

A. Both in line 8

B. Both in line 9

C. Both in line 7

D. 10 in line 6 and 20 in line 7

E. 10 in line 6 and 20 in line 8

Pregunta 5

Given the following code:

1. Integer k=5;
2. int p=10, s;
3. k=k+p;
4. s=k;
5. System.out.println(s);

What is the result?

A. 15

B. 5

C. Compilation error in line 3

D. Compilation error in line 4

Pregunta 6

G Given:

```
public class Tester{
    static double res;
    int a=1, b=3;
    public static void main(String[] args){
        double a, b, c;   //line 1
        if(res==0){     //line 2
            a=1.5;
            b=2;
            c=1;
        }
        res=a*b*c;    //line 3
        System.out.println("result "+res);
    }
}
```

What is the result?

A. result 3.0

B. result 0.0

C. Compilation fails at line 1

D. Compilation fails at line 2

E. Compilation fails at line 3

SOLUCIONES

1. A, D y F. La A es incorrecta porque no se puede convertir implícitamente un float en un int. La D es incorrecta porque la conversión de double a float tampoco se puede hacer de forma implícita, como tampoco se puede convertir implícitamente de double a int y por ello la F es también incorrecta.

2. B, C, F y G. La B y la G son correctas porque de forma explícita cualquier tipo se puede convertir en otro, excepto boolean. La C es correcta porque la representación del literal entero binario es correcta. La F también es correcta porque un entero se puede convertir implícitamente en un float.

3. A. Solamente se crea un objeto. las variables obj1 y obj2 apuntan ambas al mismo objeto, el creado en la primera línea del main. Con la llamada a save(), una cuarta variable obj4 apunta también al objeto.

4. E. El objeto Integer creado en la línea 4 es marcado para recolección en la línea 6 porque la única variable que lo apuntaba (uno), pasa ahora a apuntar al objeto creado en línea 5. El objeto creado en línea 5 es apuntado por la variable dos hasta que esta sale de ámbito, es decir, al finalizar el método en línea 8.

5. A. La instrucción k=k+p, provoca un unboxing del objeto Integer apuntado por k para sumar su valor a p, realizando después un autoboxing para asignar el resultado a k. Al asignar s=k, se produce de nuevo un unboxing para extraer el resultado, que es 15.

6. E. Las variables a, b y c a las que se refiere la instrucción de la línea 3 son las variables locales del main, como estas variables no están inicializadas de forma explícita, el compilador determina que en caso de no cumplirse la condición del if estas variables no tendrían valor, por lo que no puede operarse con ellas.

3 Operadores y estructuras de decisión

OPERADORES

Los operadores son esenciales en cualquier lenguaje de programación, pues son los símbolos que utilizamos para realizar operaciones con los datos de un programa. El juego de operadores Java es bastante extenso, así que vamos a revisar algunas cuestiones importantes sobre su funcionamiento que debemos tener presentes durante las preguntas de examen.

Los operadores Java podemos clasificarlos en los siguientes grupos:

- Aritméticos
- Asignación
- Condicionales
- Lógicos
- Otros

Operadores aritméticos

Como se desprende del nombre, este tipo de operadores nos van a permitir realizar operaciones aritméticas en un programa con los datos de tipo numérico, que incluyen además de los tipos primitivos (a excepción de boolean) a las clases de envoltorio.

Los operadores aritméticos existentes en Java son: +,-,*,/,%,++,--

OPERADORES SIMPLES

Entendemos por operadores aritméticos simples a los cinco primeros (suma, resta, multiplicación, división y resto).

Sobre ellos debemos indicar que cuando se utilizan con números enteros, independientemente de su tipo, el resultado siempre es int, salvo que alguno de los operandos sea long, en cuyo caso el resultado será también long:

```
long l1=10;

int n=l1*2;//error de compilación, el resultado es long
```

Fijémonos en este otro ejemplo:

```
byte a=2, b=5;

byte n=a+b; //error de compilación
```

La instrucción anterior produce un error de compilación porque la suma de a con b da como resultado un int, aunque ambos operandos sean de tipo byte. Para que fuera correcto, se debería hacer un casting o conversión del resultado a byte:

```
byte n=(byte)(a+b);
```

Observemos este otro ejemplo:

```
int x=7, y=2;

int z=x/y;
```

El resultado de esta operación es 3, pues al tratarse de **operandos enteros el resultado de la división será también entero**, despreciándose la cifra decimal.

En el caso de operaciones aritméticas con operandos decimales, el resultado será siempre el del tipo mayor:

```
float f1=2.3f;

double d1=3.7;

f1=f1+d1;// error de compilación, el resultado es double
```

Por último, un ejemplo sobre el operador %, que se utiliza para calcular el resto de una división:

```
int res=10%3; //el resultado es 1
```

OPERADORES INCREMENTO Y DECREMENTO

Estos operadores merecen atención expresa, pues cuando aparecen involucrados en una expresión con otras operaciones el resultado puede ser sorprendente. Es muy probable que aparezcan estos operadores en más de una pregunta de examen, por lo que debemos analizar con detalle su funcionamiento.

Funcionamiento

En principio, indicar que se trata de operadores unarios, es decir, se aplican sobre una única variable. Además, solo es aplicable con datos de tipo entero.

Si suponemos que "a" es una variable entera, la siguiente instrucción:

```
a++;
```

sería equivalente a:

```
a=(int)(a+1);
```

Es decir, el operador lo que hace es sumar uno a la variable y depositar el resultado en la misma variable, previa conversión de este resultado al tipo de la variable.

Si la variable fuera de otro tipo entero, por ejemplo byte, la operación a++ equivaldría a:

```
a=(byte)(a+1);
```

Como vemos, **siempre se convierte al tipo de la variable**. Es importante diferenciar la operación incremento con el operador ++ del incremento manual. Observemos el siguiente bloque de instrucciones:

```
byte b=5;

b=b+1;//error de compilación

b++;//correcto
```

Como vemos, la primera instrucción no compilaría, ya que el resultado de la suma es int, sin embargo, el operador incremento funcionaría perfectamente con la variable byte, y esto es debido a que se realiza la conversión antes de asignar el resultado.

El operador decremento (--) funciona exactamente igual que ++, solo que restando una unidad a la variable.

Posición del operador

Los operadores incremento y decremento pueden utilizarse delante o detrás de la variable, por lo que la instrucción:

```
a++;
```

produce el mismo resultado que

```
++a;
```

Sin embargo, la cosa cambia cuando estos operadores aparecen involucrados en una expresión con otros operadores. Observemos el siguiente código:

```
int a=5,b=5;

int x=a++;

int y=++b;
```

¿Cuál sería el valor de las variables x e y después de ejecutarse estas tres instrucciones? La respuesta es x vale 5 e y vale 6. En el primer caso, como el operador aparece después de la variable, primero se asigna el valor de esta a x y luego se incrementa, mientras que en el segundo caso, al poner el operador delante de la variable, debe realizarse el incremento antes de poder asignarlo a y. Lógicamente, ambas variables, a y b, tienen el valor 6 tras la ejecución de este código.

La conclusión es que, cuando los operadores ++ o -- se colocan **después** de la variable y esta variable está involucrada en una expresión, **la operación de incremento o decremento es la última en efectuarse**.

Fijémonos en este otro ejemplo:

```
int a=3,b=4;

boolean res=a++==b;
```

El resultado de la operación es *false*, puesto que primero se compara el valor actual de a (3) con b antes de realizar el incremento.

Operadores de asignación

Además del operador clásico de asignación, =, que se emplea para asignar el resultado de una expresión a una variable, existen otros operadores que realizan una operación aritmética antes de la asignación, estos operadores son: +=,-=,*=,/= y %=.

Por ejemplo, dadas las siguientes instrucciones:

```
int a=5;

a=a+3;
```

La última instrucción podría haberse escrito también de esta otra forma:

```
a+=3.
```

Al igual que sucede con los operadores de incremento y decremento, los operadores de asignación anteriores incluyen un *casting* o conversión del resultado de la operación al tipo de la variable. Por ejemplo:

```
byte b=2;

b+=7;
```

La instrucción anterior equivale realmente a:

```
b=(byte)(b+7);
```

Si hiciéramos simplemente:

```
b=b+7;
```

se produciría, como ya hemos visto antes, un error de compilación.

Operadores condicionales

Se emplean en instrucciones de control de flujo. Evalúan una condición y dan como resultado un boolean (true o false).

Estos operadores serían: <,>,<=,>=, == y !=

Salvo el operador de igualdad (==), que puede utilizarse con objetos, los demás solo pueden ser aplicados sobre tipos primitivos y compatibles entre ellos:

```
int a=3;

double c=9.5, n=3.0;

boolean x=false;

System.out.println(a>c);//correcto, resultado false

System.out.println(a==n);//correcto, resultado true

System.out.println(a!=x);//error de compilación
```

La última instrucción es incorrecta porque, como sabemos, el tipo boolean es incompatible con el resto de primitivos.

Operadores lógicos

Realizan operaciones sobre operandos de tipo boolean, dando como resultado también un boolean.

Tenemos tres operadores lógicos en Java: &&(and), ||(or) y !(not). Aquí tenemos algunos ejemplos de uso:

```
int a=3;

int c=9;

int n=0;

System.out.println(a>n && a<c); //true, ambos operandos
                                //son true

System.out.println(a<n && a<c); //false, un operando
                                // es false

System.out.println(a<n || a<c); //true, un operando es true

System.out.println(!(n==0)); //false
```

Una característica de los operadores && y || que hay que tener muy presente de cara a las preguntas de examen es que **ambos funcionan en modo cortocircuito**, lo que significa que si la evaluación del primer operando determina el resultado de la operación, no se evalúa el segundo operando. Observemos el siguiente código en el que se utilizan las variables a, c y n del ejemplo anterior:

```
boolean b=(a<n && a<++c);

//muestra 9, la segunda condición no llega a evaluarse

System.out.println(c);
```

Como se indica en el comentario, al ser *false* la primera condición, el resultado es directamente *false*, por lo que la segunda condición no llega a evaluarse y, por tanto, la variable c no se incrementa, mostrándose el valor inicial que es 9.

Otros operadores

Además de los operadores anteriores, tenemos otros operadores que no entrarían en ninguna clasificación:

- **new**. Es el operador utilizado para crear objetos en Java. A continuación del operador se indica la clase de la que se quiere crear el objeto, devolviéndose el objeto creado. En el siguiente ejemplo se crea un objeto de la clase Object que es almacenado en una variable de ese tipo:

  ```
  Object ob=new Object();
  ```

- **instanceof**. Comprueba si un objeto es de un determinado tipo. Es un operador binario, el operando de la izquierda es la variable que apunta al objeto que se quiere comprobar, y el de la derecha el nombre de la clase. El resultado es boolean:

  ```
  //muestra true

  System.out.println(ob instanceof Object);
  ```

  Ya veremos algún otro ejemplo de uso cuando veamos el polimorfismo.

Otro operador muy interesante es el operador ternario, que analizaremos en el siguiente apartado.

INSTRUCCIÓN IF Y OPERADOR TERNARIO

El funcionamiento de la estructura de decisión alternativa, tanto en su versión de instrucción *if*, como en operador ternario, es otro de los objetivos de examen que forman parte de esta sección. Analicemos cada uno de ellos por separado.

Instrucción if

La instrucción *if* comprueba una condición de tipo *boolean* y ejecuta un bloque de sentencias si el resultado de esta es *true*. Opcionalmente, se puede indicar bloque *else* con las tareas a realizar si la condición no se cumple. El formato es el siguiente:

```
if(condicion){

   //sentencias

}else{

   //sentencias

}
```

Por ejemplo, el siguiente bloque nos indica si el número almacenado en la variable "a" es par o impar:

```
if(a%2==0){

    System.out.println("par");

}else{

     System.out.println("impar");

}
```

Como hemos dicho, el bloque else es opcional, este otro bloque de código incrementa la variable si su valor es mayor que 5:

```
if(a>5){

    a++;

}
```

Es muy importante tener en cuenta que la **condición evaluada por if debe dar como resultado un *boolean***. Hay que prestar atención a posibles preguntas de examen del tipo siguiente:

```
int a=3;

if(a){ //error compilación

    a++;

}
```

El código anterior **no compila** debido a que la condición evaluada por *if* no es *boolean*, sino entera.

También hay que prestar atención a las llaves, pues estas solo son obligatorias si el bloque está formado por más de una sentencia. Fijémonos en el siguiente código:

```
int n=3;

if(n<3)

    n++;

    n--;
```

¿Cuánto valdría la variable n después de ejecutar el código anterior? La respuesta es 2, pues el no cumplir la condición no evaluaría el incremento, pero sí la instrucción n--, **pues está fuera del bloque if** a pesar de que se haya escrito tabulada.

Operador ternario

El operador ternario representa una forma abreviada de instrucción *if*, utilizándose en aquellos casos en que el bloque de sentencias, tanto del *if* como del *else*, tienen que devolver un resultado.

Su formato es el siguiente:

variable=condición?valor_si:valor_no

¿Cómo funciona este operador? Pues bien, evalúa la condición de tipo *boolean* que se indica a la izquierda de la interrogación, si el resultado es verdadero, se devuelve el valor indicado después de la interrogación, si no, devolverá el valor a continuación de los dos puntos. El valor devuelto se almacena en una variable, aunque puede ser utilizado para otro fin, como imprimirlo, pasarlo como parámetro a un método, etc.

En el siguiente ejemplo, involucramos también el operador de decremento en una de las expresiones para el cálculo del resultado:

```
int a=3, b=5, c;

c=(a>b)?a*a:b--;

System.out.println(c);
```

Al resultar falsa la condición se evalúa la expresión que aparece después de los dos puntos, pero recordemos que al utilizar el operador decremento después de la variable primero se asigna el valor actual de "b" a "c" antes de incrementarse, por lo que el resultado que se imprimirá será 5.

IGUALDAD DE OBJETOS

Dentro de la sección de operadores y estructuras de decisión, otro de los objetivos de examen es la comparación entre objetos en Java. Como dijimos al explicar los operadores condicionales, el operador de igualdad == puede utilizarse también con objetos. A lo largo de este apartado, vamos a ver las diferencias existentes entre comparar objetos con == y el empleo del método *equals()*, presente en todas las clases Java.

Uso del operador == con objetos

Podemos utilizar el operador == para comparar variables de tipo objeto, pero hay que tener en cuenta que lo que **estamos comparando son las referencias almacenadas en esas variables, no los objetos**. Esto lo podemos ver claramente en el siguiente código de ejemplo en el que comparamos dos variables apuntando a objetos de tipo Integer:

```
Integer n1=new Integer (10);

Integer n2=new Integer (10);

//el resultado es falso

if(n1==n2){

}
```

Como se indica en el comentario, el resultado de la operación **n1==n2 es falso**. El motivo es que no estamos comparando los objetos, en este caso numéricos, sino las referencias almacenadas en las variables, y como ambas variables apuntan a objetos diferentes, las referencias serán diferentes:

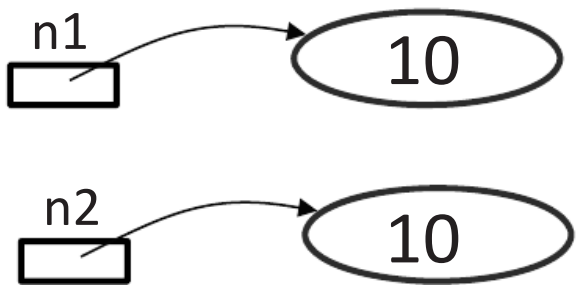


Fig. 3-1. Objetos diferentes, referencias distintas

Otro punto importante a tener en cuenta sobre el operador == es que **no permite comparar objetos de diferente tipo**. Por ejemplo, el siguiente bloque de código no compilaría, provocaría un error de compilación en la instrucción de comparación:

```
Integer n1=new Integer(10);

Double d1=new Double(10.0);

if(d1==n1){ //error de compilación

}
```

Igualdad de cadenas de caracteres

Una cadena de caracteres es un objeto de la clase String. En el último capítulo, hablaremos de los métodos que nos proporciona esta clase para manipular textos, de momento, nos centraremos en analizar las características de los objetos String y, muy especialmente, lo concerniente a la igualdad de este tipo de objetos.

De manera formal, para crear un objeto String procederemos como con cualquier clase Java, es decir, utilizando el operador new:

```
String s1=new String("Java");
```

La variable s1, como ya sabemos, no contiene el objeto, sino una referencia al mismo, por lo que aplicar el operador == con variables de tipo String tiene el efecto que comentamos anteriormente, solo dará como resultado true si ambas variables apuntan al mismo objeto:

```
String s2=new String("Java");

String s3=s1;

System.out.println(s1==s2);//false, objetos diferentes

System.out.println(s1==s3);//true, mismo objeto
```

EL POOL DE CADENAS DE CARACTERES

Según el código de ejemplo indicado anteriormente, la comparación s1==s2 da como resultado *false* por tratarse de objetos diferentes, pero fijémonos ahora en este otro ejemplo:

```
String s1="Java";

String s2="Java";
```

Dado que trabajar con cadenas de caracteres es algo muy habitual en un programa, Java permite crear los objetos String asignando directamente el literal a la variable. Sin embargo, si ahora ejecutamos la siguiente instrucción:

```
System.out.println(s1==s2);//muestra true !!
```

Como se indica en el comentario, el resultado de la comparación será *true*. ¿Cómo es posible esto? ¿No habíamos quedado en que el operador == compara referencias y no los objetos?

Las respuestas a estas preguntas están en la manera en la que internamente Java maneja los literales de texto. Y es que, de cara a optimizar el uso de la memoria, la **JVM utiliza un pool de literales de cadenas de caracteres** donde guarda este tipo de objetos. Entonces, al asignar un literal de cadena a una variable, no se crea un nuevo objeto directamente, **primero se comprueba si existe en el pool y si es así se devuelve una referencia al objeto existente, pero si no existe, se crea y se graba en el pool**.

Llevado al ejemplo anterior, esto significa que durante la primera asignación se creó el objeto "Java" en el pool, pero al asignar el mismo literal a la variable s2, no se creó un nuevo objeto sino que se asignó una referencia al ya existente:

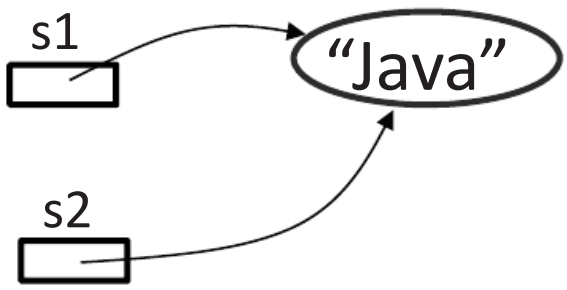


Fig. 3-2. Las dos variables apuntan al mismo objeto

EL MÉTODO EQUALS()

Para comparar dos objetos de tipo String debemos emplear el método *equals()*. Este método lo tienen todas las clases, pues es heredado de Object, sin embargo, no todas lo redefinen para adaptarlo a las características de su tipo de objetos. String sí lo tiene redefinido, de modo que **devolverá *true* si los objetos apuntados por las variables contienen el mismo texto**:

```
String n1=new String("cadena");

String n2=new String("cadena");

//el resultado es verdadero

if(n1.equals(n2)){

}
```

El método *equals()* hace distinción entre mayúsculas y minúsculas, por lo que si un texto es "Java" y el otro es "java", al compararlos el resultado será falso.

Si no queremos que se tenga en cuenta la distinción entre mayúsculas y minúsculas, utilizaremos el método *equalsIgnoreCase():*

```
String n1=new String("JAVA");

String n2=new String("java");

//el resultado es verdadero

if(n1.equalsIgnoreCase(n2)){

}
```

CONCATENACIÓN DE CADENAS DE CARACTERES

El operador + también puede utilizarse para concatenar (unir) cadenas de caracteres:

```
String s1="Java ";

String s2="estandar";

String s3=s1+s2; //el resultado es "Java estándar"
```

Es posible concatenar un String con un tipo primitivo. En este caso, el tipo primitivo es convertido implícitamente a String:

```
String s1="Java ";

int v=8;

String r=s1+v; //graba la cadena "Java 8"
```

Las cadenas de caracteres String son inmutables, lo que significa que no se pueden modificar. Si concatenamos un String con otro o con un tipo primitivo, no se está modificando el String original, se está generando uno nuevo resultado de la concatenación:

```
String s1="Hola ";

s1=s1+"adios";
```

En este caso, la variable s1 pasa a apuntar al objeto resultante de la concatenación de "hola" y "adios", el objeto original "hola" deja de estar referenciado y, por tanto, marcado para recolección:

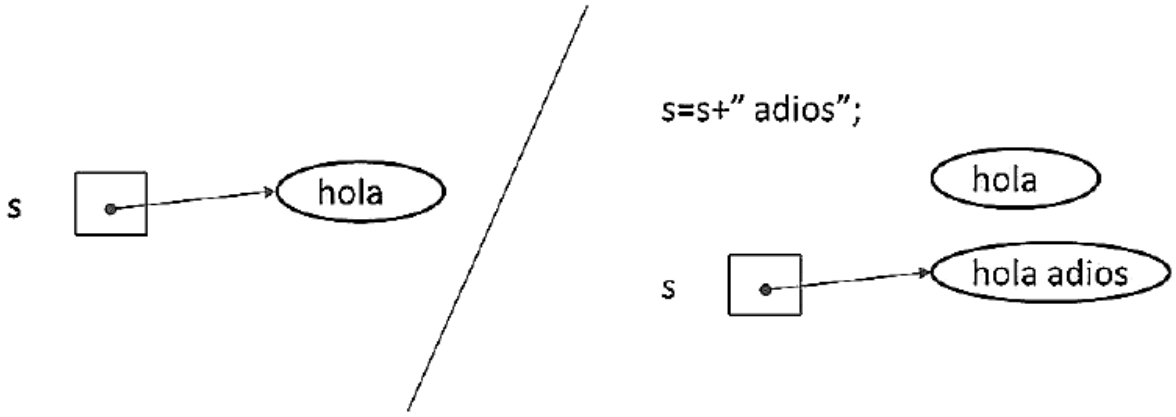


Fig. 3-3. El objeto original no se modifica al concatenar

Igualdad de objetos de envoltorio

Ya vimos durante el estudio del operador == que dos objetos de envoltorio que encapsulen el mismo valor dan como resultado *false* si se comparan con ==, pero las clases de envoltorio también disponen del método *equals()* para comparar los valores encapsulados en el objeto:

```
Integer n1=new Integer(20);

Integer n2=new Integer(20);

System.out.println(n1==n2); //false

System.out.println(n1.equals(n2)); //true
```

Con los tipos de envoltorio tenemos que tener en cuenta que, al igual que sucede con String, si asignamos directamente el mismo literal a dos variables, la comparación con == dará como resultado *true*:

```
Integer n1=20;

Integer n2=20;

System.out.println(n1==n2);// resultado true
```

Como ocurre con String, Java mantiene un pool de objetos literales numéricos, de modo que al asignar por segunda vez el mismo literal no se crea un nuevo objeto, sino que se reutiliza el ya existente.

Igualdad de objetos StringBuilder

Las cadenas de caracteres de tipo String son objetos inmutables, es decir, no se pueden modificar. Por el contrario, un objeto StringBuilder representa una cadena de caracteres **mutable**, es decir, que puede ser modificada. Para crear un objeto StringBuilder debemos utilizar el operador new, no es posible utilizar, como en String, la forma abreviada de asignación de literales de cadena a la variable:

```
StringBuilder sb1=new StringBuilder("Java");
```

En un capítulo posterior tendremos oportunidad de analizar los métodos de StringBuilder, mediante los cuales podemos modificar la cadena de caracteres.

De momento, nos vamos a centrar en la igualdad de objetos StringBuilder. Como ya sabemos, el operador == compara referencias a objetos, por lo que al aplicarlo a dos variables que apunten a objetos diferentes dará como resultado *false*, aunque el valor de cada objeto sea el mismo. Esto es también aplicable a StringBuilder:

```
StringBuilder sb1=new StringBuilder("Java");

StringBuilder sb2=new StringBuilder("Java");

System.out.println(sb1==sb2); //false
```

¿Podríamos aplicar el método *equals()* para comparar objetos StringBuilder? El método *equals()* lo tienen todas las clases Java, también StringBuilder, el problema es que esta clase **no lo ha redefinido**, utiliza la versión original del método existente en

la clase Object, en la que en método **devolverá *true* solo si las dos variables sobre las que se aplica apuntan al mismo objeto**, es decir, hace lo mismo que ==.

Por tanto, si aplicamos *equals()* a las variables anteriores sb1 y sb2, a pesar de que los objetos apuntados por ambas contienen el mismo texto, **el resultado será *false***:

```
StringBuilder sb1=new StringBuilder("Java");

StringBuilder sb2=new StringBuilder("Java");

System.out.println(sb1.equals(sb2)); //false
```

¿Cómo podemos comparar entonces los textos referenciados por dos objetos StringBuilder? La única forma de hacerlo es convertir StringBuilder en String y poder utilizar así el método *equals()* de esta clase, esto se puede hacer a través del método *toString()* de StringBuilder:

```
String s1=sb1.toString();

String s2=sb2.toString();

System.out.println(s1.equals(s2));//true
```

LA INSTRUCCIÓN SWITCH

La instrucción *switch* es una instrucción de tipo alternativa múltiple que, en función del resultado de una expresión, puede realizar diferentes tareas. Esta instrucción tiene muchas peculiaridades y casos especiales, lo que le hace objeto de más de una pregunta de examen. Por ese motivo, vamos a analizar detenidamente su funcionamiento y los casos que pueden plantearse.

Sintaxis

El formato de la instrucción switch es el que se indica a continuación:

```
switch(expresion){
   case valor1:
     //sentencias
```

```
        break;
      case valor2:
        //sentencias
        break;
      :
      default:
        //sentencias
}
```

La expresión debe dar como resultado un valor *int* o convertible implícitamente a *int* (*byte*, *short* y *char*). Si el resultado de la expresión coincide con uno de los valores indicados en los *case*, ejecutará el bloque correspondiente de sentencias, sino, entrará en el bloque *default* que es opcional.

El siguiente código mostrará el mensaje "cerca" al ser ejecutado:

```
int a=2;
switch(a*2){
    case 0:
       System.out.println("nada");
       break;
    case 4:
       System.out.println("cerca");
       break;
    case 8:
       System.out.println("acierto");
       break;
```

```
    default:

        System.out.println("error");

}
```

La instrucción *break* al final de cada bloque *case* es opcional, pero si no se indica, el programa entrará en el siguiente bloque. Por ejemplo, el siguiente código mostrará "Es 10 sin valor":

```
int a=10;

switch(a){

    case 10:

       System.out.print("Es 10");

    default:

        System.out.println(" sin valor");

}
```

Como vemos, el programa entra en el primer *case*, pero al no haber un *break* al final del mismo, a continuación pasaría a ejecutar el siguiente bloque, en este caso, el *default*.

Valores de los case

Dado que la expresión evaluada por *switch* tiene que devolver un resultado *int*, los posibles valores del case también deben ser *int* o inferior. Eso sí, deben ser literales o constantes, **no se admiten variables**. En el siguiente código puedes ver ejemplos de valores válidos e inválidos para case:

```
int p=5;

final int k=30;

int n=3;
```

```
switch(p){
    case 10: //ok, es un literal int
    case k:  //ok, es una constante int
    case p: //error de compilación, no es una constante
    case '@': //ok, char convertible implícitamente a int
}
```

El bloque default

Como indicamos anteriormente, el bloque *default* es opcional, pero además, puede aparecer en cualquier parte, no necesariamente al final:

```
int p=5;
switch(p){
    case 10:
     System.out.println("Es 10");
    default:
          System.out.println("Default");
    case 2:
          System.out.println("Es 2");
}
```

La ejecución del código anterior mostraría:

Default

Es 2

Observa cómo, a pesar de no coincidir con el valor de case 2, ejecuta estas sentencias al no haber una instrucción *break* en el bloque *default*.

Switch con cadenas String

Desde la versión Java 7 es posible evaluar también en un *switch* expresiones cuyo resultado sea una cadena de caracteres de tipo String. Los valores de los *case* deberán ser literales o constantes String:

```
String data="prueba";

final String s="hello";

switch(data){

    case "uno": //correcto, es un literal

    case s:  //correcto, es una constante

    case 10: //error de compilación

}
```

Si observamos el último *case*, vemos un error de compilación al indicar un literal de tipo int. Y es que, **si la expresión es de tipo String, en los case solo se permitirán valores de tipo String**. De igual forma, **si la expresión de un switch es int, no se permitirán valores de tipo String en los case**.

PREGUNTAS TIPO EXAMEN

Pregunta 1

Given:

```
System.out.println("6+3="+2+7);
System.out.println("6+3="+(2+7));
```

What is the result?

A. 6+3=27

6+3=27

B. 6+3=2+7

6+3=9

C. 9=9

9=9

D. 6+3=27

6+3=9

Pregunta 2

Given the following:

```
1. public class Test {
2.   static int i;
3.   public static void main (String []args) {
4.         int a = 2, b = i+1;
5.         if ((a++>++b) && (++a>5)) {
6.               a +=b;
```

```
7.          }
8.    }
9. }
```

What is the final value of a?

A. 1

B. 2

C. 3

D. 4

E. 5

Pregunta 3

Given the code fragment:

```
StringBuilder sb1=new StringBuilder("hello");
String str1=sb1.toString();
//insert code here
System.out.println(str1==str2);
```

Which code fragment, when inserted at "insert code here", enables the code to print true?

A. String str2 = str1;

B. String str2 = new String (str1);

C. String str2 = sb1. toString ();

D. String str2 = "hello";

Pregunta 4

Given the following:

```
class Test{
        public static void main(String args[]){
                String arg="hello";
                change(arg);
                System.out.println(arg);
        }
        static void change(String s){
                s=s+" bye";
        }
}
```

What is the result?

A. hello

B. hello bye

C. compilations fails

D. exception

Pregunta 5

Given the code fragment

```
public class Test{
        public static void main(String[] args){
```

```
                    StringBuilder sb=new StringBuilder(5);
                    String s="";
                    if(sb.equals(s)){
                        System.out.println("option A");
                    } else if(sb.toString().equals(s.toString())){
                        System.out.println("option B");
                    }else{
                        System.out.println("option C");
                    }
                }
            }
```

What is the result?

A. option A

B. option B

C. option C

D. NullPointerException is thrown at runtime

Pregunta 6

Given the following:

```
public class Test {

    public static void main(String[] args) {
        int x=100;
        int a=x++;
        int b=++x;
        int c=x++;
        int d=(a>b)?(a<c)?a:(b<c)?b:c:b;
```

```
            System.out.println(d);
        }

    }
```

What is the result?

A. 100

B. 102

C. 101

D. 103

E. Compilation will fail.

Pregunta 7

```
1.  class Test{
2.      public static void main(String args[]){
3.          final int j;
4.          j=2;
5.           int x= 0;
6.
7.          switch(x){
8.              case 0: {System.out.print("A");}
9.              case 1: System.out.print("B"); break;
10.             case j: System.out.print("C");
11.          }
12.     }
```

13. }

What is true?

A. The output could be A

B. The output could be AB

C. The output could be ABC

D. Compilation fails.

E. There could be no output

Pregunta 8

Given the code fragment

```
public class Test{
    public static void main(String[] args){
        //line 1
        switch(x){
            case 1:
                System.out.println("One");
                break;
            case 2:
                System.out.println("Two");
                break;
        }
    }
}
```

Which three code fragments can be independently inserted at line 1 to enable the code to print one?

A. byte x = 1;

B. short x = 1;

C. String x = "1";

D. long x = 1;

E. double x = 1;

F. Integer x = new Integer ("1");

SOLUCIONES

1. D. En la primera instrucción, al ejecutar las operaciones de izquierda a derecha, primero se concatena "6+3=" con el número 2, generándose una nueva cadena "6+3=2". Esta cadena se concatena después con el número 7, quedando finalmente el texto "6+3=2+7". En la segunda instrucción, al aparecer la suma de los dos números entre paréntesis, esta será la operación que se procesa primero, después, el número resultante que es 9 se concatena con "6+3=".

2. C. La operación a++>++b realiza la comparación:2>2, pues la variable *a* se incrementa después de la comparación. Al resultar falsa, el otro operando de la expresión lógica no se evalúa, por lo que ni se vuelve a incrementar *a* ni se procesa la instrucción a+=b. Por tanto, el valor final de *a* es 3.

3. A. Para que se muestre *true*, ambas variables str1 y str2 deben estar apuntando al mismo objeto. La instrucción que se indica en la respuesta A es la que cumple esta condición.

4. A. Aunque se pasa una referencia al objeto "hello" en la llamada a change(), como String es inmutable la concatenación que se realiza en este método genera un nuevo objeto, pero el objeto "hello" apuntado por la variable arg no cambia, sigue siendo el mismo.

5. B. Las variables s y sb apuntan a objetos diferentes, por lo que al aplicarlos el equals dará como resultado false, así que no puede ser la A. No obstante, ambos objetos representan el mismo valor de texto, que es una cadena vacía, por lo que si los comparamos como String dará como resultado verdadero.

6. B. Al realizar la comparación, la variable a vale 100 y b 200, como el resultado es falso, no se ejecutará el operador ternario anidado, así que tendremos que irnos a evaluar el resultado que aparece a continuación de los últimos dos puntos, es decir, b.

7. D. Aunque j es una constante, no se le asigna un valor en la declaración sino después. Esto es correcto hacerlo en Java, pero el *switch* solo admite en los *case* constantes que hayan sido inicializados en la declaración.

8. A, B y F. Son las instrucciones que permiten que x se evalúe como int. En el caso de F, se realizaría un unboxing en la evaluación del *switch*.

4 Creación y uso de arrays

ARRAYS DE UNA DIMENSIÓN

Un array es un conjunto de datos de un mismo tipo a los que se accede mediante una única variable. Cada dato tiene asociado un índice o posición dentro del array, siendo 0 la posición del primer elemento.

Realmente, un array es un objeto y, como tal, podemos manejarlo a través de una variable. La siguiente imagen nos muestra un array de 10 enteros apuntando por una variable v:

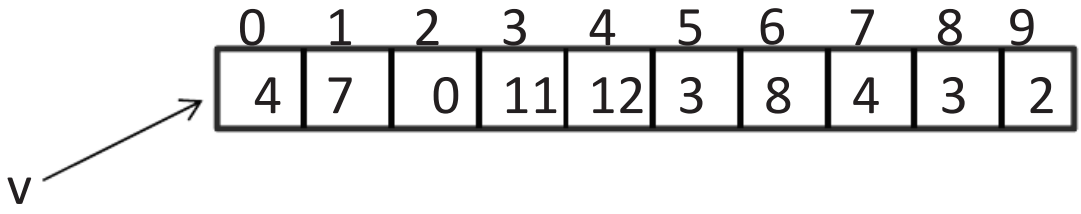


Fig. 4-1. Array de una dimensión

Utilizando la misma variable podemos acceder a cada una de las posiciones del array, para ello, indicaremos entre corchetes la posición a la que queremos acceder:

```
v[0]=4;
```

Los arrays resultan mucho más adecuados que el uso de variables individuales cuando se va a realizar la misma operación para un conjunto de datos, ya que al estar basado en posiciones, puede recorrerse fácilmente con una instrucción *for*.

Antes de entrar en detalles sobre su creación, resaltar que un array es una estructura de datos estática, **una vez definido su tamaño no se puede modificar**.

Declaración e instanciación

Al manejarse a través de variables, los arrays deben declararse.

DECLARACIÓN

Un array se declara:

tipo[] variable;

Por ejemplo, aquí tenemos un array de enteros:

```
int[] m;
```

Y aquí un array de String:

```
String[] cads;
```

Aunque lo habitual es la forma anterior, los corchetes pueden indicarse después de la variable:

```
int m[];
```

Debemos prestar atención ante alguna pregunta de examen relacionada con arrays en la que aparezca algo como esto:

```
int [5] nums; //error de compilación
```

Se trata de un **error de compilación**, pues no se puede indicar el tamaño del array en la declaración de la variable, el tamaño se indica durante la instanciación o creación del array.

Los identificadores de las variables array siguen las mismas normas que las de cualquier otro tipo de variable. En cuanto a la inicialización implícita, al ser un tipo

objeto, las variables array declaradas como atributo de la clase se inicializan a *null*, mientras que las locales no se inicializan de forma implícita.

INSTANCIACIÓN

Al igual que con cualquier otro tipo de objeto, para crear un array utilizaremos el operador *new* aunque, en el caso del array, a continuación del operador se indicará el nombre del tipo de datos del array y entre corchetes el tamaño que se le quiere asignar.

Por ejemplo, la siguiente instrucción crea un array de 10 números enteros que es apuntado por la variable nums:

```
int[] nums=new int[10];
```

Independientemente de que se trata de una variable atributo o local, una vez que se crea un array **todas sus posiciones se inicializan al valor por defecto**:

```
System.out.println(nums[0]);//muestra 0
```

Creación abreviada

Se puede declarar, instanciar e inicializar un array en una misma instrucción:

```
int [] valores=new int[]{3,5,20,11};
```

La instrucción anterior declara y crea un array de enteros de cuatro elementos, inicializando cada uno de estos a los valores 3, 5, 20 y 11. De forma más abreviada, se puede hacer lo mismo de la siguiente manera:

```
int [] valores={3,5,20,11};
```

Acceso a los elementos de un array

Como ya hemos indicado, para acceder a los elementos de un array utilizaremos la variable y entre corchetes el índice o posición del elemento:

```
datos[] int=new int[10];

datos[0]=15;

datos[10]=2; //error!
```

Fijémonos en la última instrucción. Se producirá un error o excepción en tiempo de ejecución de tipo ArrayIndexOutOfBoundsException, debido al intento de acceder a una posición que está fuera de los límites del array.

Todos los arrays disponen de un atributo llamado *length*, que nos indica en todo momento el tamaño del array. Así pues, si quisiéramos acceder a la posición del último elemento, sería length-1:

```
datos[datos.length-1]=30;
```

Si queremos acceder a cada una de las posiciones del array, podemos hacerlo de forma sencilla con una instrucción repetitiva de tipo for. En el siguiente capítulo estudiaremos con detalle esta instrucción, pero a continuación mostramos el siguiente bloque de código de ejemplo mediante el que almacenamos los 10 primeros números pares en un array:

```
int [] datos=new int[10];

//almacena los 10 primeros números pares

for(int i=0;i<datos.length;i++){

    datos[i]=i*2;

}
```

Paso de parámetros de tipo array

Un array es un objeto, por tanto, las variables de tipo array no contienen el array como tal, sino una referencia al mismo. Esto lo vemos en el siguiente ejemplo en el que se le pasa un parámetro de tipo array a un método:

```
void metodo(){

    int[] datos=new int[5];

    guardar(datos);

    System.out.println(datos[0]); //muestra 10

}
```

```
void guardar(int[] valores){

       valores[0]=10;

}
```

Ambas variables, datos y guardar, apuntan al mismo array:

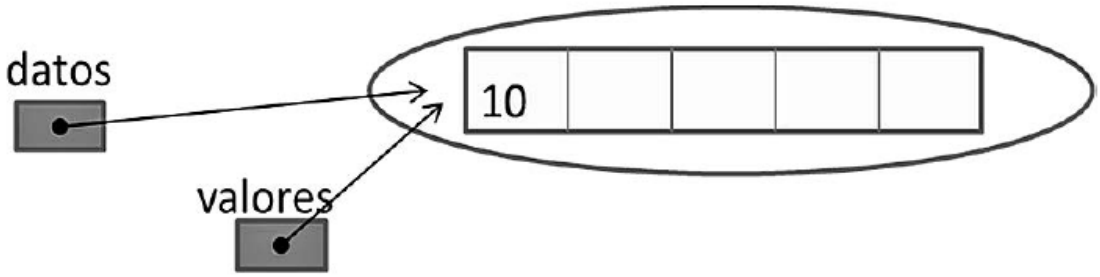


Fig. 4-2. Referencias a array

Por tanto, si utilizamos una de ellas para modificar alguna de las posiciones del array, esa modificación será vista por la otra variable.

Número variable de argumentos

Desde la versión Java 5, es posible declarar métodos en clases Java que reciban un número variable de argumentos. En el siguiente ejemplo vemos cómo hacerlo:

```
public void metodo(int...m)
```

Como vemos, en la lista de parámetros se indica el tipo de estos, seguido de unos puntos suspensivos, y a continuación el nombre de la variable. En la llamada a este método podemos pasarle desde 0 a varios argumentos de tipo *int*. Por ejemplo, las siguientes instrucciones serían todas llamadas válidas al método anterior:

```
metodo();

metodo(4);

metodo(3,5);

metodo(1,9,2);
```

Un parámetro con número variable de argumentos equivale a un array, por tanto, también se podría realizar la llamada al método anterior de la siguiente forma:

```
metodo(new int[]{4,9,3,1});
```

De hecho, en el interior del método los parámetros con número variable de argumentos se tratarían como un array. El siguiente código muestra una implementación de *metodo()*, que consiste en mostrar por pantalla todos los argumentos recibidos:

```
public void metodo(int...m){

    for(int i=0;i<m.length;i++){

          System.out.println(m[i]);

    }

}
```

Es posible tener métodos que combinen número fijo de argumentos con número variable:

```
public void metodo2(int a, int...r)

public void metodo3(int x, String p, long...w)
```

La condición para combinar número fijo con número variable de argumentos es que **el parámetro que reciba número variable de argumentos tiene que ser el último de la lista de parámetros**. Por ejemplo, los siguientes métodos **no compilarían**:

```
public void prueba(int...r, int k)

public void tester(int...m, String...f)
```

ARRAYS MULTIDIMENSIONALES

A los arrays que hemos visto hasta ahora se les conoce también como arrays de una dimensión, pero en Java podemos tener arrays de dos dimensiones, tres dimensiones, etc. Normalmente, en un programa no se suelen utilizar arrays de más de una dimensión, pero es muy probable encontrar preguntas de examen que traten con arrays de varias dimensiones, por ese motivo los estudiaremos con detenimiento y analizaremos sus peculiaridades.

Declaración

Los arrays de varias dimensiones se declaran igual que los de una dimensión, solo que indicaremos tantas parejas de corchetes ([]) como dimensiones tenga el array. A continuación vemos unos ejemplos:

```
int [][] ar;// array de dos dimensiones

int[]ar1[]; //también array de dos dimensiones

int [][][] ar2; //array de tres dimensiones
```

Como podemos ver en la segunda de las declaraciones, es posible indicar algunas de las parejas de corchetes delante de la variable y otras detrás.

Instanciación y acceso a elementos

Para instanciar un array multidimensional, utilizamos igualmente el operador *new*, indicando entre los corchetes el tamaño de cada dimensión:

```
ar1=new int[3][4]; //array de 12 elementos

ar2=new int[2][5][10]; //array de 100 elementos
```

Los arrays de dos dimensiones podemos imaginarlos como una tabla, por ejemplo, el array de la variable ar1 equivale a una tabla de tres filas por cuatro columnas:

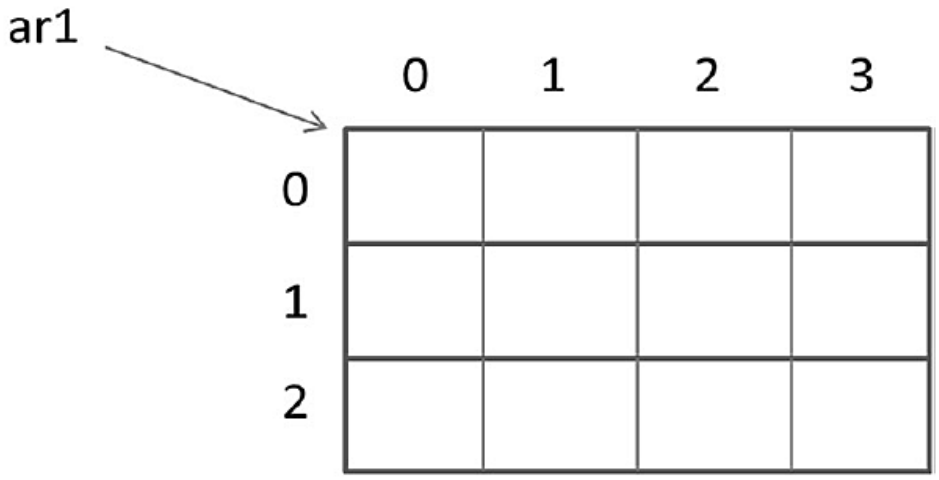


Fig. 4-3. Array de dos dimensiones

Por su parte, el array de tres dimensiones lo imaginamos como una especie de cubo tridimensional. Aunque de cuatro en adelante ya no podemos hacernos una representación mental de cómo quedaría, siempre se cumple que el número de elementos totales es el producto de los tamaños de cada dimensión.

El acceso a los elementos de un array multidimensional se realiza con un índice por dimensión:

```
ar1[1][2]=23;

ar2[0][3][0]=8
```

Recorrido de un array multidimensional

Para recorrer un array de varias dimensiones utilizaremos una instrucción repetitiva *for* para cada dimensión:

```
int [][] nums=new int[5][7];

for(int i=0;i<nums.length;i++){ //longitud primera dimensión

        for(int k=0;k<nums[i].length;k++){//longitud

                                    //segunda dimensión
```

```
                System.out.println(nums[i][k]);
            }
    }
```

Arrays irregulares

Durante la creación de un array multidimensional podemos dejar alguna de sus dimensiones sin asignar tamaño:

```
int[][] d=new int[5][];

int [][][] n=new int[4][][];

int [][][] v=new int[2][10][];
```

Siempre tiene que ser la última o últimas dimensiones las que se queden sin asignar. Por ejemplo, la siguiente instrucción sería incorrecta porque deja sin asignar tamaño a una dimensión intermedia:

```
int [][][] h=new int[6][][4]; //error de compilación
```

A este tipo de arrays se les conoce como **arrays irregulares** porque, a cada posición definida se le puede asignar un array de tantas dimensiones como queden por asignar, pudiendo ser cada uno de estos arrays de tamaño diferente. Veamos un ejemplo, supongamos que tenemos el siguiente array:

```
int[][] data=new int[3][];
```

Se trata de un array irregular de dos dimensiones al que solo se ha dado tamaño a la primera dimensión. Podemos imaginarlo como un array de tres elementos, donde **cada uno de ellos podrá contener un array de una dimensión**, aunque implícitamente, se encuentran inicializados a *null*:

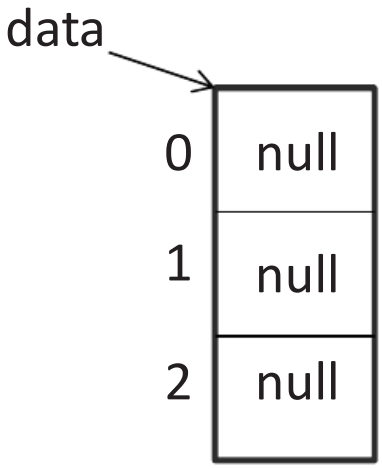


Fig. 4-4. Inicialización por defecto de la primera dimensión

Para asignar contenido (un array) a cada elemento, sería:

```
data[0]=new int[2];

data[1]=new int[5];

data[2]=new int[]{8,3,5};
```

Como vemos, tenemos un array irregular, es decir, un array de arrays donde cada uno puede tener un tamaño diferente:

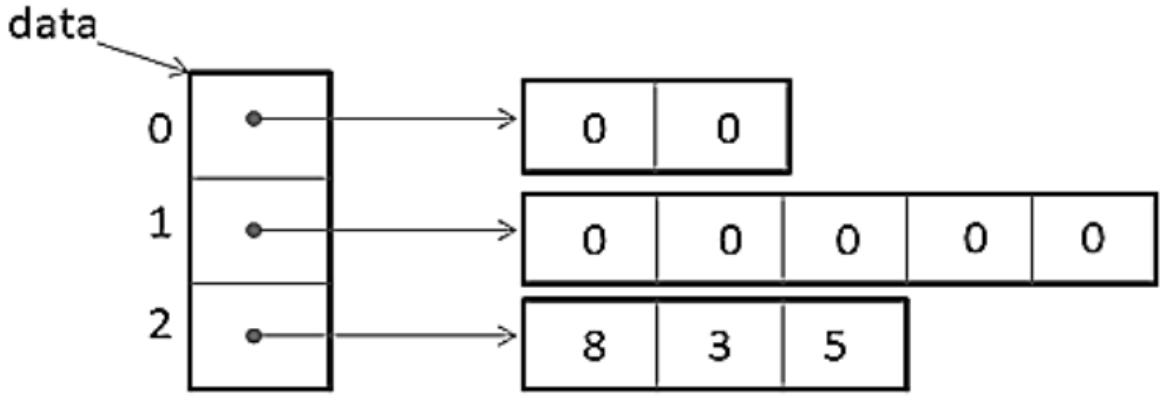


Fig. 4-5. Array irregular

El acceso a los elementos se realiza de la misma forma que en un array multidimensional regular:

```
System.out.println(data[2][0]);//muestra 8
```

Es seguro que encontremos alguna pregunta del examen de certificación en la que aparezcan involucrados arrays multidimensionales. En estas preguntas, se nos darán una serie de arrays y habrá que determinar si ciertas instrucciones de creación/asignación son correctas. Por ejemplo, fijémonos en los siguientes arrays:

```
int[] a=new int[10];

long[][] b= new long[2][3];

int[][] c=new int[3][];

long[][] d[]=new long[5][][];

String [][] s=new String[][5];
```

De las cinco instrucciones anteriores, las cuatro primeras serían correctas, pero la quinta no, ya que se ha dejado una de las dimensiones intermedias sin asignar tamaño y no compilaría.

Por su parte, dados los arrays anteriores a, b, c y d, las siguientes asignaciones serían correctas:

```
b[0][0]=a[1];

c[1]=a; //cada elemento de la primera dimensión es un array

d[0]=b; //cada elemento de la primera dimensión

     //es un array bidimensional
```

Pero estas otras serían incorrectas, el motivo se explica en cada comentario:

```
c[0][0]=a[1]; //no se ha dado tamaño a la segunda dimensión

c[2]=a[3]; //en la primera dimensión de c se
```

```
            //debe asignar un array
d[1]=a; //tiene que ser array de dos dimensiones
d[0]=c; //misma dimensión, pero un array de int
        //no se puede asignar a una variable array de long
```

Para finalizar con el estudio de los arrays multidimensionales, comentar que es posible crear e inicializar un array en una misma instrucción:

```
int[][]nums={{2,3,4},{5,6}};
```

En la instrucción anterior tenemos un array bidimensional irregular, donde el primer elemento de la primera dimensión es un array de tres elementos y el segundo es un array de dos.

PREGUNTAS TIPO EXAMEN

Pregunta 1

Given the following:

```
public class Test {
  static int [] n;
  public static void main (String []args) {
        System.out.println(n[0]);
  }
}
```

What is the result?

A. The output is 0

B. The output is null

C. Compilation will fail

D. NullPointerException

Pregunta 2

Given the following:

```
public class Test {
  public static void main (String []args) {
        static int [] n;
        System.out.println(n[0]);
  }
}
```

What is the result?

A. The output is 0

B. The output is null

C. Compilation will fail

D. NullPointerException

Pregunta 3

Given the following:

```
10.  float f1 = 4.3f;
11.  float [] f2 = new float[5];
12.  float [][] f3 = new float [2][];
13.  ?
```

Which of the following expresions could be placed in the line 13? (choose two)

A. f2[2] = f1;

B. f3[0][0] = f1;

C. f3[1]= f2[0];

D. f3[0] = f2;

Pregunta 4

Which of these array instantations are not legal? (choose 3)

A. String[] str = String[3];

B. int[]k[] d = new int[3][];

C. int[][][] s=new int[2][][];

D. long[][] k=new int[3][4];

E. String[][] c=new String[][5];

F. int[][] n={{4,5,7},{8,9}};

Pregunta 5

Given the following:

```
public class Test {
   public static void main(String[] args) {
      int sum=0;
      int [][] s=new int[2][];
      s[0]=new int[]{1,1,2};
      //line x
      for(int c[]:s){
         for(int n:c){
            sum+=n;
         }
      }
      System.out.println(sum);
   }
}
```

Which of the following statements must be placed in line x to program prints 5?(choose 2)

A. s[1]=1

B. s[1]=new int[]{0,1};

C. s[1]=new int[]{0,0};

D. s[1]=new int[]{1};

Pregunta 6

Given the following array:

```
int[] mAr={5,9,12,4,7}
```

Which two code fragments, independently, print each element in this array?

A.
```
for(int i:mAr){
        System.out.println(mAr[i]+" ");
}
```

B.
```
for(int i:mAr){
        System.out.println(i+ " ");
}
```

C.
```
for(int i=0:mAr){
        System.out.println(mAr[i]+" ");
        i++;
}
```

D.
```
for(int i=0;i<mAr.length;i++){
        System.out.println(i+" ");
}
```

E.
```
for(int i=0;i<mAr.length;i++){
        System.out.println(mAr[i]+" ");
}
```

F.
```
for(int i;i<mAr.length;i++){
        System.out.println(mAr[i]+" ");
}
```

SOLUCIONES

1. D. Se produce un NullPointerException porque la variable array está inicializada implícitamente a null.

2. C. En este caso la variable array es local y no se inicializa implícitamente, por lo que al intentar utilizarla sin haber sido inicializada se producirá un error de compilación.

3. A y D. La A es correcta porque f2 es un array unidimensional de tipo float y, dado que f1 es un dato float, puede asignarse a cualquiera de las posiciones del array. La D es correcta porque f3, al ser un array bidimensional, cada posición es un array de float y dado que f2 es un array de este tipo, puede asignarse a las posiciones de f3.

4. A, D y E. En la instrucción de A falta el operador new. La D no es correcta porque un array de int no puede asignarse a un array de long. La E no es correcta porque, en un array bidimensional, no puede quedar sin tamaño una dimensión intermedia.

5. B y D. El programa suma todos los números contenidos en el array bidimensional, por tanto la segunda posición de la primera dimensión deberá ser un array cuyos elementos sumen 1. Tanto la instrucción B como la D cumplen con esta característica.

6. B y E. La A no es correcta porque la variable de control de un *enhanced for* no se usa como índice, sino como referencia a cada elemento del array. En la C, la definición del enhanced for es incorrecta, mientras que en la D la variable del for no contiene al elemento, debe utilizarse como índice para acceder a cada posición.

5 Estructuras repetitivas

INSTRUCCIONES REPETITIVAS FOR Y WHILE

Las instrucciones repetitivas o bucles se emplean para ejecutar varias veces un determinado bloque de sentencias. Existen dos instrucciones de este tipo en Java: *for* y *while*, ya hemos visto cómo utilizar la instrucción *for* para recorrer un array, pero vamos a analizar en detalle la sintaxis de cada una de ellas.

Instrucción for

Ejecuta un grupo de instrucciones un número determinado de veces, definido por los valores que va tomando una variable de control.

SINTAXIS

Su sintaxis es la siguiente:

```
for(inicializacion;condicion;incremento){

    //instrucciones

}
```

En la instrucción de inicialización, que se ejecuta una única vez, se inicializa la variable de control del bucle for. Mientras la condicion se cumpla, se ejecutará el bloque de sentencias definidas dentro del for. Al finalizar cada iteración (ejecución del bloque) se ejecutará la instrucción de control que, como su nombre indica, suele realizar un incremento o decremento de la variable de control.

El siguiente código de ejemplo nos muestra por pantalla los números naturales del 1 al 10:

```
for(int i=1;i<10;i++){

    System.out.println(i);

}
```

CONSIDERACIONES

A la hora de utilizar la instrucción *for*, y de cara a las posibles preguntas de examen que puedan aparecer enfocadas en esta instrucción, debemos de tener en cuenta las siguientes consideraciones:

- Las llaves solo son obligatorias si el bloque está compuesto por más de una instrucción.

- Las tres **instrucciones de control son opcionales**. Podría no aparecer alguna de las instrucciones, o incluso ninguna (bucle infinito), pero es necesario siempre indicar el punto y coma de separación. Por ejemplo, el siguiente bloque de instrucciones sería equivalente al anterior:

```
//muestra los números del 1 al 10

int i=1;

for(;i<10;){  //los ; se ponen igualmente

      System.out.println(i);

      i++;

}
```

- Las instrucciones de control pueden contener más de una sentencia, separándolas por una coma. La siguiente sentencia nos muestra un ejemplo:

```
for(int a=0,b=10;a<b;a++,b--){

}
```

Instrucción enhanced for

La instrucción *enhanced for*, conocida también como *for each*, es una variante de for diseñada para el recorrido de arrays y colecciones en modo lectura.

Con un *enhanced for*, en lugar de utilizar un índice para acceder a cada elemento, la variable de control apunta directamente a cada una de las posiciones del array.

El formato de esta instrucción es el siguiente:

```
for(tipo variable:array){

    //instrucciones

}
```

Por ejemplo, supongamos que tenemos el siguiente array de enteros:

```
int[] nums={4,8,1,5};
```

Si queremos recorrer el array con un *enhanced for* y mostrar cada uno de sus elementos por pantalla, sería:

```
for(int n:nums){

    System.out.println(n);

}
```

La variable de control *n* va pasando por todas las posiciones del array, NO contiene la posición del elemento, sino el propio elemento:

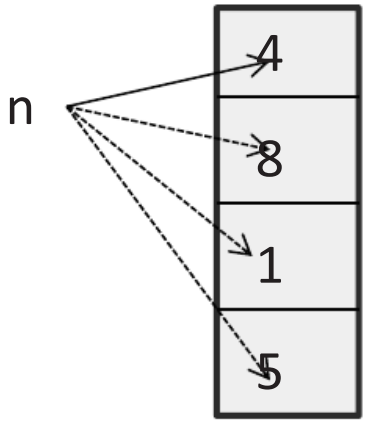


Fig. 5-1. Recorrido de array con enhanced for

Instrucción while

La instrucción *while* ejecuta un grupo de sentencias mientras se cumpla una condición.

FORMATO

El formato de la instrucción *while* es el siguiente:

```
while(condicion){

    //instrucciones

}
```

Se comprueba la condición, que debe ser de tipo *boolean*, y si resulta verdadera, se ejecutará el bloque de sentencias. Al finalizar el bloque, se vuelve a evaluar la condición y si sigue siendo verdadera se ejecutará de nuevo el bloque, así hasta que llegue el momento en que sea falsa.

Las acciones dentro del bloque provocarán que en algún momento la condición deje de cumplirse, sino estaríamos ante un bucle infinito.

El siguiente bloque de código de ejemplo realiza la suma de todos los números naturales desde 1 en adelante hasta que la suma alcance o supere el valor 1000:

```
int n=0;s=0;

while(s<1000){

    s+=n++;

}
```

En este caso se debe utilizar un while en lugar de un for porque el número de veces que hay que realizar la operación no es conocido de antemano ni definido por ninguna variable.

UTILIZACIÓN DE DO WHILE

Existe una variante del bucle *while*, conocida como *do while*, donde primero se ejecuta al bloque de sentencias y después se comprueba la condición, si esta se cumple, vuelve a ejecutarse de nuevo el bloque. Este es su formato:

```
do{

    //instrucciones

} while(condicion);
```

Suele utilizarse en aquellos casos en los que primero debe obtenerse un valor para poder comprobar la condición después. Por ejemplo, el siguiente bloque realizaría la lectura de un número a través de un hipotético método *leerNumero()*, mientras este número sea negativo:

```
int n=0;

do{

    n=leerNumero();

}while(n<0);
```

LAS INSTRUCCIONES BREAK Y CONTINUE

Las instrucciones *break* y *continue* se utilizan para abandonar de forma forzada una instrucción repetitiva. Ambas se pueden emplear tanto en bucles *for* como *while*/*do while*, vamos a ver cómo funciona cada una de ellas.

Instrucción break

Provoca la **salida de la instrucción repetitiva**, pasando el control del programa a la siguiente instrucción. En el siguiente código de ejemplo vamos sumando todos los números naturales desde 1 hasta un número leído, pero si encontramos el número 100 abandonamos la operación:

```
int n=leerNumero();

int s=0;

for(int i=1;i<n;i++){

    s+=i;

    if(s>100){

         break; //sale fuera del for

    }

}
```

Instrucción continue

La instrucción *continue* no provoca una salida de la instrucción repetitiva, sino que **salta a la siguiente iteración**. En el caso de un *for*, la llamada a *continue* nos llevaría directamente a la instrucción de incremento. El siguiente código de ejemplo muestra por pantalla todos los números del 1 al 10, excepto el 5:

```
for(int i=1;i<10;i++){

    if(i==5)

         continue; //salta a la instrucción i++

    System.out.println(i);

}
```

Bucles etiquetados

Es posible establecer una etiqueta en un bucle *for* o *while*. Esto permite, en bucles anidados, especificar a través de las instrucciones *break* o *continue* cuál de los bucles se quiere abandonar.

Para etiquetar una instrucción repetitiva, se indica el nombre de la etiqueta delante de la instrucción, seguido de dos puntos:

etiqueta: for(...){}

etiqueta: while(...){}

En el interior del bloque de instrucciones, para indicar que queremos aplicar el *break* o *continue* sobre un determinado bucle utilizamos:

break etiqueta;

continue:etiqueta;

La siguiente figura nos muestra una estructura clásica de utilización:

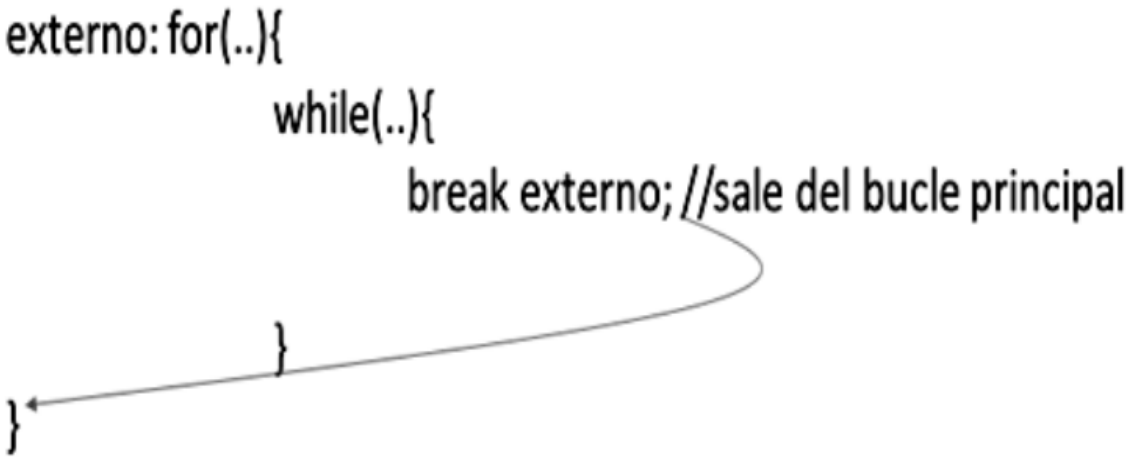


Fig. 5-2. Salida de un bucle externo

Observemos el siguiente código de ejemplo:

```
public class TestBuclesAnidados {

  public static void main(String[] args) {

    int a=0,s=0,i=1;

     principal:

      for(;i<10;i++){

        while(a<5){

           a=i++;

           s=a+i;

           if(s>=10){

              break principal;

           }

         }

       }

       System.out.println(i+":"+a+":"+s);

   }

}
```

¿Qué se mostraría al ejecutar el programa anterior?

La respuesta es 6:5:11.

PREGUNTAS TIPO EXAMEN

Pregunta 1

Given the following:

```
1. public class Runner {
2. static int a=1;
2. public static void main (String []args) {
3.    int b = 6;
4.    while (a++) {
5.              b--;
6.    }
7.    System.out.println("a=" + a + " b =" + b);
8. }
9. }
```

What is the result?

A. The output is a = 6 b = 0

B. The output is a = 7 b = -1

C. The output is a = 6 b = -1

D. The output is a = 7 b = 0

E. Compilation will fail.

Pregunta 2

Given the code fragment:

```
public static void main(String[] args){
    int x=0;
    int y=7;
    for(x=0;x<y-1;x=x+2){
        System.out.println(x+" ");
    }
}
```

What is the result?

A. 2 4

B. 0 2 4 6

C. 0 2 4

D. Compilation fails

Pregunta 3

Given the code fragment:

```
public static void main(String[] args){
    String[] cars={"A","B","C","D"};
    for(int i=0;i<cars.length;i++){
        System.out.print(cars[i]+" ");
        if(cars[i].equals("C")){
```

```
                continue;
            }
            System.out.println("End");
            break;
        }
}
```

What is the result?

A. A B C End

B. A B C D End

C. A End

D. Compilation fails

Pregunta 4

Given the code fragment:

```
public static void main(String[] args){
    String[][] cars={{"A","B","C"},{"D","E"}};
    ex: for(int i=0;i<cars.length;i++){
            for(int k=0;k<cars[i].length;k++){
                System.out.print(cars[i][k]+" ");
                if(cads[i][k].equals("B")){
                    break;
                }
```

```
                    continue ex;
                }


        }
}
```

What is the result?

A. A B C

B. A D

C. A B D E

D. Compilation fails.

Pregunta 5

Given the code fragment:

```
int m[]={1,2,3,4,5};
for(---){
    System.out.print(m[i]);
}
```

Which option can replace --- to enable the code to print 135?

A. int i = 0; i <= 4; i ++

B. int i =0; i <5; i += 2

C. int i = 1; i <= 5; i += 1

D. int i = 1; i <5; i +=2

Pregunta 6

Given the following:

```
public static void main(String[] args){
    int a=5;
    while(isPresent(a)){
            System.out.println(a);  //line 1
            //line 2
    }
}
public static boolean isPresent(int a){ //line 3
    return a-->0?true:false;
}
```

Which modification enables the code to print 54321?

A. Replace line 1 with System. out. print (--a) ;

B. At line 2, insert a --;

C. Replace line 1 with --a; and, at line 2, insert system. out. print (a);

D. Replace line 3 With return (a > 0) ?false: true;

SOLUCIONES

1. E. Se produce un error de compilación en la línea 4: while (a++). Recordemos que la condición del while, al igual que la del if, debe dar siempre como resultado un tipo boolean.

2. C. Se recorre desde 0 hasta 5, en pasos de dos en dos, por lo que se imprimirán los números 0, 2 y 4.

3. C. Tras imprimirse la primera lectura del array, "A" y el texto End, se abandona el bucle con un break, finalizando el programa.

4. B. Tras imprimirse el primera cadena del array, "A", la instrucción continue ex provoca que se abandone el bucle interno y se pase a la siguiente iteración del externo, esto hará que se acceda a la posición [1,0], donde se encuentra la cadena "D". Después de imprimirse esta, se volverá a ejecutar continue ex y como el bucle externo ya ha finalizado su recorrido, el programa termina.

5. B. Se trata de imprimir las posiciones impares del array. Es la combinación indicada en B la que realiza esta tarea al realizar incrementos de dos en dos y partir desde la posición 0.

6. B. Para que salgan todos los números de 5 a 1, hay que ir decrementando la variable a hasta 1. La A no es correcta porque se debe mostrar el valor de la variable antes de decrementar, así comenzará desde el valor actual de la misma.

6 Métodos y encapsulación

CREACIÓN DE MÉTODOS EN JAVA

Aunque ya hemos creado y utilizado métodos de clases Java en capítulos anteriores, en este apartado vamos a analizar el detalle de la creación de métodos y nos vamos a centrar en una de las características, tanto de métodos como de constructores, que pueden ser objetivo de preguntas de examen: la sobrecarga.

Definición y estructura de un método

Un método es una función, que se define dentro de una clase Java y que realiza algún tipo de tarea. Puede recibir parámetros y devolver un resultado. La estructura de un método es:

```
modificador tipo_devolucion nombre_metodo(parametros){

}
```

El modificador es una palabra reservada que define el comportamiento de un método, a lo largo de este capítulo veremos los modificadores más importantes utilizados en la definición de un método.

Ejemplos de definición de métodos serían:

```
public int sumar(int a, int b){
    int s=a+b;
    return s; //devolución
}
public void imprimir(int a){
    System.out.println(a);
}
```

Como vemos, los métodos que devuelven un resultado utilizan la palabra *return*, seguido de la expresión que genera el resultado a devolver. Los métodos de tipo *void*, es decir, que no devuelven ningún resultado al punto de llamada, no necesitan utilizar la palabra *return*.

Llamada a métodos

Los métodos se definen dentro de clases, por lo que para llamarlos desde fuera de esta se debe crear primero un objeto de la clase y utilizar la sintaxis:

objeto.metodo(argumentos);

Desde otro método de su propia clase se puede llamar al método directamente, utilizando la expresión:

metodo(argumentos);

Si el método devuelve un resultado, este se recogerá en una variable o formará parte de alguna otra expresión.

Por ejemplo, dada la siguiente clase Calculadora en la que se define el método sumar:

```
class Calculadora{
   public int sumar(int a, int b){
```

```
    int s=a+b;
    return s; //devolución
   }
}
```

Para crear un objeto de esta clase y realizar la llamada al método sería:

```
class Test{
   public static void main(String[] ar){
         Calculadora cl=new Calculadora();
         int s=cl.sumar(3,8);//devuelve 11 en s
}
```

Los argumentos de la llamada tienen que coincidir con la lista de parámetros definidos en el método, de modo que cada argumento se volcará en la variable parámetro según el orden en el que se indiquen:

Fig. 6-1. Paso de parámetros a métodos

En este caso, el argumento 8 se volcaría en la variable a y el 3 en b. En siguientes apartados veremos las diferencias entre pasar argumentos de tipo primitivo y tipo objeto.

Sobrecarga de métodos

La sobrecarga de métodos consiste en definir, en una misma clase, varios métodos con el mismo nombre, pero diferenciándose en la lista de parámetros. El motivo de tener varios métodos con el mismo nombre es ofrecer al programador

varias opciones para realizar una misma función, en lugar de darles nombres diferentes, podemos llamar de la misma forma a todos los métodos que realizarán esa tarea.

Por ejemplo, en la clase Calculadora podríamos tener tres métodos sumar para realizar la operación de suma. Estos métodos podrían definirse:

public int sumar(int a, int b){..}

public int sumar(int a){..}

public int sumar(long b){..}

Como vemos, la lista de parámetros de los tres métodos es diferente, por lo que no podría darse ninguna ambigüedad a la hora de llamar a cada método. Es importante resaltar que el tipo de devolución de un método no afecta a la sobrecarga, es decir, los métodos sobrecargados pueden tener el mismo tipo de devolución o diferente.

La versión del método que será llamado se determina en función de los argumentos de la llamada:

Fig. 6-2. Sobrecarga de métodos

A continuación, presentamos algunos ejemplos válidos de sobrecarga, es decir, de métodos que podrían estar definidos dentro de la misma clase:

```
public int imprimir(int a){..}

public void imprimir(){..}
```

```
public int imprimir(long b){..}
```

En cambio, si el método siguiente estuviera definido en una clase:

```
public int imprimir(int a){..}
```

Los métodos que se indican a continuación no podrían estar definidos en esa misma clase:

```
//error compilación, coincidencia de parámetros
public void imprimir(int s){..}
//compila, pero no sobrecarga, habría un imprimir() y otro Imprimir()
public int Imprimir(int b){..}
```

Es posible encontrar alguna pregunta de examen en la que aparezca una clase con varias versiones de un método y una instrucción de llamada que podría coincidir con varios de estos métodos. Por ejemplo, fijémonos en el siguiente código:

```
public class Test {
    public static void main(String[] args) {
          metodo(5);
    }
    static void metodo(int k) {
          System.out.println("int");
    }
    static void metodo(long k) {
          System.out.println("long");
    }
    static void metodo(Integer k) {
          System.out.println("Integer");
```

```
    }

    static void metodo(int... k) {

          System.out.println("variable");

    }


}
```

La pregunta aquí sería: ¿qué se imprimiría al ejecutar el programa anterior?, que es lo mismo que preguntarnos: ¿cuál de los cuatro métodos se ejecutaría en la llamada?

Para responder correctamente a este tipo de preguntas, tenemos que tener en cuenta las reglas que sigue el compilador a la hora de determinar cuál de los métodos debe ser llamado. Según estas reglas, cuando hay varios posibles métodos que se pueden ejecutar en una llamada **primero se intenta buscar coincidencia exacta, si no intenta promoción de tipos, si no autoboxing y, en último lugar, busca un método con número variable de argumentos**.

Según lo anterior, se imprimiría *int* porque llamaría al primero de los métodos, que es el que recibe un parámetro que coincide exactamente con el tipo argumento de llamada. Si este primer método no estuviera, llamaría al que recibe un *long*, si tampoco estuviera, llamaría al tercero y si tampoco estuviera, llamaría al que recibe un número variable de argumentos.

Siguiendo estas pautas nos resultará siempre muy sencillo determinar qué ocurrirá al ejecutar un programa con sobrecarga de métodos.

Si lo que nos encontramos es algo como esto:

```
public class Test {

    public static void main(String[] args) {

          metodo(5);

    }

    static void metodo(Long k) {
```

```
            System.out.println("Long");
        }
    }
```

Aquí directamente tendríamos un **error de compilación en la llamada a metodo()**, ya que no hay ningún método en la clase que coincida con el argumento *int* de llamada. Al ser de tipo *Long*, para poder llamarlo se debería hacer una promoción de tipo a *long* y después un *autoboxing*, pero **el compilador no puede hacer esas dos operaciones en una llamada**. O promoción de tipos, o autoboxing o llamada a número variable de argumentos, pero **no dos de ellas**.

PASO DE PARÁMETROS A MÉTODOS

En este apartado vamos a analizar los efectos de pasar parámetros de tipos primitivos a un método, frente al paso de parámetros de tipo objeto, ya que podemos encontrar más de una pregunta de examen centrada en este objetivo.

Paso de tipos primitivos

Al pasar un tipo primitivo a un método, estamos pasando una copia del dato, de modo que si el método modifica su copia, el original no se ve afectado. Fijémonos en el siguiente ejemplo, supongamos que tenemos esta clase:

```
class Calc{
    public void modif(int a){
        a=a+3;
    }
}
```

Al realizar una llamada al método modif() con un número entero, este se copia en la variable a, de modo que si esta variable se modifica como en el ejemplo, ese cambio solo afecta a la variable no al dato original:

```
class Test{

   public static void main(String[] ar){

      Calc cl=new Calc();

      int n=5;

      cl.modif(n);

      System.out.println(n); //muestra 5, n no cambia

}
```

La siguiente imagen nos ilustra lo que sucede en la memoria con el paso de parámetros de tipo primitivo:

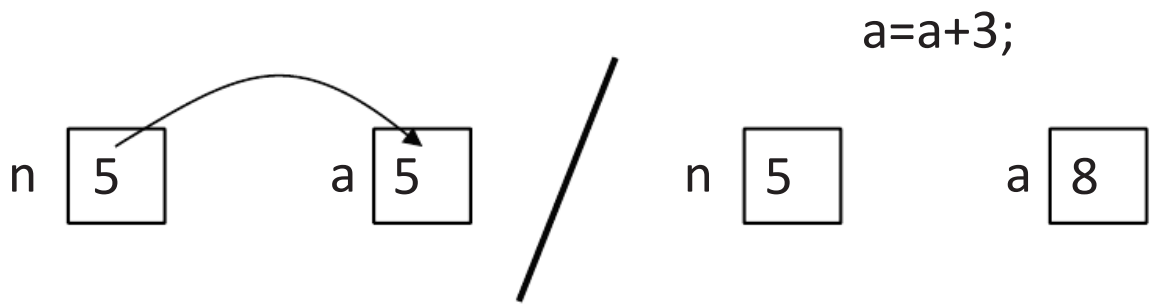


Fig. 6-3. Paso de parámetros de tipos primitivos

Paso de tipos objeto

Cuando pasamos un objeto en la llamada a un método, estamos pasando una **copia de la referencia al objeto**. Esto significa que tanto la variable argumento como la variable parámetro apuntan al mismo objeto, de modo que si dentro del método hacemos una llamada a un método del objeto que haga algún cambio en el mismo, este cambio también lo veremos al acceder al objeto con la variable original.

Por ejemplo, supongamos que tenemos la siguiente clase en donde definimos un método que recibe como parámetro un StringBuilder:

```
class Calc{

   public int modif(StringBuilder d){

        d.append(" bye");

   }

}
```

El método *append()* de StringBuilder lo estudiaremos más adelante, de momento indicar que lo que hace es agregar un nuevo texto al objeto original. En la siguiente clase creamos un objeto Calc y llamamos al método *modif()* con un texto, si después de la llamada mostramos el texto por pantalla veremos que ha cambiado:

```
class Test{

   public static void main(String[] ar){

      Calc cl=new Calc();

      StringBuilder sb=new StringBuilder("hello");

      cl.modif(sb);

      System.out.println(sb.toString()); //hello bye

}
```

El motivo es, como hemos dicho, que ambas variables, el argumento y el parámetro, apuntan al mismo objeto:

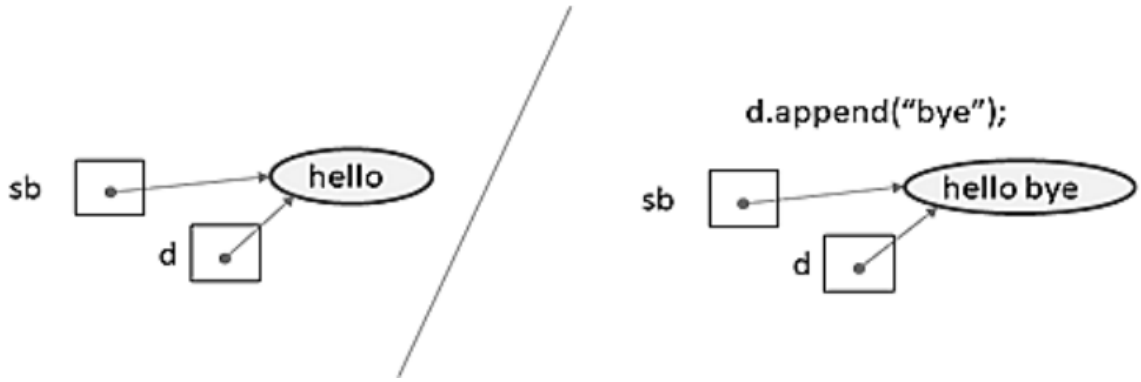


Fig. 6-4. Paso de parámetros de tipo objeto

Paso de objetos tipo String

Como ya vimos en el capítulo 3, las cadenas de caracteres String, aunque son objetos, son inmutables, lo que significa que si se pasan como parámetro a un método y este modifica la cadena, lo que realmente estará haciendo es generar una nueva resultante de la operación, por tanto, la original no se verá afectada. Por ejemplo, si tenemos la siguiente clase:

```
class Calc{

   public int modif(String d){

        d+=" bye";

   }

}
```

Tras ejecutar el siguiente código se mostrará la misma cadena que se pasó como parámetro:

```
class Test{

   public static void main(String[] ar){

      Calc cl=new Calc();

      String sb=new String("hello");

      cl.modif(sb);

      System.out.println(sb); //hello

}
```

La siguiente figura ilustra la situación:

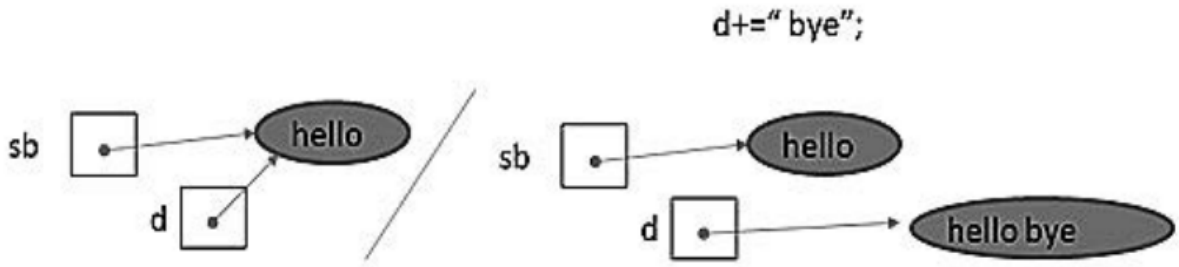


Fig. 6-5. Parámetros tipo String

MIEMBROS ESTÁTICOS DE UNA CLASE

Los miembros estáticos de una clase son aquellos que pueden ser utilizados sin necesidad de crear ninguna instancia de la clase, pues son independientes de estas. Cuando hablamos de miembros estáticos, nos referimos a atributos y métodos estáticos, aunque también hay otro elemento que vamos a estudiar en este capítulo que son los bloques estáticos. Comencemos.

Métodos estáticos

Un método estático es aquel que **no está asociado a ningún objeto particular de la clase**. La ejecución del mismo no depende de ningún dato propio de una instancia, dando siempre el mismo resultado para todas ellas.

CREACIÓN

Un método estático se crea igual que cualquier otro método de la clase, indicando la palabra *static* delante del tipo de devolución:

```
class Calc{

   public static int cuadrado(int a){

        return a*a;

   }

}
```

Como vemos en este ejemplo, la ejecución del método cuadrado solo depende del parámetro recibido, no emplea ningún atributo asociado a una instancia para su ejecución, su resultado sería el mismo independientemente del objeto con el que se llamase a este método.

LLAMADA A UN MÉTODO ESTÁTICO

Como no dependen de ningún objeto de la clase, **no es necesario crear un objeto para llamar a estos métodos,** se utiliza el nombre de la clase:

```
int res=Calc.cuadrado(3);
```

Eso no impide que no se pueda crear un objeto y llamar con él al método estático:

```
Calc c=new Calc();

int s=c.cuadrado(3); //correcto
```

Por si lo encontramos en alguna pregunta de examen, tener en cuenta que lo anterior es perfectamente correcto, aunque, como hemos dicho, no sería necesario.

Si llamásemos al método con otro objeto diferente:

```
Calc c2=new Calc();

int r=c2.cuadrado(3);
```

El resultado sería exactamente el mismo que con el primer objeto, pues el método solo hace uso del parámetro para su ejecución.

CONSIDERACIONES SOBRE EL USO DE MÉTODOS ESTÁTICOS

Dado que no dependen de ningún objeto particular de la clase, los métodos estáticos **solo pueden llamar a otros miembros de su misma clase que también sean static**. En el siguiente código de ejemplo podemos ver algunos casos válidos y no válidos de utilización de otros miembros de la clase:

```
class Test{

   int a=2;

   static int b=5;
```

```
    public static int metodo(){
         int c=a*3;// error de compilación
         int n=b+1; //correcto
         imprime(n); //correcto
    }
    static void imprime(int s){..}
}
```

Como se indica en el comentario, no es posible hacer uso del atributo *a* desde el método estático, dado que los atributos representan características de los objetos.

Tampoco podemos utilizar las palabras reservadas *this* y *super* dentro de un método estático, pues están asociadas al objeto en ejecución.

Atributos estáticos

Se les conoce también como **variables de clase**, ya que no están asociados a ningún objeto particular de la misma y no representan características de un objeto. No son técnicamente atributos.

Se definen con la palabra *static* delante del tipo:

```
class Test{
     static int n=0;
     public void inc(){
          n++;
     }
     public int getN(){return n;}
}
```

Habitualmente, estas variables no son privadas sino que suelen ser accesibles desde fuera de la clase. Como en el caso de los métodos, el acceso a estas variables es NombreClase.variable:

Test.n=10;

Es probable encontrar alguna pregunta de examen en la que se manipule alguna variable estática utilizando dos o más objetos de la clase. En estos casos debemos de tener en cuenta que las variables estáticas son compartidas por todos los objetos de la clase. Fijémonos en el siguiente código que hace uso de la clase anterior:

```
class Prueba{

   public static void main(String[] ar){

         Test t1=new Test();

         t1.inc();

         Test t2=new Test();

         t2.inc();

         System.out.println(t1.getN());//muestra 2

         System.out.println(t2.getN());//muestra 2

   }

}
```

Como el método inc() manipula una variable estática, al ser llamado sobre dos objetos, ambos están haciendo uso de la misma variable.

Bloques estáticos

Un bloque estático es un bloque de código que se ejecuta **una sola vez durante la vida de la clase** dentro de una aplicación. Se definen con la palabra *static* y, a continuación, entre llaves , el código del bloque:

class Clase{

```
    static{
        //bloque static
    }
}
```

En el interior del bloque estático solo podemos hacer uso de miembros estáticos (variables de clase y métodos).

El bloque estático se ejecutaría una sola vez, por ejemplo, con la creación del primer objeto de la clase o con la primera llamada a uno de sus métodos estáticos. Fijémonos en la siguiente clase:

```
class Test{
    static int n=0;
    static{
        n++;
    }
    public int getN(){return n;}
}
```

Si desde otra clase creamos varios objetos de esta, solo con la creación del primero se ejecutaría el bloque estático:

```
class Prueba{
   public static void main(String[] ar){
        Test t1=new Test();
        Test t2=new Test();
        System.out.println(t1.getN());//1
        System.out.println(t2.getN());//1, no se ha
```

```
                                    //ejecutado el bloque static
    }
}
```

En caso de que la clase anterior tuviera un constructor, el bloque *static* se ejecutaría antes **que el constructor**.

CONSTRUCTORES

Los constructores son bloques de código que se definen en una clase y que se ejecutan cada vez que se crea un objeto de la misma. Su finalidad es realizar algún tipo de inicialización antes de que el objeto sea utilizado, por ejemplo, asignar valores iniciales a los atributos.

Sintaxis

Los constructores se definen de forma similar a los métodos, aunque su nombre debe coincidir con el de clase y no tienen tipo de devolución. Al igual que los métodos, pueden recibir parámetros.

Como se ilustra en la siguiente imagen, el constructor se ejecuta durante la creación de un objeto de la clase a través del operador new:

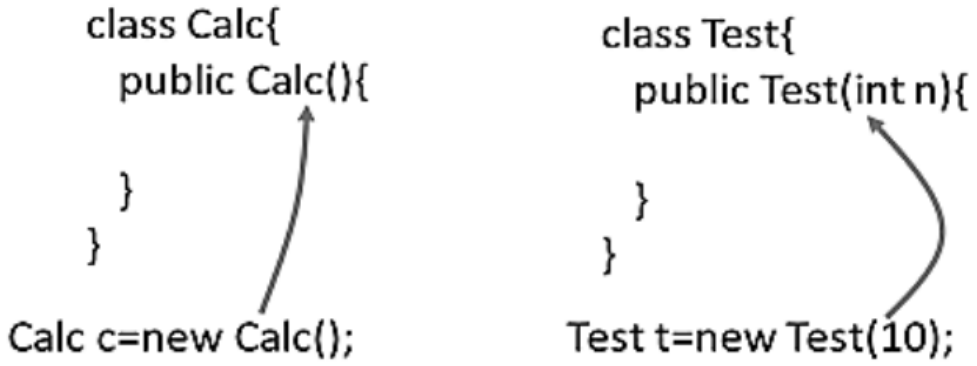


Fig. 6-6. Ejecución de constructor

Constructor por defecto

Toda clase Java debe tener al menos un constructor. Si no se define uno de forma explícita, el compilador añade el llamado **constructor por defecto**, que no tiene parámetros y tampoco ninguna instrucción. Por ejemplo, si definimos la siguiente clase:

```
public class Test{

}
```

El compilador Java la transforma en:

```
public class Test{

    public Test(){}

}
```

Aunque el constructor por defecto esté vacío, su existencia permitirá que se puedan crear objetos de la clase.

Es importante tener en cuenta que si se define explícitamente un constructor, el compilador ya **no crea el constructor por defecto**. Por ejemplo, si definimos la clase Test de esta manera:

```
class Test{

    public Test(int m){}

}
```

El compilador ya no añade el constructor por defecto, por lo que si intentamos crear un objeto de la siguiente forma se producirá un error de compilación:

```
Test t=new Test(); //error de compilación
```

Al disponer solo del constructor con parámetro entero, se debería proporcionar un entero durante la creación del objeto:

```
Test t=new Test(2);//correcto
```

Sobrecarga de constructores

Al igual que los métodos, los constructores se pueden sobrecargar, lo que permite a la clase disponer de varias opciones para inicializar los objetos.

Se siguen las mismas reglas que con la sobrecarga de métodos. La siguiente imagen nos muestra unos ejemplos:

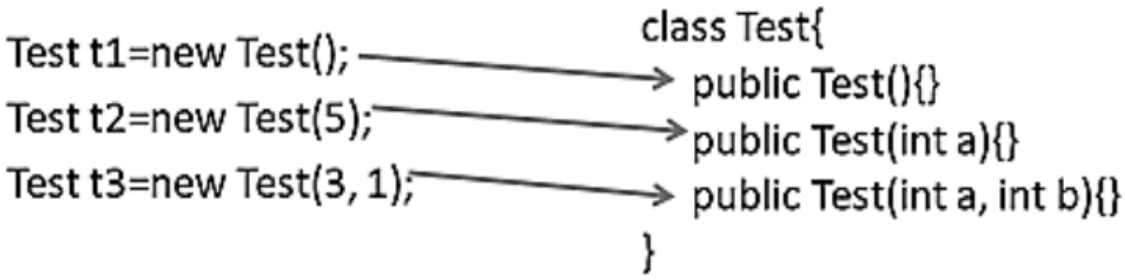


Fig. 6-7. Sobrecarga de constructores

Llamadas a otro constructor

Desde un constructor se puede llamar a otro constructor de la misma clase utilizando la expresión *this(argumentos)*. Esta instrucción **debe ser la primera del constructor** que realiza la llamada.

En la siguiente clase de ejemplo, se realiza una llamada al constructor con parámetro entero desde el constructor sin parámetro. También se intenta una llamada desde el constructor con dos parámetros, pero en este caso se produce un error de compilación por no ser esa instrucción de llamada la primera del constructor:

```
class Test{

     public Test(){

        this(5);//correcto

     }
```

```
        public Test(int a){}

        public Test(int a, int b){

           int s=a+b;

           this(s);//error compilación

        }

   }
```

Bloque de inicialización de instancia

Además de constructores, una clase puede incluir los llamados bloques de inicialización de instancia, que son bloques de código delimitados por llaves y que se **ejecutan cada vez que se crea un objeto** de la clase, antes del constructor.

La siguiente clase de ejemplo define un constructor sin parámetros y un bloque de inicialización de instancia:

```
   class Test{

        {

           System.out.print("bloque");

        }

        public Test(){

            System.out.print(" constructor");

        }

   }
```

Si creamos un objeto de esta clase, veremos que **primero se ejecutará el bloque de inicialización de instancia y después el constructor**. Por ejemplo, al ejecutar la siguiente clase se imprimirá "bloque constructor"

```
class Prueba{

   public static void main(String[] ar){

        Test t1=new Test();

   }

}
```

Los bloques de inicialización de instancia no se utilizan mucho de forma práctica, ya que para eso tenemos los constructores, sin embargo, no sería de extrañar encontrar uno de estos bloques en el código de una pregunta de examen, por ello, es importante comprender su funcionamiento, concretamente, el momento preciso en el que se ejecutan.

MODIFICADORES DE ACCESO

Es el momento de abordar en detalle los modificadores de acceso que podemos utilizar a la hora de definir una clase y los miembros de la misma.

Función y tipos

Los modificadores de acceso determinan la visibilidad de los miembros de una clase, es decir, el lugar desde donde estos pueden ser utilizados. La siguiente tabla indica los tipos de modificadores existentes y la aplicabilidad de cada uno:

	public	**protected**	**(default)**	**private**
clase	SI	NO	SI	NO
atributo	SI	SI	SI	SI
método	SI	SI	SI	SI
constructor	SI	SI	SI	SI

Fig. 6-8. Tabla de ámbitos de acceso

El modificador *protected* lo abordaremos durante el estudio de la herencia; a continuación, analizaremos el uso de los otros tres.

Modificador public

Cuando una clase o miembro de la misma se define como public, significa que puede utilizarse **desde cualquier clase**, tanto de su mismo paquete como de cualquier otro. Por ejemplo, dada la siguiente clase:

```
package p1;

public class Test{

     public Test(int a){}

     public void metodo(){}

}
```

Desde otra clase de otro paquete podrían crear objetos Test y llamar al método:

```
package p2;

import p1.Test;

class Otra{

    void metodoEx(){

          Test t=new Test(10); //correcto

           t.metodo(); //correcto

    }

}
```

Modificador (default)

No es un modificador como tal, sino la ausencia de uno, es decir, el ámbito default, conocido también como ámbito de paquete, es el ámbito que se aplica cuando no se indica ningún modificador. El elemento con este ámbito solo es **accesible desde clases de su mismo paquete**. Por ejemplo, dada la siguiente clase:

```
package p1;

public class Test{

     Test(){}

     public Test(int a){}

     void metodo(){}

}
```

Si definimos otra clase en su mismo paquete, desde ella se podrán crear objetos Test utilizando el constructor sin parámetros y se podrá llamar al método:

```
class Test2{

     void tester(){

          Test t=new Test(); //correcto

          t.metodo(); //correcto

     }

}
```

Sin embargo, dicho constructor y método, al tener ámbito de paquete, solo podrán ser utilizados por clases de ese paquete. Desde clases de otros paquetes no se tendría acceso a los mismos:

```
package p2;

import p1.Test;

class Otra{

    void metodoEx(){

          Test t=new Test(10); //correcto

          t.metodo(); //error de compilación

          Test t2=new Test(); //error de compilación,
```

```
                    //constructor no public
    }
}
```

Modificador private

Este modificador no es aplicable a clases, pero sí a los miembros de la misma. Cuando un miembro se define como privado, **solo es accesible desde el interior de la clase**.

Este modificador suele utilizarse con los atributos de la clase para aplicar los principios de la encapsulación y evitar que dichos atributos sean accesibles directamente desde el exterior:

```
public class Mesa{
    private int largo;
    private int ancho;
    private String color;
    :
}
class Otra{
    void metodo(){
        Mesa m=new Mesa();
        m.largo=2; //error compilación
    }
}
```

Posiblemente nos preguntemos qué sentido puede tener la existencia de constructores private. El motivo es evitar que puedan crearse objetos de la clase desde fuera de esta a través del operador new. Por ejemplo, dada la siguiente clase:

```
public class Test{

    private Test(){

    }

}
```

Al disponer de un constructor explícito, **el compilador ya no añade el constructor sin parámetros**, que como sabemos, es público y sin parámetros. Por tanto, la clase solo contaría con este constructor privado, impidiendo que se puedan crear objetos de ella externamente:

```
Test t=new Test(); //error de compilación
```

Normalmente, este tipo de clases suelen proporcionar una forma alternativa para la creación de los objetos, como por ejemplo, un método estático:

```
public class Test{

    private Test(){

    }

    public static Test getTest(){

          return new Test();

    }

}
```

De esta forma, si desde fuera de la clase se quiere obtener un objeto de la misma debería hacerse de la siguiente forma:

```
Test t=Test.getTest();
```

Esta forma de obtener objetos de una clase permite que la propia clase pueda controlar los objetos que se crean. Por ejemplo, podría utilizarse para definir **un Singleton.**

Singleton

Un singleton es una clase que solo permite **crear una única instancia de la misma**. Una implementación de la clase Test como singleton sería:

```
public class Test{

    private static Test ts;

    private Test(){

    }

    public static Test getTest(){

          //solo crea la instancia si no existe

          //previamente

          if(ts==null){

                ts=new Test();

          }

          return ts;

    }

}
```

Si desde fuera de esta clase se hacen varias llamadas a getTest(), se estará utilizando la misma instancia:

```
Test t1=Test.getTest();

Test t2=Test.getTest();
```

Ambas variables, t1 y t2, apuntan al mismo objeto.

ENCAPSULACIÓN

La encapsulación es una de las características de la programación orientada a objetos que suele ser objetivo directo de alguna pregunta de examen. Veamos en qué consiste.

Definición

Se trata de un principio que consiste en mantener como privados los atributos que definen características de un objeto, proporcionando acceso a los mismos a través de métodos y constructores.

El objetivo es que los **atributos no sean accesibles directamente desde el exterior** y evitar así que puedan ser "corrompidos" con valores no permitidos o incongruentes.

Por ejemplo, supongamos que tenemos la siguiente clase:

```
public class Mesa{

    public int alto;

    public int ancho;

}
```

En estas circunstancias no se podría evitar que desde otra clase se crease un objeto Mesa y se asignasen valores inadecuados a sus atributos:

```
class Otra{

    void metodo(){

        Mesa m=new Mesa();

        m.largo=-2; //no tendría sentido

    }

}
```

Aplicación de la encapsulación

Para evitar el problema anterior, los atributos deberían definirse como privados y proporcionar el acceso a los mismos de forma controlada a través de los llamados métodos *setter* y *getter*. Mediante el constructor, se permitiría inicializar los atributos durante la creación del objeto. Así es como redefiniríamos la clase Mesa aplicando el principio de la encapsulación:

```
public class Mesa{
    private int largo;
    private int ancho;
    public Mesa(int largo, int ancho){
     if(largo>0) this.largo=largo;
     if(ancho>0) this.ancho=ancho;
    }
    public void setLargo(int largo){
    if(largo>0) this.largo=largo;
    }
    public int getLargo(){
    return largo;
    }
    public void setAncho(int ancho){
    if(ancho>0) this.ancho = ancho;
    }
    public int getAncho(){
    return ancho;
    }
}
```

PREGUNTAS TIPO EXAMEN

Pregunta 1

Given the following:

```
public class Test {
  static int sum(Integer a, int b){return a+b;}
  static long sum(Long x, int y){return x+y+10;}
  static double sum(int n, double r){return n+r;}
  public static void main (String []args) {
        System.out.println(sum(3,2));
  }
}
```

What is the result?

A. The output is 5

B. The output is 15

C. The output is 5.0

C. Compilation will fail

D. It will throw an Exception

Pregunta 2

Given the following:

```
class Number{
        private int n;
        public void setN(int a){
                n=a;
```

```
        }
        public int getN(){
                return n;
        }
    }
    public class Test {
     public static void main (String []args) {
        Number num=new Number();
        processing(num);
        System.out.print(num.getN());
     }
     static void processing(Number x){
        x.setN(x.getN()+5);
        System.out.print(x.getN());
     }
    }
```

What is the result?

A. The output is 0

B. The output is 5

C. The output is 55

D. Compilation fails

Pregunta 3

Which of the following methods can't be in the same class with: public void mt(long r)? (choose 2)

A. void mt(int s)

B. int mt(long a)

C. int mt()

D. public void mt(Long a)

E. void mt(long v)

Pregunta 4

Given the following

```
class Test{
        int a;
        static int b;
        static{
                b++;
        }
        Test(){
                while(a<5){
                        b++;
                        a++;
                }
        }
        public static void main(String[] args){
                Test t1=new Test();
                Test t2=new Test();
                System.out.println(t1.a+":"+t2.b);
        }
```

```
}
```

Which is the result?

A. 10:10

B. 5:10

C. 5:11

D. 11:5

E. Compilation fails

Pregunta 5

Given the following:

```
public class Test {
   public Test(){
         System.out.println("No params");
   }
   public void Test(int j){
         System.out.println("Param "+j);
   }
   public static void main(String[] args) {
         Test t=new Test(3);
   }
}
```

Which is the result?

A. Param 3

B. No params

C. Compilation fails

D. Exception

Pregunta 6

Given the following:

```
class Vehicle {
        String name;
        void setName (String name) {
                this.name = name;
        }
        String getName() {
                return name;
        }
}
```

Which action would apply encapsulation in this class?

A. Define name variable as public

B. Define name variable as private

C. Define Vehicle class as public

D. Define setter and getter methods as public

E. Define setter and getter methods as private

Pregunta 7

Given:

C1.java

```
package p1;
class C1{
        int p;
        private int k;
        public int s;
}
```

C2.java

```
package p2;
import p1.C1;
public class C2{
        public static void main(String[] args){
                C1 obj=new C1();
        }
}
```

Which statement is true?

A. Both p and s are accesible by obj

B. Only s is accesible by obj

C. None of the variables are accesible by obj

D. Compilation fails

Pregunta 8

Which are true? (choose 2)

A. Default constructor should be always there for any class.

B. Default constructor must have parameters

C. When defining our own constructor we can’t use any access modifier.

D. A constructor should not have a return type.

E. We can have more than one constructor in a class.

SOLUCIONES

1. C. Al aplicar las reglas definidas para llamar a métodos sobrecargados, encontramos una coincidencia exacta del primer parámetro en el tercero de los métodos y del segundo parámetro en el primero y segundo de los métodos. Pero el segundo de los métodos debemos descartarlo, porque para poder pasarla al 3, se debería realizar una promoción de tipos más un autoboxing, y ambas cosas no son posibles. De los métodos que quedan, en el primero habría que hacer un autoboxing y en el tercero una promoción de tipos que, según las reglas definidas, tiene prioridad sobre el autoboxing, por tanto, se ejecuta el tercero de los métodos. Como el tipo de devolución es doble, se muestra como decimal.

2. C. En la llamada a processing, el objeto Number es modificado y se le asigna el valor 5, que es imprimido. Cuando se llama de nuevo a getN() con la variable num, se vuelve a obtener un 5 dado que ambas variables apuntan al mismo objeto.

3. B y E. Al coincidir la lista de parámetros con la del método indicado, no sería un caso válido de sobrecarga y no podrían estar en la misma clase.

4.C. La variable b es estática y, por tanto, compartida por los dos objetos creados de la clase Test. El bloque estático se ejecuta una sola vez, por tanto, al ejecutar el constructor del primer objeto esta variable vale 1, pero cuando se ejecuta el constructor del segundo, la variable vale 6, por lo que su valor final será 11. Como la variable a no es static, cada objeto tiene su copia y cada constructor deja esta variable al valor 5.

5. C. La clase no cuenta con ningún constructor que reciba un entero, por tanto, se producirá un error de compilación en la creación del objeto Test. El bloque public void Test(int j){...} es un método, no un constructor.

6. B. Se corresponde al principal principio de la encapsulación. En ese sentido, la A no es correcta porque expresa todo lo contrario, C no es correcta porque el ámbito de la clase es irrelevante para la encapsulación, D no es correcta porque setter y getter no tienen que ser obligatoriamente public, pueden tener ámbito de paquete como en este caso. Lo que no pueden es ser privados, por lo que E es incorrecta.

7. D. Se produce un error de compilación en la instrucción C1 obj=new C1();, pues al ser el constructor C1 de ámbito de paquete, no es posible crear objetos de esta clase usando este constructor desde fuera de su paquete.

8. D y E. La A no es correcta porque no todas las clases tienen constructor por defecto. La B no es correcta porque un constructor por defecto no tiene parámetros y la C tampoco es cierta porque, como hemos visto, los constructores pueden tener cualquier modificador de acceso, incluso private.

7 Herencia

CONCEPTO DE HERENCIA Y PROPIEDADES

La herencia es posiblemente la característica más importante de la orientación a objetos. Además de la implicación de su uso directo, hay otras muchas características de la orientación a objetos, como la sobrescritura o el polimorfismo, que se basan en la herencia. Vamos primeramente a analizar la herencia como tal y sus propiedades.

Definición

La herencia es una característica de la orientación a objetos que permite crear nuevas clases a partir de clases ya existentes, de forma que la nueva clase adquiera (herede) los miembros de la ya existente.

A la clase "padre" se le conoce como **superclase**, mientras que a la hija se le llama **subclase**. Gráficamente, la herencia se representa con una flecha que sale de la subclase y apunta a la superclase:

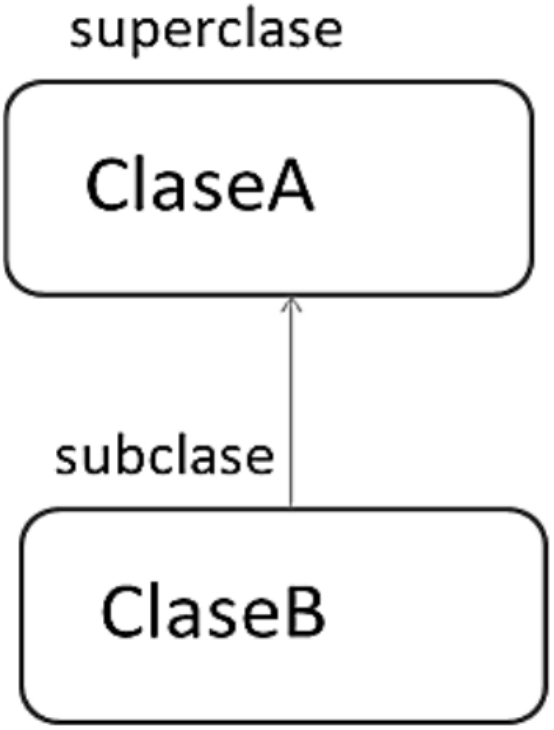


Fig. 7-1. Representación gráfica de la herencia

Para crear una clase que herede otra clase, debemos utilizar la palabra **extends**, seguido del nombre de la clase a heredar, durante la definición de la subclase:

```
class Clase1{

   public void metodo(){}

}

class Clase2 extends Clase1{

   //automáticamente adquiere metodo()

}
```

Si desde otra clase se crea un objeto de Clase2, podría llamar al método heredado:

```
Clase2 c=new Clase2();

c.metodo();
```

Consideraciones

De cara a utilizar la herencia en Java, hay que tener en cuenta las siguientes consideraciones:

- Una clase **solo puede heredar otra clase**. La herencia múltiple, que consiste en la posibilidad de que una clase pueda heredar dos o más clases directamente, no está permitida en Java. Sí es posible que una clase que hereda otra, pueda a su vez ser heredada por una tercera y así hasta n niveles:

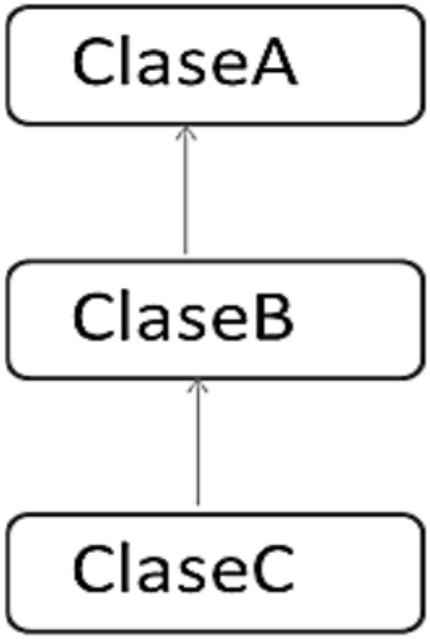


Fig. 7-2. Herencia multinivel

- Una misma clase puede ser heredada por varias clases.
- Los miembros privados de la superclase no son accesibles directamente desde la subclase. Aunque son heredados, desde una subclase no podemos llamar directamente a los miembros privados de la superclase.

Clases finales

Una clase final es aquella que no puede ser heredada. Para definir una clase como final, se utiliza el modificador *final* delante de class:

```
final class Clase1{

}
```

Si se intenta crear una clase que herede Clase1, se producirá un error de compilación:

```
class Clase2 extends Clase1{ //error de compilación


}
```

En Java estándar existen varias clases de uso habitual que son finales. Por ejemplo, la clase String y todas las de envoltorio son clases finales.

Relación "es un"

Entre una clase y su superclase existe lo que se conoce como una relación "es un", puesto que un objeto de la subclase también "es un" objeto de la superclase. Fijémonos en los siguientes ejemplos de herencia:

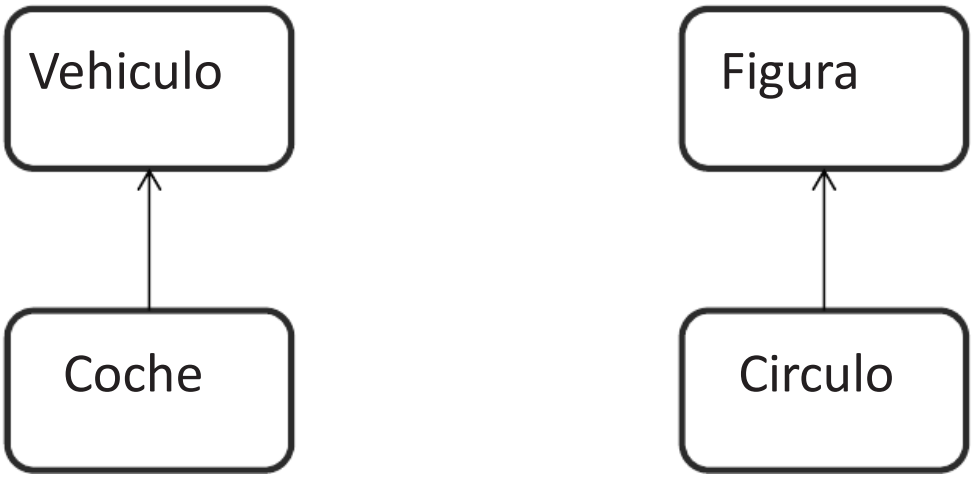


Fig. 7-3. Ejemplos de herencia

Podemos decir que un Coche *es un* Vehiculo y que un Circulo *es una* Figura.

Esta relación puede utilizarse para detectar casos incorrectos de herencia. Por ejemplo, la siguiente relación de clases no sería de herencia:

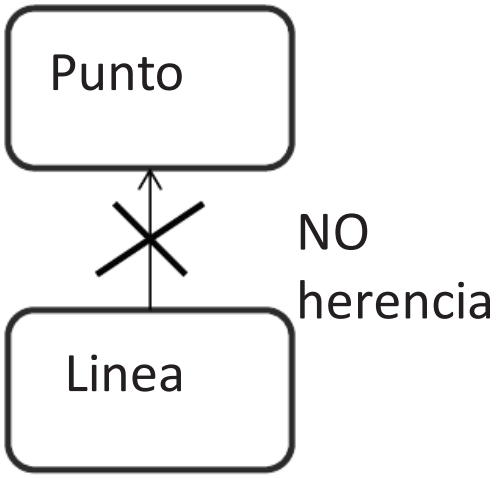


Fig. 7-4. Caso incorrecto de herencia

Y es que una Linea NO es un Punto. Sería otro tipo de relación, pero no herencia.

Herencia de Object

Todas las clases Java heredan directa o indirectamente la clase Object, que se encuentra en el paquete java.lang.

Si definimos una clase sin indicar que hereda otra clase, **implícitamente estamos heredando Object**. Es decir, si definimos una clase de la siguiente manera:

```
class Test{


}
```

Es equivalente a haberla definido así:

```
class Test extends Object{


}
```

Si nuestra clase hereda explícitamente otra clase, ya no estaría heredando directamente Object, pero la superclase sí, por lo que al final, todas las clases heredan a Object.

Por tanto, los métodos de Object están presentes en todas las clases Java. Entre estos métodos están *toString(), equals()* y *hashCode()*.

CONSTRUCTORES EN LA HERENCIA

Cuando creamos un objeto de una clase, además de ejecutarse el constructor de dicha clase se ejecuta también el de la superclase, lo que tiene implicaciones a la hora de definir una jerarquía de clases.

Vamos a analizar el flujo de ejecución de constructores en la herencia y cómo modificar el comportamiento por defecto.

Llamada a constructor de la superclase

Toda clase Java incluye de forma implícita, como primera línea de código en sus constructores, la instrucción *super();,* que es una llamada al constructor sin parámetros de la superclase.

Por ejemplo, si tenemos esta clase:

```
class Clase1 extends Otraclase{

    public Clase1(int a){

        System.out.println(a);

    }

}
```

Esto será equivalente a:

```
class Clase1 extends Otraclase{

    public Clase1(int a){

        super();
```

```
            System.out.println(a);
        }
    }
```

Donde la instrucción *super()* realiza una llamada al constructor sin parámetros de la superclase.

Esta instrucción **la incluyen todos los constructores de forma implícita**, incluido el constructor por defecto generado por el compilador.

Veamos un ejemplo. Dada la siguiente jerarquía de clases:

```
class Clase1{
    Clase1(){
            System.out.println("Clase1");
    }
}
class Clase2 extends Clase1{
    Clase2(){
            System.out.println("Clase2");
    }
}
```

Si creamos un objeto de Clase2:

```
Clase2 c2=new Clase2();
```

Se mostrará por pantalla:

Clase1

Clase2

Como vemos, el orden de ejecución de los constructores es inverso al de herencia.

Si la superclase no dispusiera de un constructor sin parámetros, se produciría un error de compilación. Por ejemplo, dadas las siguientes clases:

```
class Clase1{

    Clase1(int n){}

}

class Clase2 extends Clase1{ //error de compilación



}
```

Se producirá un **error de compilación en la definición de Clase2**. El motivo es que su constructor, que en este caso es el constructor por defecto incluido por el compilador, incluye una llamada al constructor sin parámetros de la superclase. Como Clase1 no dispone de constructor sin parámetros (al tener un constructor explícito ya no se incluye él por defecto), se produce un error de compilación.

Llamada a un constructor con parámetros

Es posible llamar, desde un constructor de la subclase, a otro constructor de la superclase que no sea el constructor sin parámetros. Para hacer esto, debemos utilizar de forma explícita la instrucción:

super(argumentos);

La lista de argumentos deberá coincidir con los parámetros del constructor que se quiere llamar. Es importante indicar que esta instrucción **debe ser la primera del constructor**. Aquí tenemos un ejemplo:

```
class Clase1{

    Clase1(int a){}

}

class Clase2 extends Clase1{
```

```
    Clase2(String n){
        super(10); //Llama al constructor de Clase1
        System.out.println(n);
     }
}
```

Según indicamos durante la explicación de los constructores, es posible llamar a un constructor de la propia clase desde otro utilizando la instrucción:

this(argumentos);

En este caso, el constructor que incluya la instrucción anterior no podrá llamar también al constructor de la superclase:

```
class Clase1{
    Clase1(int a){}
}
class Clase2 extends Clase1{
    Clase2(int x){
        super(x);
        System.out.println("C2");
     }
     //el siguiente constructor no incluye llamada a
    //constructor de superclase, ni explícita ni implícita
     Clase2(){
         this(10); //llamada al otro constructor
     }
```

```
    Clase2(String s){
        super(2);
        this(); //error de compilación!!
    }
}
```

Fijémonos en el error de compilación del tercer constructor de Clase2, el cual es debido a que no pueden utilizarse en un mismo constructor las llamadas al constructor de la superclase (super) y al de la propia clase (this).

SOBRESCRITURA DE MÉTODOS

Una de las consecuencias más interesantes de la herencia es la sobrescritura de métodos. Vamos a analizar en qué consiste, las reglas de aplicación y también veremos algunos ejemplos de uso.

Definición de sobrescritura

La sobrescritura de métodos consiste en **volver a redefinir en la subclase un método que ha sido heredado de la superclase**. De esta manera, el método heredado queda anulado y en la subclase solo quedaría la nueva versión del método.

Por ejemplo, dada la siguiente clase:

```
class Clase1{
    public void test(){
        System.out.println("uno");
    }
}
```

Si queremos crear una subclase de Clase1 que redefina el método test para que muestre otro mensaje, sería:

```
class Clase2 extends Clase1{

    //el método vuelve a definirse

    public void test(){

        System.out.println("dos");

    }

}
```

Si creamos un objeto de Clase2 y llamamos a test(), se ejecutará la nueva versión:

```
Clase2 c=new Clase2();

c.test(); //imprime dos
```

Anotación @Override

Cuando sobrescribimos un método debemos respetar el formato del método original, pero si no lo hacemos, el compilador no generará ningún error, simplemente dispondremos de dos métodos, el heredado y el nuevo. Observemos las siguientes clases:

```
class Clase1{

    public void test(){

        System.out.println("uno");

    }

}

class Clase2 extends Clase1{

    public void Test(){

        System.out.println("dos");

    }

}
```

Al intentar sobrescribir el método test(), hemos creado uno nuevo llamado Test(), como Java es *case sensitive*, ahora nuestra Clase2 cuenta con dos métodos. Si queremos evitar que esto suceda y que el compilador nos avise si la sobrescritura no es correcta, debemos utilizar la anotación @Override a la hora de sobrescribir un método:

```
class Clase1{

    public void test(){

        System.out.println("uno");

    }

}

class Clase2 extends Clase1{

    @Override

    public void Test(){

        System.out.println("dos");

    }

}
```

En este caso, se producirá un **error de compilación** en Clase2 durante la sobrescritura del método *test()*, ya que hemos incumplido las reglas que se deben seguir para la sobrescritura de un método.

Reglas de la sobrescritura

A la hora de sobrescribir un método debemos de tener presentes una serie de reglas. Es muy probable que encontremos alguna pregunta de examen en la que tengamos que aplicar estas reglas para determinar si una posible sobrescritura de método se ha realizado correctamente o, simplemente, si determinado código compila. Veamos estas reglas.

Para sobrescribir correctamente un método debemos de tener en cuenta:

- El nombre de método y lista de parámetros debe ser idéntico al del original.

- El ámbito del nuevo método debe ser igual o menos restrictivo que el del original.

- El tipo de devolución debe ser igual o un subtipo del original.

- La nueva versión del método no debe propagar excepciones tipo *checked* que no estén definidas en el original, aunque pueden ser subtipos de la declarada en el método de la superclase o, incluso, no declarar ninguna. Esta restricción NO afecta a las excepciones Runtime. Hablaremos de las excepciones en el próximo capítulo.

A continuación, vamos a ver algunos ejemplos de sobrescritura de métodos en donde podemos ver cómo se aplican las reglas anteriores. Por ejemplo, dada la siguiente clase:

```
class Clase1{

    public Object test(){ }

}
```

Podemos sobrescribir el método test() en una subclase de Clase1 de la siguiente manera:

```
class Clase2 extends Clase1{

    @Override

    public String test(){ }

}
```

Se trata de un caso de sobrescritura válido, pues el nuevo método se llama igual que el original, tiene la misma lista de parámetros, el ámbito es el mismo y el tipo de devolución (String) es un subtipo del original (Object).

Fijémonos ahora en esta otra definición de Clase1:

```
class Clase1{
```

```
     void test(){ }

}
```

La siguiente sobrescritura de *test()* también sería válida pues, además de respetar nombre, parámetros y tipo de devolución, el ámbito es menos restrictivo (public) que el del método original que es *default*:

```
class Clase2 extends Clase1{

    @Override

    public void test(){ }

}
```

Por último, supongamos que tenemos la siguiente definición de Clase1:

```
class Clase1{

     void test() throws IOException{ }

}
```

Para sobrescribir correctamente test en una subclase, **el nuevo método deberá declarar la misma excepción, un subtipo de esta o ninguna**:

```
class Clase2 extends Clase1{

    @Override

    public void test() throws FileNotFoundException{ }

}
```

A continuación, veremos algunos casos de subrescritura incorrecta. Si tenemos la siguiente clase:

```
class Clase1{

    public void test(){ }

}
```

Las siguientes subclases de Clase1 sobrescribirían de forma incorrecta el método *test()*, produciéndose en ambos casos un error de compilación al haber indicado la anotación @Override:

```
class Clase2 extends Clase1{

    @Override

    public String test(){ } //el tipo debería ser void

}

class Clase3 extends Clase1{

    @Override

       void test(){ } //no puede tener ámbito inferior

}
```

Tampoco la siguiente clase, Clase3, sobrescribiría de forma correcta el método *test()* de Clase1 al declarar una excepción no existente en el método original:

```
class Clase3 extends Clase1{

    @Override

    //error de compilación

     public void test() throws SQLException{ }

}
```

Sobrescritura vs sobrecarga

Es común confundir la sobrescritura de métodos con la sobrecarga cuando hay herencia entre clases. Podemos encontrar alguna pregunta de examen que vaya en esa línea.

Por ejemplo, dada la clase:

```
class Clase1{
```

```
    public void test(){

    }

}
```

La siguiente implementación de Clase2 **no sobrescribe test(), sino que lo sobrecarga**, por tanto, la subclase tendría dos métodos test().

```
class Clase2 extends Clase1{

    //el método incluye un parámetro

    public void test(int s){

    }

}
```

La sobrecarga de un método en la subclase, lógicamente, no provoca error de compilación, salvo que se hubiera anotado con @Override.

El modificador de acceso protected

Como indicamos durante el estudio de los modificadores de acceso, *protected* puede utilizarse en la declaración de atributos, constructores y métodos. Si uno de estos elementos se declara como *protected*, significa que es accesible desde cualquier clase de su mismo paquete y también desde sus subclases, independientemente de dónde estas se encuentren definidas.

Por tanto, el modificador *protected* establece una visibilidad a los miembros de una clase que es superior a la *default* (ámbito de paquete), pero inferior a la *public*.

El acceso desde una subclase a un miembro *protected* de la superclase se debe hacer siempre dentro del contexto de la herencia. Veamos un ejemplo, tenemos la siguiente clase:

```
package p1;

public class Prueba{

    protected int k=2;
```

```
    protected void metodo(){}

}
```

Si definimos una subclase de Prueba en otro paquete, desde el interior de esta clase se tendrá acceso a k y metodo():

```
package p2;

public class Test extends Prueba{

    public void nuevoMetodo(){

          this.k=10;//correcto

          this.metodo();//correcto

    }

}
```

Sin embargo, si desde una subclase que se encuentre en otro paquete distinto a p1 **creamos un objeto de la clase** Prueba, **no tendremos acceso a los miembros protegidos** a través de este objeto. Esto es lo que significa que el acceso solo sea a través del contexto de la herencia:

```
package p3;

public class Nueva extends Prueba{

    public void miMetodo(){

          Prueba p=new Prueba();

          p.k=3; //error de compilación

          p.metodo(); //error de compilación

    }

}
```

TIPO DE OBJETO Y TIPO DE REFERENCIA

Otra consecuencia de la herencia es la posibilidad de asignar una referencia a un objeto de un tipo en una variable del tipo de su superclase. Veamos cómo realizar esta operación y las consecuencias de ello.

Asignación de referencias a objetos en tipos de superclase

Por ejemplo, podríamos hacer lo siguiente:

```
Object ob=new String("hello");
```

El tipo de la referencia (Object) es superclase del tipo del objeto (String). Esto puede aplicarse a varios niveles, por ejemplo, si tenemos las siguientes clases:

```
class Clase2 extends Clase1{}

class Clase3 extends Clase2{}
```

Podríamos hacer:

```
Clase1 c=new Clase3();
```

Llamadas a métodos comunes

Con esta referencia a objeto almacenada en una variable de la superclase, se puede llamar a métodos del objeto, pero **SOLO a aquellos que han sido heredados o sobrescritos en la subclase**. Dicho de otra manera, solo podemos llamar a los métodos comunes en ambas clases.

Por ejemplo, dada la siguiente instrucción:

```
Object ob=new String("hello");
```

Podríamos utilizar la variable ob para llamar a los métodos del objeto String, pero solo a los que herede de Object:

```
System.out.println(ob.toString()); //muestra hello

if(ob.equals("hello")){} //también correcto, true
```

Pero la siguiente instrucción provocaría un error de compilación, ya que el método ***length()*** **está definido en String, no en Object**:

```
System.out.println(ob.length()); //error de compilación
```

Hay que tener en cuenta, que si aplicamos el operador instanceof sobre la variable ob y los tipos String y Object, en **ambos casos nos indicará *true***:

```
System.out.println(ob instanceof String); //true

System.out.println(ob instanceof Object); //true
```

Casting entre tipos objeto

Tenemos una referencia a un objeto en una variable de su superclase, para almacenar esa referencia en una variable del tipo explícito del objeto debemos efectuar un *casting* o conversión explícita:

```
String s=(String)ob;
```

Sobre la variable s sí podríamos llamar a los métodos exclusivos de String.

El compilador permite hacer *casting* de una referencia de un tipo a cualquier tipo de sus subclases, pero **si el objeto no es de ese tipo** se producirá una **ClassCastException**:

```
Integer r=(Integer)ob;
```

La instrucción anterior compila correctamente, pero al ejecutarla se producirá una excepción **ClassCastException, pues el tipo de objeto referenciado por ob es String**, no Integer.

CLASES ABSTRACTAS Y POLIMORFISMO

El polimorfismo es otro de los conceptos clave de la programación orientada a objetos, normalmente, aparece vinculado a las clases abstractas por lo que será el primer punto que abordemos en este apartado. También es aplicable el polimorfismo con interfaces, que las estudiaremos en el siguiente apartado.

Clases abstractas

Es una clase que cuenta, al menos, con un método abstracto. Un método abstracto es aquel que está declarado en la clase pero no implementado. Su objetivo es proporcionar un formato de método determinado, para que sean las subclases las que se encarguen de darle cuerpo a través de la sobrescritura.

Tanto la declaración de la clase como la de los métodos abstractos debe realizarse con el modificador *abstract*:

```
abstract class Clase1{

    public abstract int calculo();

}
```

Como vemos, el método abstracto calculo() se declara con un punto y coma al final, sin cuerpo alguno.

CONSIDERACIONES SOBRE LAS CLASES ABSTRACTAS

De cara no solo a posibles preguntas de examen, sino también para el correcto uso de las mismas, debemos de tener en cuenta las siguientes consideraciones o características sobre las clases abstractas:

- No es posible crear objetos de una clase abstracta. Por ejemplo, dada la siguiente clase abstracta anterior, la siguiente instrucción provocaría un error de compilación:

  ```
  Clase1 c=new Clase1();
  ```

- Además de métodos abstractos, las clases abstractas pueden incluir atributos, constructores y métodos estándares. Aunque no se puedan crear objetos de una clase abstracta, tiene sentido que puedan contar con constructores, los cuales se ejecutarán durante la herencia de la clase abstracta.

- Una clase que herede una clase abstracta está obligada a sobrescribir los métodos abstractos heredados (o declararse también como *abstract*).

- Una clase puede declararse como *abstract* aunque no tenga métodos abstractos:

```
abstract class MiClase{

    public void imprimir(){}

}
```

La clase anterior solo cuenta con métodos estándares, ninguno abstracto, pero puede ser declarada como abstract. Eso sí, no será posible crear objetos de ella.

- El objetivo de los métodos abstractos es forzar a que todas las subclases tengan el mismo formato de método. Al definir un **método como abstracto, obligamos a todas las subclases a que lo sobrescriban** y, por tanto, respeten el mismo formato definido en la clase padre.

EJEMPLOS

Veamos un ejemplo de definición de clases abstractas. Supongamos que queremos definir una clase que represente una figura geométrica, esta clase dispondrá de atributos (por ejemplo, color), métodos no abstractos que permitan acceder a dichos atributos, y también un método abstracto superficie().

Toda figura geométrica tiene que tener una superficie, pero la manera de calcularla depende de las subclases específicas de Figura. Aquí tenemos un ejemplo de cómo quedaría definida la clase Figura:

```
abstract class Figura{

     private String color;

     public Figura(String color){

    this.color=color;

     }

     //aquí irían métodos getter/setter

     :

     public abstract double area();
}
```

La clase Figura define cómo quiere que sea este método en todas las clases de Figura, siendo responsabilidad de cada una de las subclases su definición.

A continuación, presentamos dos posibles subclases de Figura, Triangulo y Circulo:

```
class Circulo extends Figura{
     private int radio;
     public Circulo(String color, int radio){
          super(color);
          this.radio=radio;
     }
     public double area(){
          return Math.PI*radio*radio;
     }
}
class Triangulo extends Figura{
     private int base,altura;
     public Triangulo(String color, int base, int altura){
          super(color);
          this.base=base;
          this.altura=altura;
     }
     public double area(){
          return base*altura/2;
     }
}
```

El hecho de que el formato del método sea el mismo en todas las subclases es importante porque nos permitirá aplicar el polimorfismo.

Polimorfismo

El polimorfismo se basa en una característica que hemos estudiado en uno de los apartados anteriores, y que consiste en la asignación de referencias a objetos en variables de su superclase (abstractas o no).

Por ejemplo, en el ejemplo de las figuras geométricas, podríamos asignar a una variable de tipo Figura un objeto Triangulo:

```
Figura f=new Triangulo(..);
```

O también un objeto Circulo:

```
Figura f=new Circulo(..);
```

La ventaja de esto lo tenemos en la definición de polimorfismo: consiste en poder utilizar una **misma expresión para llamar a diferentes versiones de un mismo método**. Por ejemplo, para llamar al método *superfice()* de Triangulo podemos hacer:

```
f=new Triangulo(..);

f.superfice();
```

Y para llamar a superficie de Circulo:

```
f=new Circulo(..);

f.superfice();
```

Como vemos, la misma instrucción *f.superfice();*, se puede utilizar para llamar a dos versiones diferentes de un mismo método. Y, ¿qué ventaja tiene esto?, principalmente, **reutilización de código** ya que la misma o mismas instrucciones se pueden emplear con varios objetos.

En el siguiente sencillo programa vemos un uso muy concreto del polimorfismo con las clases estudiadas anteriormente, donde el método *mostrarDatos()*, utiliza el polimorfismo para llamar a los métodos *getColor()* y *superfice()* a través de una variable Figura y que pueden ser aplicados sobre cualquier subclase de Figura:

```
public class TestPolimorfismo {

    public static void main(String[] args) {

          mostrarDatosFiguras(new Triangulo("amarillo",4,8));

          mostrarDatosFiguras(new Circulo("azul",3));

    }

    private static void mostrarDatosFiguras(Figura f){

          System.out.println("Color "+f.getColor());

          System.out.println("Superficie "+f.superfice());

    }

}
```

Además de la reutilización de código, otros importantes beneficios que proporciona el polimorfismo son la flexibilidad a la hora de poder trabajar con superclases en lugar de clases específicas, y el dinamismo, pues el tipo de objeto sobre el que se aplicarán los métodos se determina dinámicamente en tiempo de ejecución.

Métodos abstractos vs métodos finales

Lo contrario a un método abstracto es un método final. Un método final es aquel que no puede ser sobrescrito. Para definir un método como final, empleamos el modificador *final* delante del tipo:

```
class Clase1{

    public final int calculo(){}

}
```

Si intentáramos sobrescribir este método en una subclase de Clase1 se produciría un error de compilación:

```
class Clase2 extends Clase1{

    public int calculo{} //error de compilación

}
```

INTERFACES EN JAVA

Además de las clases, otro de los componentes que se utilizan en el desarrollo de las aplicaciones Java son las interfaces. Gracias al polimorfismo, la programación basada en interfaces está muy extendida y podemos encontrar numerosos ejemplos de ello, tanto en Java estándar como en enterprise. Además, desde la versión Java 8 las interfaces han incorporado importantes novedades. Vamos a estudiar en este apartado este importante elemento de programación.

Concepto

Una interfaz es un **conjunto de métodos abstractos**. Su misión es definir el formato que tienen que tener ciertos métodos, de modo que las clases que los van a implementar se ajusten todas al mismo formato.

Además de métodos abstractos, una interfaz puede contener constantes.

Definición de una interfaz

Una interfaz se define con la palabra *interface*, seguido del nombre de la interfaz. Al igual que las clases, se definen en archivos .java y siguen las mismas normas que estas a la hora de nombrarlas y definir los modificadores de acceso, es decir, una interfaz solo puede declararse con ámbito public o por defecto y, en caso de que sea pública, se deberá llamar igual que el archivo .java:

```
Test.java

public interface Test{

:

}
```

MÉTODOS DE UNA INTERFAZ

Todos los métodos de la interfaz deben ser **obligatoriamente públicos y abstractos**, por ello, los modificadores public y abstract resultan redundantes, por lo que pueden omitirse:

```
public interface Test{

    void metodo1(int x);

    int metodo2(String s);

}
```

CONSTANTES

Como dijimos antes, una interfaz puede definir también constantes. Estas deben ser **obligatoriamente públicas y estáticas**, por lo que los modificadores public, static y final resultan redundantes y pueden omitirse:

```
public interface Test{

    int CONV=8.75;

    void metodo1(int x);

    int metodo2(String s);

}
```

Implementación de una interfaz

Cuando una clase va a definir código para los métodos de una interfaz, tiene que implementarla, para lo cual se utilizará la palabra *implements*:

```
class ClasePrueba implements Test{

    :

}
```

La clase está obligada a implementar (cuando es una interfaz se dice implementar, no sobrescribir) todos los métodos declarados en la interfaz. Si por algún motivo la clase no va a implementar alguno de los métodos, deberá definirse como abstracta pues sería como si heredase el método abstracto que no ha implementado.

A la hora de implementar los métodos en la clase se siguen las mismas normas que en la sobrescritura, esto implica que **deberán definirse como public**:

```
class ClasePrueba implements Test{

    public void metodo1(int x){...}

    public int metodo2(String s){...}

}
```

Si la interfaz contiene constantes, se podrá acceder a ellas con el nombre de la interfaz:

```
Test.CONV
```

Con el nombre de la clase que la implementa:

```
ClasePrueba.CONV
```

Y también con una referencia a los objetos de las clases que la implementan:

```
ClasePrueba cp=new ClasePrueba();

cp.CONV
```

IMPLEMENTACIÓN MÚLTIPLE

Las interfaces ofrecen una serie de flexibilidades a la hora de trabajar con métodos abstractos que no tienen las clases abstractas:

Una clase puede implementar más de una interfaz. Es lo que se conoce como **implementación múltiple:**

```
class ClaseN implements Inter1, Inter2{

    :

}
```

Lógicamente, está obligada a implementar los métodos de todas las interfaces.

Una clase también puede heredar otra clase y, al mismo tiempo, implementar una o varias interfaces:

```
class ClaseN extends ClaseA implements Inter1, Inter2{

    :

}
```

En estos casos en los que la clase hereda otra clase e implementa una o varias interfaces, resaltar que primero se debe indicar la clase que hereda y después las interfaces que implementa. Si se indica al revés no compilará:

```
class Clase2 implements Test extends Clase1{ //error

}
```

Referencias a objetos en una interfaz

En una variable interfaz se puede almacenar una referencia a cualquier objeto de las clases que la implementan. Con esta referencia se podría llamar solamente a las implementaciones de los métodos declarados en la de la interfaz:

```
interface Test{

    int CONV=8.75;

    void metodo1(int x);

    int metodo2(String s);

}

class ClasePrueba implements Test{

    public void metodo1(int x){...}

    public int metodo2(String s){...}

    public void metodoPropio(){...}
```

```
}
class Ejemplo{
    public static void main(String[] args){
          Test ts=new ClasePrueba();
          ts.metodo1(10);
          ts.metodo2("hello");
          ts.metodoPropio();//Error de compilación
    }
}
```

Esto significa que se puede aplicar también el polimorfismo con interfaces, pues con una misma expresión se pueden llamar a diferentes versiones de un método de la interfaz.

Herencia entre interfaces

Una interfaz puede ser heredada por otra interfaz, además, una interfaz puede heredar múltiples interfaces:

```
interface I1{..}
interface I2{..}
interface I3 extends I1,I2{..}
```

Cuando una clase implemente una interfaz que hereda otras interfaces, estará obligada a implementar los métodos de todas las interfaces de la jerarquía.

Interfaces Java 8

En Java 8 las interfaces incorporar importantes novedades, concretamente, el hecho de que puedan incluir métodos con código. En este sentido, además de métodos abstractos, una interfaz puede incluir dos tipos de métodos:

- **Métodos por defecto**. Consisten en implementaciones por defecto de ciertos métodos de la interfaz. Si una clase implementa una de estas interfaces, sería como si heredase los métodos por defecto de ella, no tiene la obligación de implementarlo (aunque podría hacerlo). Un método por defecto se declara con la palabra *default*:

```
interface Inter{
    default void m(){
          System.out.println("implementación por defecto");
    }
}
```

Los métodos default, al igual que los abstractos, **son métodos públicos**.

Cuando una clase implemente la interfaz, heredará los métodos default:

```
class Prueba extends Inter{
:
}
Prueba p=new Prueba();
p.m(); //se llamará al método default
```

- **Métodos estáticos.** Desde Java 8 las interfaces pueden incluir métodos estáticos. Los métodos estáticos están asociados únicamente a la interfaz, las clases que implementan la interfaz no los heredan, por lo que no podrán ser llamados con el nombre de la clase o con una referencia a un objeto de la misma, solo con el nombre de la interfaz:

```
interface Inter{
    //los métodos estáticos son públicos, el modificador
          //public es redundante
    static void pr(){
          System.out.println("estatico");
    }
    default void m(){
          System.out.println("implementación por defecto");
    }
}
class Prueba extends Inter{
:
}
Prueba p=new Prueba();
p.m(); //llamada a default
p.pr();//error de compilación!
Prueba.pr();//error de compilación!
Inter.pr();//correcto, llamada al método estático
```

PREGUNTAS TIPO EXAMEN

Pregunta 1

Which statement is/are true?

I. Default constructor only contains "super();" call.

II. Only constructor with no parameters in the superclass can be called from subclass.

III. super o this call must be the first statement in all constructors.

A. Only I.

B. Only II.

C. Only I and II.

D. Only I and III.

E. AII

Pregunta 2

Given the following:

```
class ClaseOne{
        public ClaseOne (int n){
                System.out.println("Second constructor");
         }
}
public class ClaseTwo extends ClaseOne{
        public ClaseTwo(){
                System.out.println("constructor one from object");
```

```
        }
        public ClaseTwo (int p){
                System.out.println("constructor one from object");
        }

}
```

And the following main method:

```
        ClaseTwo cd = new ClaseTwo(10);
```

Which is the result?

A. constructor one from object

Second constructor

B. Second constructor

constructor two from object

C. constructor two from object

D. Compilations fails

Pregunta 3

Given the following:

```
class Data1 {
        int x;
        Data1(){
                this(100);    //line 1
        }
```

```
        Data1(int n){
            this.x=n;
        }
    }
    class Data2 extends Data1{
        int y;
        Data2(){
            super();
            this(5);    //line 2
        }
        Data2(int n){
            Data1();  //line 3
            this.y=n;
        }
        public String toString(){
            return super.x+":"+this.y;
        }
    }
```

And given the following fragment:

```
Data2 dt=new Data2();
System.out.println(dt);
```

What is the result?

A. 100:5

B. 0:5

C. Compilation fails at line 1

D. Compilation fails at line 2

E. Compilation fails at line 2 and line 3

Pregunta 4

Given the following:

```
class Data1 {
        private int x;
        Data1(int x){
                this.x=x;
        }
}
class Data2 extends Data1{
        int y;
        Data2(int x, int y){
                //line 1
        }
}
```

And given the following fragment:

```
Data2 dt=new Data2(2,7);
```

Which code fragment should you use at line 1 to instantiate the dt object successfully?

A. super.x=x
   this.y=y;

B. super(x);

this(y);

C. super(x);

this.y=y;

D. this.x=x;

super(y);

Pregunta 5

Given:

```
public class Test {
        void myMethod(){}
}
class Exam extends Test{
        ____ void myMethod(){}
}
```

Which two of the following can fill in the blank in this code to make it compile?

A. abstract

B. int

C. private

D. protected

E. public

Pregunta 6

Given:

```
Class A { }
```

```
Class B { }

Interface X { }

Interface Y { }
```

Which two definitions of class C are valid?

A. class C extends A implements X { }

B. class C implements Y extends B { }

C. class C extends A, B { }

D. class C implements X, Y extends B { }

E. class C extends B implements X, Y { }

Pregunta 7

Given the following:

```
abstract class Car{
        protected void run(){}    //line 1
        abstract Object stop();      //line 2
}
class MyCar extends Car{
        void run(){}    //line 3
        protected void stop(){}        //line 4
}
```

Which two modifications are necessary to enable the code to compile?

A. Make the method at line 1 public.

B. Make the method at line 2 public.

C. Make the method at line 3 public.

D. Change the return type of method in line 4 to String.

E. Make the method at line 4 public.

Pregunta 8

Given the code:

```
public static void main(String[] args){
        Short a=100;
        Integer b=300;
        Long c=(long)a+b; //line 1
        String d=(String)(c*b);  //line 2
        System.out.println("Result: "+d)
}
```

What is the result?

A. Sum is 400

B. Compilation fails at line 1.

C. Compilation fails at line 2.

D. A ClassCastException is thrown at line 1.

E. A ClassCastException is thrown at line 2.

Pregunta 9

Which two are benefits of polymorphism?

A. Faster code at runtime

B. More efficient code at runtime

C. More dynamic code at runtime

D. More flexible and reusable code

E. Code that is protected from extension by other classes

Pregunta 10

Given the following class declarations:

```
public abstract class Animal

public interface Hunter

public class Cat extends Animal implements Hunter

public class Tiger extends Cat
```

Which answer fails to compile?

A.
```
ArrayList<Animal> ml=new ArrayList<>()

ml.add(new Tiger());
```

B.
```
ArrayList<Hunter> ml=new ArrayList<>()

ml.add(new Cat());
```

C.
```
ArrayList<Hunter> ml=new ArrayList<>()

ml.add(new Tiger());
```

D.
```
ArrayList<Tiger> ml=new ArrayList<>()

ml.add(new Cat());
```

E.
```
ArrayList<Animal> ml=new ArrayList<>()

ml.add(new Cat());
```

SOLUCIONES

1. D. La segunda afirmación es falsa porque, como hemos visto, a través de la expresión super(argumentos) podemos llamar a cualquier constructor de la superclase.

2. D. La clase ClaseTwo no compila. Sus constructores incluyen de forma implícita una llamada al constructor sin parámetros de la superclase, pero como este constructor no existe, se produce un error de compilación en la subclase.

3. E. La instrucción de line 2 es incorrecta, porque la llamada al constructor de la propia clase debería ser la primera instrucción del constructor. Line 3 también es incorrecta porque un constructor no es un método y no se le puede llamar por su nombre, sino a través de this.

4. C. Para inicializar el atributo heredado, se debe hacer a través del constructor de la superclase y la llamada a este debe ser la primera instrucción. Para inicializar el atributo de la propia clase se puede utilizar this.nombre_atributo.

5. D y E. En la sobrescritura el ámbito debe ser igual o superior que el del método original. Como el método original es default, los posibles ámbitos del nuevo método pueden ser protected y public.

6. A y E. La B no es correcta porque primero se debe indicar la clase que se hereda y después la interfaz que se implementa. C no es correcta porque una clase no puede heredar dos clases y D tampoco es correcta por el mismo motivo que la B.

7. C y D. Como el método run() de Car tiene ámbito protected, el run() de MyCar debe tener el mismo o superior, por lo que la C es correcta. El tipo de devolución del método stop() de MyCar debe ser igual o un subtipo del original, así que la D es también correcta.

8. C. La operación de la línea1: Long c= (long)a hace que se lleve a cabo unboxing de a y su conversión en long, para posteriormente hacer un autoboxing para convertirlo a objeto, por lo tanto, esta línea compila bien. Pero en la línea 2 se intenta convertir una operación numérica en String, lo cual es incorrecto y provoca un error de compilación en esa instrucción.

9. C y D. Por la definición de polimorfismo, no hace que sea más rápido el código en ejecutarse ni más eficiente, pero sí más flexible y dinámico.

10. D. Tiger es un Cat y no al revés, por ello, en una variable de tipo Tiger (o en una colección de tipo Tiger), no se pueden almacenar directamente referencias a objetos Cat. Tiger es un Animal, por lo que A es correcta, Cat es un Hunter al implementar la interfaz, por lo que B es correcta. Tiger es un Hunter porque hereda Cat, por lo que C es correcta, y Cat es un Animal, lo que implica que E es también correcta.

8 Excepciones

EXCEPCIONES. CONCEPTO Y TIPOS

El conocimiento de los tipos de excepción, las principales clases existentes, así como la manera de capturar y relanzar excepciones, es algo que debemos conocer bien para responder correctamente a las varias preguntas de examen que vamos a encontrar sobre este objetivo.

Concepto de excepción

Una excepción es una situación anómala que se puede producir durante la ejecución de un programa. Las excepciones se producen debido a fallos de programación unas veces, y otras a situaciones que escapan al control del programador (error en la introducción de un dato de usuario, corrupción de un fichero, etc.).

Un programa puede recuperarse de una excepción a través de una gestión de excepciones. El control de excepciones debe aplicarse en aquellos casos donde la excepción pueda ser producida por situaciones no debidas a fallos de programación, como la introducción de datos incorrectos por parte del usuario, fallos de conexión con una base de datos, etc.

Clases de excepciones

Las diferentes excepciones que pueden producirse en un programa Java están representadas por una clase. Todas las clases de excepción heredan Exception. La siguiente figura nos muestra un extracto de las clases de excepción más comunes:

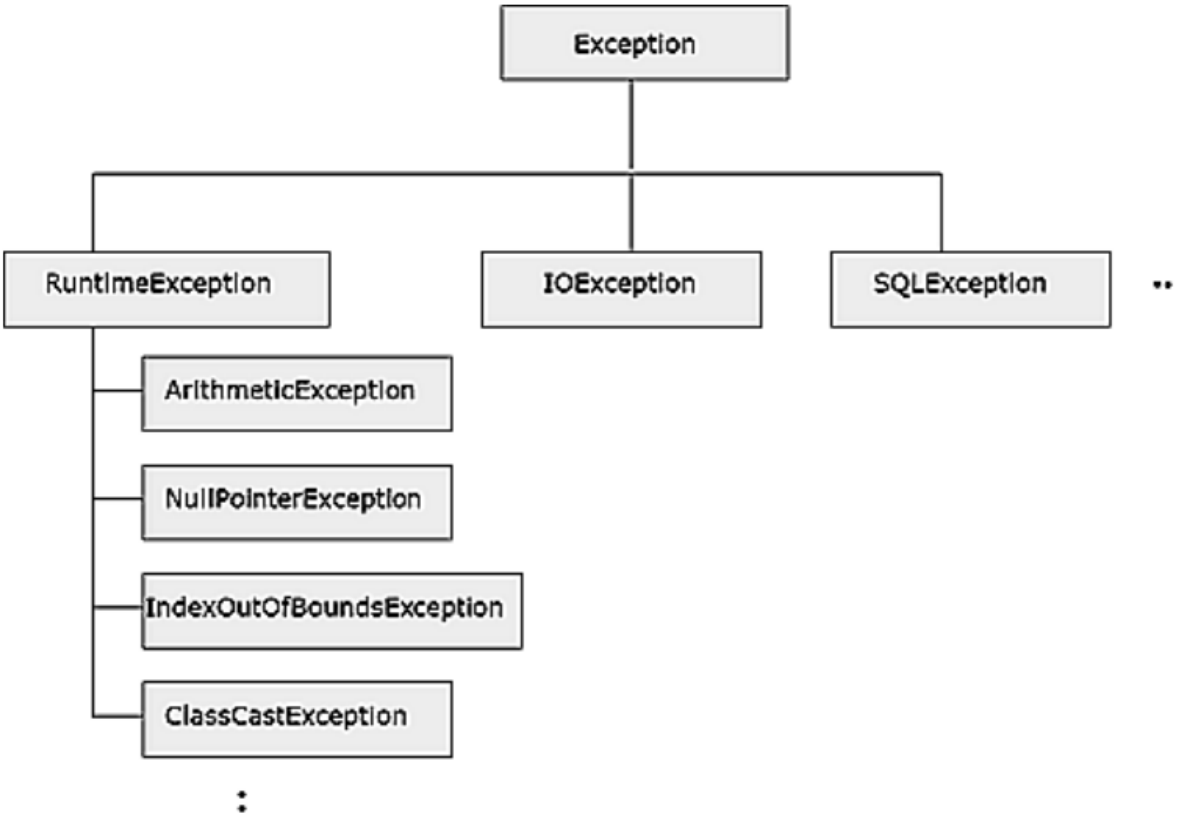


Fig. 8-1. Clases de excepciones

Clasificación de las excepciones

En función de su naturaleza, podemos clasificar las excepciones en dos grandes grupos:

- **Unchecked**. Conocidas también como excepciones de sistema y vienen representadas por subclases de RuntimeException. Se producen habitualmente por errores de programación, por ese motivo, Java **no obliga a capturar este tipo de excepciones**. Salvo algunos casos concretos que sí tendríamos que capturar, lo que se debe hacer con estas excepciones es corregirlas durante la fase de depuración del programa.

- **Checked**. Estas excepciones son lanzadas por métodos de clases del API Java, debido a alguna situación anómala producida en el interior de ese método. Estas excepciones no heredan RuntimeException sino que vienen directamente de Exception. Unas veces pueden producirse por errores de programación y otras por situaciones inesperadas, pero en cualquier caso, **el compilador Java obliga a realizar una captura** de este tipo de excepciones.

Excepciones Runtime

De cara al examen es necesario reconocer algunas de las excepciones *unchecked* más importantes, así que vamos a presentarlas:

- **ArrayIndexOutOfBoundsException.** Esta excepción se produce cuando intentamos acceder a una posición que está fuera de los límites de un array. Por ejemplo:

```
int [] datos=new int[10];

datos[10]=3; //arrayIndexOutOfBoundsException
```

  Es un claro ejemplo de excepción debida a errores de programación, por lo que no se debe capturar en ningún caso. Una variante de este tipo de excepción es la StringIndexOutOfBoundsException, que se produce al intentar acceder a una posición no existente dentro de una cadena de caracteres. Aunque estudiaremos los métodos de String en el próximo capítulo, aquí tenemos un ejemplo de este tipo de excepción:

```
String cad="Java";

System.out.println(cad.charAt(8));//excepción
```

- **NullPointerException.** Se trata de uno de los errores más habituales cometidos en programación. Esta excepción se produce cuando intentamos **el acceso a métodos de un objeto con referencia a null**. En el siguiente ejemplo vemos un caso típico de esta excepción al intentar utilizar una variable atributo de tipo objeto, a la que no se le ha asignado ningún objeto y que, por tanto, se encuentra inicializada implícitamente a null:

```
class Test{
```

```
        String s; //inicializada a null

        public static void main(String[] args){

              int n=s.length();//NullPointerException

        }

}
```

- **SecurityException.** Es una excepción que se produce por una violación de seguridad en el interior de la JVM.

- **ClassCastException.** Comentamos sobre la posibilidad de producirse esta excepción cuando hablamos en el capítulo anterior del casting entre tipos de objetos. Esta excepción tiene lugar cuando se asigna una variable de la superclase a un tipo de la subclase y el objeto referenciado por la variable de la superclase no es del tipo de la subclase. Veámoslo con un ejemplo:

```
Object ob=new String("34");

Integer in=(Integer)ob;//ClassCastException
```

- **ArithmeticException.** Esta excepción se da cuando se intenta realizar una operación aritmética no permitida, como por ejemplo, dividir un número entero entre 0:

```
int n=5/0;//ArithmeticException

double r=3/0.0;//ok
```

  Fijémonos en el ejemplo anterior como la división entre el cero double, es decir, 0.0 **no provoca esta excepción**, daría como resultado *Infinity*. Si realizamos una operación con una variable double con valor *Infinity*, el resultado será también *Infinity* o *NaN*, que significa resultado no numérico.

- **IllegalArgumentException.** Se produce cuando un método recibe como parámetro un valor no válido para el método. Sintácticamente, el valor puede ser correcto porque corresponda al tipo del parámetro y, por tanto, el compilador no genera ningún tipo de error, pero si el valor no

tiene ningún sentido para el método generará esta excepción. Aquí tenemos un ejemplo:

```
Thread.sleep(-10);
```

El método *sleep()* de la clase Thread recibe un entero con el tiempo de espera del hilo, pero si el valor que se le pasa es negativo (no tiene sentido un valor de tipo negativo), se producirá esta excepción.

Un subtipo de esta excepción es NumberFormatException, que tiene lugar cuando se intenta convertir una cadena de texto no válida a número:

```
String num="23g";

int n=Integer.parseInt(num);//NumberFormatException
```

Errores

De cara al examen de certificación, es muy importante saber distinguir entre excepciones y errores.

Mientras que una excepción es una situación inesperada, pero de la que el programa se puede recuperar, un error es fallo general dentro del programa del que **no hay posibilidad de recuperación**, pues afecta al funcionamiento de la JVM. Un ejemplo de error es cuando se produce una falta de memoria o un desbordamiento en la pila de llamadas.

Aunque los errores no se tratan dentro de un programa, también están representados por clases, clases que heredan Error: OutOfMemoryError, StackOverFlowError, InternalError, etc.

Por ejemplo, la ejecución del siguiente método provocaría un StackOverFlowError, que es un desbordamiento en la pila de llamadas debido a una llamada recursiva a método infinita:

```
private static void print() {

      print();

}
```

CAPTURA DE EXCEPCIONES

Capturar una excepción consiste en incluir dentro del programa un control de dicha situación, de modo que si la excepción se produce, el flujo de ejecución se transfiera a un bloque de código para el tratamiento del error.

La ventaja del sistema de control de excepciones de Java es que el tratamiento de la excepción se realiza en un bloque de código diferente al bloque de código donde se generó, lo que facilita la codificación y el seguimiento de los programas.

Bloques try catch

Las excepciones se capturan en un programa Java a través de los bloques try catch. La estructura de un bloque try catch se muestra en el siguiente esquema:

```
try{

 //instrucciones

}

catch(TipoExcepcion1 ex){

//tratamiento excepción

}

catch(TipoExcepcion2 ex){

//tratamiento excepción

}
```

Dentro del bloque try se incluyen las instrucciones donde se puede producir la excepción y también aquellas que deban ejecutarse si no hay ningún error.

En caso de que se produzca una excepción, el programa saltará al bloque catch que se encarga de tratar ese tipo de excepción. Después de ejecutarse el catch correspondiente, el flujo de ejecución continúa con la siguiente instrucción después del último catch, **no se vuelve al punto donde se generó la excepción**.

Si no hay ningún catch que capture la excepción producida, se interrumpirá la ejecución del programa y se mostrará en la consola el volcado de error. No obstante, lo anterior solo podría ocurrir con excepciones de tipo unchecked, ya que en el caso de las excepciones de tipo checked el compilador obliga a incluir un catch para el tratamiento de la excepción.

UTILIZACIÓN PRÁCTICA

En el siguiente programa de ejemplo vemos una utilización práctica de la captura de excepciones. Se trata de un programa que solicita la introducción de dos números y nos muestra el resultado de dividir el primero entre el segundo. Mediante los bloques try catch capturamos las excepciones NumberFormatException y AritmetichException:

```
Scanner sc=new Scanner(System.in);
int a=0,b=0,res;
try {
    System.out.println("Numerador: ");
    a=Integer.parseInt(sc.nextLine());
    System.out.println("Denominador: ");
    b=Integer.parseInt(sc.nextLine());
    res=a/b;
    System.out.println("División: "+res);
}
catch(NumberFormatException ex) {
    System.out.println("Error en los números ");
}
catch(ArithmeticException ex) {
    System.out.println("No se puede dividir entre 0");
}
```

CONSIDERACIONES SOBRE EL USO DE BLOQUES TRY CATCH

A la hora de definir los bloques try catch para el tratamiento de excepciones, debemos de tener en cuenta un par de detalles que deberás tener muy presentes de cara a alguna posible pregunta de examen:

No puede haber ninguna instrucción entre los bloques try y catch. Por ejemplo, lo siguiente genera un error de compilación:

```
try{...}

System.out.println("hello"); //error de compilación

catch(…){}
```

Si se capturan varios tipos de excepciones que tienen relación de herencia entre ellas, los **catch de las subclases deben situarse delante de los de las superclases**. Por ejemplo, la siguiente captura de excepciones es correcta, puesto que FileNotFoundException es subclase de IOException:

```
catch(FileNotFoundException ex){

..

}

catch(IOException ex){

..

}
```

Sin embargo, esta otra captura provocaría un error de compilación, ya que AritmetichException es subclase de RuntimeException y debería aparecer delante de esta:

```
catch(RuntimeException ex){

..

}
```

```
catch(ArithmeticException ex){ //Error de compilación

..

}
```

Multicatch

Cuando varios bloques catch de diferentes excepciones tienen que realizar la misma tarea, podemos agruparlos en un multicatch. Por ejemplo, dados los siguientes bloque catch:

```
catch(IOException ex){

    System.out.println("error");

}

catch(SQLException ex){

    System.out.println("error");

}
```

Podemos agruparlos en un multicatch de la siguiente manera:

```
catch(IOException|SQLException ex){

    System.out.println("error");

}
```

Como vemos, las clases de excepción se separan por una barra vertical. Podemos tener tantas excepciones como queramos capturar en el mismo bloque.

Es importante tener en cuenta que las excepciones agrupadas en un multicatch **no pueden tener relación de herencia**. Por ejemplo, el siguiente multicatch provocaría un error de compilación:

```
catch(FileNotFoundException|IOException ex){

..

}
```

Métodos de Exception

Todas las clases de excepción heredan los siguientes métodos de Exception que podemos utilizar durante el tratamiento de cualquier excepción:

- **String getMessage()**. Devuelve una cadena de caracteres con un mensaje de error asociado a la excepción.
- **void printStackTrace()**. Genera un volcado de error que es enviado a la consola. Este método es útil cuando nos vemos obligados a capturar una excepción y, antes de realizar cualquier tarea de tratamiento de la misma, realizamos el volcado de error en la consola para depurar posibles problemas de programación.

Bloque finally

La utilización de un bloque *finally* garantiza que un conjunto de instrucciones se ejecuten siempre, se produzca o no la excepción.

Por ejemplo, fijémonos en esta nueva versión del programa de las divisiones entre dos números:

```
Scanner sc=new Scanner(System.in);

int a=0,b=0,res;

try {

    System.out.println("Numerador: ");

    a=Integer.parseInt(sc.nextLine());

    System.out.println("Denominador: ");

    b=Integer.parseInt(sc.nextLine());

    res=a/b;

    System.out.println("División: "+res);

}
```

```
catch(NumberFormatException ex) {
    System.out.println("Error en los números ");
    return;
}
catch(ArithmeticException ex) {
    System.out.println("No se puede dividir entre 0");
}
System.out.println("Continuando");
```

La diferencia entre este código y la versión anterior es que en el catch de la excepción NumberFormatException hemos incluido un *return* para que se abandone el método si se produce una excepción. Además, después del último catch tenemos una instrucción que muestra un mensaje que queremos que se visualice después de la operación, tanto si hay excepción como si no.

El problema es que, si se produce una NumberFormatException, el *return* hará que se abandone la ejecución del método y, por tanto, el mensaje "Continuando" no aparecerá si se da esta circunstancia.

Si lo que queremos es garantizar que el mensaje se muestre, pase lo que pase, tendremos que englobarlo en un bloque finally:

```
catch(NumberFormatException ex) {
    :
}
catch(ArithmeticException ex) {
    :
}
finally{
    System.out.println("Continuando");
}
```

Las instrucciones que se encuentran dentro del *finally* se van a ejecutar en cualquier caso, **incluso si se produce una llamada a *return***. En este caso, antes de abandonar el método se ejecutaría el bloque *finally*.

LANZAMIENTO Y PROPAGACIÓN DE EXCEPCIONES

En muchas ocasiones, el método en el que se produce una excepción no es el encargado de tratarla, sino que debe hacerlo el bloque de código donde se ha realizado la llamada al método, pero para que esto sea posible, el método deberá propagar la excepción al punto de llamada.

Además de esto, veremos también en este apartado cómo generar desde código una excepción, así como la definición de excepciones personalizadas.

Propagación de una excepción

Como hemos indicado, si un método que debe capturar una excepción no desea hacerlo, puede propagarla al lugar de llamada al método. Para ello, declararemos la excepción en la cabecera del método con la instrucción **throws:**

metodo(...) throws ClaseExcepcion{

:

}

Por ejemplo, supongamos que estamos implementando un método en el que hacemos uso del método *readLine()* de BufferedReader. Este método obliga a capturar la IOExcepcion:

```
class Test{

 void metodo(){

    BufferedReader bf=new BufferedReader(…);

    try{

       //la llamada a readLine puede

       //provocar una IOException

       String s=bf.readLine();
```

```
        }
        catch(IOEXception ex){
        }
    }
    }
```

Pero si no queremos capturar aquí la excepción, deberíamos propagarla al punto de llamada a *metodo()*. Simplemente, declararemos la excepción en la cabecera del método:

```
class Test{
    metodo() throws IOException{
                BufferedReader bf= new BufferedReader(…);
            String s=bf.readLine();
    }
}
```

En este caso, será el código donde se haga uso de metodo() donde se tendrá que incluir la captura de la IOException:

```
class OtraClase{
    void otroMetodo(){
          Test t=new Test();
          try{
                t.metodo();
          }
          catch(IOException ex){ex.printStackTrace();}
    }
}
```

Lanzamiento de una excepción

Desde un método de una clase se puede lanzar una excepción para que sea capturada desde el punto de llamada al método. Se trata de una forma de informar al punto de llamada de que se ha producido una situación anómala en nuestro método.

Para lanzar una excepción se utiliza la instrucción:

throw objeto_excepcion;

Donde objeto_excepcion representa un objeto de la clase de excepción que queremos lanzar.

De cara a posibles preguntas de examen, debemos prestar atención a la diferencia entre el uso de *throw* y *throws*. La primera es para lanzar una excepción, mientras que la segunda es para propagarla.

Habitualmente, ambas sentencias se emplean conjuntamente, ya que si un método lanza una excepción al punto de llamada, debe propagarla también para que el compilador no le obligue a capturarla:

```
metodo() throws IOException{
    :
    //creación y lanzamiento de la
    //excepción
    throw new IOException();
}
```

Si no declaramos la excepción en la cabecera del método, no compilará, pues al tratarse de una excepción checked, el compilador nos obligaría a capturarla en la misma instrucción donde la lanzamos. Si se trata de excepciones **Runtime, no necesitamos declararlas** para que pueda compilar el código:

```
metodo(){
    :
    //creación y lanzamiento de la
    //excepción
    throw new RuntimeException();
}
```

Si se produce la excepción se propagará automáticamente al punto de llamada.

Excepciones personalizadas

Si queremos crear nuestras propias clases de excepción personalizadas para posteriormente lanzarlas y propagarlas desde nuestros métodos, bastará con que estas clases **hereden Exception**:

```
class TestException extends Exception{
}
```

Por el hecho de heredar directamente Exception nuestra clase representaría una excepción de tipo *checked*. Podríamos utilizarla desde cualquier método de nuestro código como se indica en la siguiente clase de ejemplo:

```
class C1{
    //propaga la excepción que lanza
    public void metodo() throws TestException{
        :
        throw new TestException();
    }
}
```

Como con cualquier otra excepción de tipo *checked*, si desde otra clase creamos un objeto de C1 y utilizamos *metodo()*, estaríamos obligados a capturar esta excepción:

```
C1 c=new C1();

try{

      //al utilizar metodo() se debe capturar

      //la excepción

      c.metodo();

}

catch(TestException t){

    :

}
```

PREGUNTAS TIPO EXAMEN

Pregunta 1

Which three are advantages of the Java exception mechanism?

A. Improves the program structure because the error handling code is separated from the normal program function

B. Provides a set of standard exceptions that covers all the possible errors

C. Improves the program structure because the programmer can choose where to handle exceptions

D. Improves the program structure because exceptions must be handled in the method in which they occurred

E. Allows the creation of new exceptions that are tailored to the particular program being created

Pregunta 2

Given the following:

```
public static void main(String[] args){
        ArrayList lst=new ArrayList();
        String[] mr;
        try{
                while(true){
                        lst.add(new String("cad"));
                }
        }
        catch(RuntimeException ex){
                Sustem.out.println("Is a RuntimeException");
```

```
            }
            catch(Exception ex){
                    Sustem.out.println("Is a Exception");
            }
            Sustem.out.println("End");
    }
```

What is the result?

A. Execution terminates in the first catch statement, and caught a RuntimeException is printed to the console.

B. Execution terminates In the second catch statement, and caught an Exception is printed to the console.

C. A runtime error is thrown in the thread "main".

D. Execution completes normally, and "End" is printed to the console.

E. The code fails to compile because a throws keyword is required.

Pregunta 3

Given the following classes:

```
public class TestException extends RuntimeException {}
public class Test{
        public static void main(String[] args){
                try{
                        myMethod();
```

```
                }
                catch(TestException ex){
                        System.out.print("A");
                }
        }
        public static void myMethod(){  //line 1
                try{
                        throw (Math.random()>0.5)?new TestException():
                                        new RuntimeException();
                }
                catch(RuntimeException ex){
                        System.out.print("B");
                }
        }
    }
```

What is the result?

A. A

B. B

C. Either A or B

D. AB

E. Compilation fails at line 1

Pregunta 4

Which two are Java System Exception classes?

A. SercurityException

B. DuplicatePathException

C. IllegalArgumentException

D. TooManyArgumentsException

Pregunta 5

Given:

```
public class Test {
    public static void main(String[] args) {
        int ax = 10, az = 30;
        int aw = 1, ay = 1;
        try {
            aw = ax % 2;
            ay = az / aw;
        } catch (ArithmeticException e1) {
            System.out.println("Invalid Divisor");
        } catch (Exception e2) {
            aw = 1;
            System.out.println("Divisor Changed");
        }
        ay = az /aw; // Line 14
```

```
                System.out.println("Succesful Division " + ay);
        }
}
```

What is the result?

A. Invalid Divisor

Divisor Changed

Successful Division 30

B. Invalid Divisor

Successful Division 30

C. Invalid Divisor

Exception in thread "main" java.lang.ArithmeticException: / by zero

at test.Teagle.main(Teagle.java:14)

D. Invalid Divisor

Exception in thread "main" java.lang.ArithmeticException: / by zero

at test.Teagle.main(Teagle.java:14)

Successful Division 1

Pregunta 6

Given:

```
public class Test {
public static void main(String[] args) {
int arr[] = new int[4];
arr[0] = 1;
```

```
arr[1] = 2;

arr[2] = 4;

arr[3] = 5;

int sum = 0;

try {

for (int pos = 0; pos <= 4; pos++) {

sum = sum + arr[pos];

}

} catch (Exception e) {

System.out.println("Invalid index");

}

System.out.println(sum);

}

}
```

What is the result?

A. 12

B. Invalid Index

12

C. Invalid Index

D. Compilation fails

SOLUCIONES

1. A, C y E. El número de posibles excepciones es ilimitado, por lo que la B es incorrecta. Las excepciones no tienen que ser capturadas en el método donde se producen, se pueden propagar, por lo que la D tampoco es correcta.

2. C. Al incluirse la instrucción lst.add(new String("cad")); en un bucle infinito, llegará un momento en que el programa se quede sin memoria y se producirá un error de tipo OutFoMemoryError.

3. B. Independientemente de que se produzca TestException o RuntimeException, ambas son RuntimeException, por lo que entrará en el catch y se imprimirá B. La excepción no se propaga, por lo que no entrará en el catch del main.

4. A y C. Las excepciones indicadas en B y D no son subclases de Runtime, son de tipo checked.

5. C. La instrucción que está dentro del try: ay = az / aw;, provoca una ArithmeticException al realizar una división entera entre 0, pero como esta instrucción está también después del catch al ejecutarse producirá una excepción que no está capturada y se mostrará la traza en pantalla.

6. B. El for sale de los límites del array, por lo que se produce una excepción y se entra dentro del catch. Como después se imprime la suma, se mostrarán también los números del array que se han sumado, que han sido todos.

9 Estudio de las clases del API de Java

MANIPULAR CADENAS CON STRING

En este último capítulo vamos a estudiar las clases básicas de Java estándar que son objetivo de examen.

Empezaremos hablando de la clase String. Durante el capítulo 3 comentamos algunas de las características de esta clase durante el estudio de la igualdad de objetos. En este apartado nos centraremos en analizar los principales métodos de String, en concreto, aquellos que serán objeto de alguna pregunta de examen.

Fundamentos sobre String

Como ya sabemos, un objeto de la clase String es una cadena de caracteres **inmutable**, lo que significa que no se pueden modificar.

Existen varios métodos de la clase String que operan con la cadena de caracteres para realizar algún tipo de transformación de la misma, pero estos métodos **no alteran la cadena original** porque es inmutable, lo que hacen es devolver una copia a de la cadena modificada.

Para crear un objeto String podemos proceder de la siguiente forma:

```
String n1=new String("mi cadena");
```

La instrucción anterior fuerza a la creación de una nueva instancia de la clase String con el texto indicado. Pero también podemos hacer:

```
String ni="mi cadena";
```

En este caso no necesariamente se crea un nuevo objeto de texto, sino que se puede reutilizar uno de los objetos del pool del literal de cadenas.

Métodos de la clase String

A continuación vamos a analizar los métodos más importantes de la clase String y que pueden aparecer en algunas preguntas de examen. No incluimos el método *equals()*, ya estudiado durante la igualdad de objetos:

- **int length()**. Devuelve la longitud de la cadena.
- **String toLowerCase(), toUpperCase()**. Devuelven la cadena convertida a minúsculas y mayúsculas, respectivamente. Recordar lo que indicábamos antes sobre la inmutabilidad de String y que queda patente en el siguiente ejemplo en el que se ve que, tras llamar a estos métodos, el objeto no cambia:

```
String n1="cadena";

System.out.println(n1.toUpperCase()); //muestra:

                                          // CADENA

System.out.println(n1); //muestra: cadena.

                          //No ha cambiado
```

- **String substring(int a, int b)**. Devuelve la parte de la cadena comprendida entre las posiciones a y b-1. Tenemos que tener en cuenta que el primer carácter de la cadena ocupa la posición 0 y el último length()-1. El siguiente bloque de código mostraría "o es u" al ejecutarse:

```
String n1="esto es un texto";

System.out.println(n1.substring(3,9));//muestra:o es u
```

De cara a alguna pregunta de examen en la que aparezca este método debemos de tener en cuenta que si a es negativo y b mayor que la longitud de la cadena, se producirá una **StringIndexOutOfBoundsException**.

- **char charAt(int pos)**. Devuelve el carácter que ocupa la posición indicada. De la misma forma que el método anterior, si se indica una posición no válida, se producirá una StringIndexOutOfBoundsException:

```
String n1="esto es un texto";

System.out.println(n1.charAt(0)); //muestra: e

//StringIndexOutOfBoundsException

System.out.println(n1.charAt(20));
```

- **int indexOf(String cad)**. Devuelve la posición de la cadena que se proporciona como parámetro. **Si no existe, devuelve -1**:

```
String n1="esto es un texto";

System.out.println(n1.indexOf("un")); //muestra: 8
```

Existe una versión de este método en el que se le puede indicar como segundo parámetro, la posición a partir de la cual comenzar la búsqueda. También hay una versión de *indexOf()* que recibe como parámetro un carácter de búsqueda en lugar de una cadena.

- **String replace(CharSequence c1, CharSequence c2)**. Devuelve la cadena resultante de reemplazar la subcadena c1 por c2. Observemos que el parámetro recibido es una implementación de CharSequence; **String implementa esta interfaz**. El siguiente bloque de código mostraría la cadena "deto de un texto" al ser ejecutado:

```
String n1="esto es un texto";

//muestra: deto de un texto

System.out.println(n1.replace("es","de"));
```

Como sucediera con los métodos *toLowerCase* y *toUpperCase*, este método **devuelve un nuevo texto con el resultado de la operación**, el original no se ve alterado.

- **boolean startsWith(String s), endsWith(String s)**. Indican, respectivamente, si la cadena empieza o termina con el texto recibido como parámetro:

```
String n1="esto es un texto";

System.out.println(n1.endsWith("to")); //muestra: true

System.out.println(n1.startsWith("eso"));//false
```

- **String trim()**. Devuelve la cadena resultante de eliminar espacios al principio y al final del texto sobre el que se aplica:

```
String n1="  cade prueba nueva  ";

System.out.println(n1.trim().length()); //muestra: 17
```

- **String concat(String s)**. Devuelve el resultado de concatenar la cadena con el texto recibido como parámetro. En definitiva, tiene el mismo efecto que el operador de concatenación +. Volvemos a recalcar que el método devuelve un nuevo texto resultante de la operación, el texto original permanece inalterable:

```
String s="Java";

System.out.println(s.concat(" EE"); //muestra Java EE

System.out.println(s); //muestra Java
```

- **boolean isEmpty().** Indica si se trata o no de una cadena vacía. Tiene el mismo efecto que aplicar el método *equals()* sobre una cadena vacía:

```
String cad="n";

//ambas instrucciones generan mismo resultado

System.out.println(cad.isEmpty());

System.out.println(cad.equals(""));
```

MANIPULACIÓN DE CADENAS CON STRINGBUILDER

Al igual que String, esta clase fue presentada durante el estudio de la igualdad de objetos en Java. En este capítulo nos centraremos en analizar los principales métodos de esta clase y su efecto en el tratamiento de cadenas de caracteres mutables.

Fundamentos de StringBuilder

Como sabemos, los objetos de la clase StringBuilder representan cadenas de caracteres mutables, es decir, que pueden ser modificadas. Esto significa que los métodos actúan sobre el propio objeto, no generan una copia del mismo.

Para crear un objeto StringBuilder no podemos recurrir a la asignación de la cadena en la variable como en String, debemos emplear el operador new:

```
StringBuilder sb=new StringBuilder("cadena mutable");
```

No se admite la concatenación entre dos StringBuilder, pero sí la de un StringBuilder con un String. En este caso, se llamará implícitamente al método *toString()* de StringBuilder y se generará un nuevo String resultante de la concatenación de los dos textos:

```
String s="hola "+sb;

System.out.println(s);//muestra: hola cadena mutable
```

Métodos de StringBuilder

Durante el estudio de la igualdad de objetos, comentamos que StringBuilder dispone de la versión del método equals() heredada de Object, por lo que al aplicar este método a dos objetos diferentes siempre devolverá falso aunque contengan el mismo texto.

Entre los métodos definidos en la clase StringBuilder tenemos:

- **StringBuilder append(tipo dato)**. Se trata de un método sobrecargado que añade a la cadena el dato (existe una versión para cada tipo primitivo más String) que se le pasa como parámetro. Este método **modifica la cadena original** y, además, devuelve una referencia al propio objeto:

```
StringBuilder sb=new StringBuilder("mutable");

StringBuilder sb2=sb.append(" nueva");

System.out.println(sb); //llamada a toString(),

                //muestra: mutable nueva

System.out.println(sb==sb2); //muestra true,

                //apuntan al mismo objeto
```

- **StringBuilder append(String dato, int a, int b).** Es una variante del método anterior, que consiste en añadir a la cadena solo una parte del texto que se pasa como parámetro, el comprendido entre los caracteres a y b-1:

```
StringBuilder sb=new StringBuilder("Java");

sb.append("document",0,3);

System.out.println(sb);//muestra Javadoc
```

- **StringBuilder insert(int pos, tipo dato)**. Inserta un dato dentro de la cadena en la posición indicada:

```
StringBuilder sb=new StringBuilder("es texto");

sb.insert(3,200)

System.out.println(sb); //muestra: es 200texto
```

- **StringBuilder replace(int a, int b, String s).** Reemplaza los caracteres que se encuentran situados entre las posiciones a y b-1 por la cadena que se indica como tercer parámetro. Devuelve también una referencia al propio objeto:

```
StringBuilder sb=new StringBuilder("primera cadena");

sb.replace(0,7,"segunda");

System.out.println(sb); //muestra: segunda cadena
```

- **StringBuilder delete(int a, int b)**. Elimina de la cadena los caracteres situados entre las posiciones a y b-1, y devuelve una referencia al objeto:

```
StringBuilder sb=new StringBuilder("cadena nueva");

sb.delete(3,6);

System.out.println(sb); //muestra: cad nueva
```

- **StringBuilder reverse()**. Invierte el orden de los caracteres de la cadena, devolviendo la referencia al objeto:

```
StringBuilder sb=new StringBuilder("hola");

sb.reverse();

System.out.println(sb); //muestra: aloh
```

Además de estos métodos propios de StringBuilder, la clase cuenta con otros métodos iguales a los de String:

- char charAt(int pos)
- int indexOf(String s)
- int length()
- String substring(int a, int b)

UTILIZACIÓN DE LISTAS

En este apartado vamos a analizar otro de los objetivos de examen que es la creación y manejo de colecciones de tipo lista. Una colección es una agrupación dinámica de objetos, que no tiene un tamaño fijo. A través de los métodos de la clase de colección, podemos añadir nuevos elementos, eliminar elementos existentes, recorrer la colección, etc.

Una lista es un tipo de colección en la que cada elemento de la misma tiene asociado un índice basado en la posición que ocupa dentro de la colección. La clase de colección de tipo lista más utilizada es ArrayList.

Fundamentos de ArrayList

La clase ArrayList permite crear colecciones de objetos de tipo lista, donde cada elemento ocupa una posición, siendo 0 la posición del primer elemento. La clase ArrayList se encuentra en el paquete java.util.

La colección ArrayList está **definida como tipo genérico**, es decir, puede ser utilizada con cualquier tipo Java, indicando en tiempo de ejecución el tipo específico Por tanto, para crear un objeto ArrayList se debe declarar la variable indicando mediante el operador diamante, el tipo de objeto a almacenar. En las siguientes instrucciones se declaran dos ArrayList, uno de enteros y otro de cadenas:

```
ArrayList<Integer> nums;//ArrayList de enteros

ArrayList<String> cads;//ArrayList de cadenas
```

De cara a crear los objetos, se utilizará el operador diamante después del nombre de la clase pero no será necesario especificar de nuevo el tipo de objeto:

```
nums=new ArrayList<>();

cads=new ArrayList<>();
```

Las colecciones anteriores **solo admitirán el tipo de objeto** para el que se han definido o un subtipo del mismo, produciéndose un error de compilación si se le intenta asignar otro diferente:

```
nums.add(100);//correcto

nums.add("Java");//error de compilación
```

ArrayList y la herencia

De cara a alguna posible pregunta de examen en esa línea, hay que tener en cuenta que el hecho de que una clase sea superclase de otra no significa que una colección de objetos de la subclase sea una colección de objetos de la superclase.

Para aclarar esto, centrémonos en dos clases relacionadas mediante herencia, por ejemplo, String y Object. Pues bien, en virtud del polimorfismo, en una colección de Object es posible almacenar objetos String:

```
ArrayList<Object> obs=new ArrayList<>();

obs.add("Es un String"); //correcto
```

Pero tengamos presente que una **colección de String NO es una colección de Object**. El siguiente código no compilaría:

```
ArrayList<Object> obs;

ArrayList<String> cads=new ArrayList<>();

obs=cads;//Error de compilación!
```

Un ArrayList de String no es un ArrayList de Object.

Métodos de ArrayList

A continuación, vamos a estudiar algunos de los métodos más importantes de ArrayList y que pueden ser objeto de alguna pregunta de examen:

- **boolean add(E elemento)**. Añade el elemento a la colección, situándolo al final de la misma. La letra E representa el tipo de elemento para el que se haya declarado el ArrayList. Siempre devuelve true:

```
ArrayList<Integer> nums=new ArrayList<>();

nums.add(20); //aplica autoboxing para

          //guardar el int como Integer
```

- **boolean add(int index,E elemento)**. Añade el elemento en la posición indicada, desplazando el que ocupaba esa posición y los siguientes hacia adelante:

```
ArrayList<Integer> nums=new ArrayList<>();

nums.add(20);

nums.add(40);

nums.add(1,30);//lo coloca entre el 20 y el 40
```

Si la posición indicada es mayor que el tamaño o negativa, **se producirá una IndexOutOfBoundsException**

- **E get(int index).** Devuelve el elemento que ocupa la posición indicada. Se producirá una IndexOutOfBoundsException si el índice está fuera de rango.

- **E remove(int index)**. Elimina el elemento que ocupa la posición indicada y devuelve el elemento eliminado. Desplaza los siguientes hacia atrás.

```
ArrayList<String> dias=new ArrayList<>();

dias.add("lunes");

dias.add("martes");

dias.add("miércoles");

//elimina el primero, quedarían martes y miércoles

dias.remove(0);
```

  Si la posición está fuera de rango se producirá una IndexOutOfBoundsException.

- **boolean remove(Object ob)**. Elimina la primera ocurrencia del objeto indicado. Devuelve true si dicho elemento existía y, por tanto, ha sido eliminado.

- **E set(int index, E elemento)**. Reemplaza el elemento existente en esa posición por el nuevo elemento, devolviendo el elemento sustituido. Como en otros casos, si la posición está fuera de rango se producirá una IndexOutOfBoundsException.

- **int size()**. Devuelve el tamaño de la colección.

- **String toString()**. Este método es heredado de Object, sin embargo, ArrayList **lo sobrescribe** para devolver una cadena con cada uno de los elementos del ArrayList, separados por una coma y delimitados por paréntesis:

```
ArrayList<Integer> nums=new ArrayList<>();

nums.add(20);nums.add(10);nums.add(5);

System.out.println(nums);//muestra [20,10,5]
```

- **T[] toArray(T[] a)**. Devuelve un array con los elementos de la colección. Requiere un array inicial del tipo de elementos de la colección. El tamaño de este array que se le debe pasar como parámetro estará entre 0 y el tamaño de la colección; independientemente de este valor, el método devolverá un array con todos los elementos de la colección.

```
ArrayList<Integer> nums=new ArrayList<>();

nums.add(20);nums.add(10);nums.add(5);

//se le proporciona un array vacío

Integer[] ns=nums.toArray(new Integer[0]);
```

Recorrido de un ArrayList

Dado que el método *size()* nos indica el tamaño del ArrayList en todo momento, se puede recorrer con un bucle for estándar. El siguiente bloque de código nos muestra cada uno de los elementos de la colección.

```
ArrayList<Integer> nums=new ArrayList<>();

nums.add(20);nums.add(10);nums.add(5);

for(int i=0;i<nums.size();i++){

    System.out.println(nums.get(i));

}
```

Se puede emplear también un enhanced for para recorrer un ArrayList:

```
ArrayList<Integer> nums=new ArrayList<>();

nums.add(20);nums.add(10);nums.add(5);

for(int n:nums){

    System.out.println(n);

}
```

Como variable de control del enhanced for se puede usar tanto *Integer* como *int*, ya que aplica el unboxing para extraer el dato de la colección y guardarlo en la variable.

La interfaz List

ArrayList implementa la interfaz List, por lo que es habitual trabajar con la interfaz que contiene una referencia al objeto ArrayList:

```
List<String> cadenas=new ArrayList<>();
```

Todos los métodos de ArrayList que hemos estudiado anteriormente están definidos en List, por lo que pueden aplicarse con referencias de tipo List:

```
List<String> cadenas=new ArrayList<>();

cadenas.add("lunes");

cadenas.add("miércoles");

cadenas.add(1,"martes");

cadenas.remove(0);

for(String s:cadenas){

      System.out.println(s); //imprime: martes miércoles

}
```

OBTENCIÓN DE OBJETOS LIST

Además de la clase ArrayList que implementa esta interfaz, es habitual encontrar preguntas de examen en las que se obtiene una implementación de List utilizando el método estático *asList()* de la clase Arrays.

El método asList() tiene el siguiente formato:

static List<T> asList(T... a);

Como vemos, a partir de un número variable de argumentos de un tipo devuelve un List de objetos de ese tipo:

```
List<Integer> nums=Arrays.asList(20,3,5,20);
```

Comentar que, aunque serían aplicables todos los métodos estudiados anteriormente, las listas obtenidas a través de este método son de tamaño fijo, por lo que **no se podrían añadir nuevos elementos ni eliminar elementos existentes**. Las llamadas a los **métodos *add()* y *remove()* sobre una colección creada de esta manera provocarían una excepción**:

```
nums.remove(0);//excepción
```

TRABAJAR CON FECHAS EN JAVA

El examen de certificación se centra en las nuevas clases para el manejo de fechas incluidas en el paquete java.time.

Podemos dividir el estudio de estas clases en dos grandes bloques:

- Clases para manejo de fechas/horas.
- Clases para intervalos de tiempo.

Comencemos pues con el estudio del primer grupo.

Clases para el manejo de fechas y horas

Antes de presentar las diferentes clases existentes para el manejo de fechas y horas, hay que resaltar el hecho de que, en todos los casos, **los objetos creados son inmutables**, es decir, una vez creados, sus valores de fecha/hora no pueden ser modificados.

CLASE LOCALDATE

Empezamos hablando de esta clase cuyos objetos representan una fecha concreta. Esta clase, como el resto de las que vamos a presentar en este documento, **no dispone de constructores públicos** para la creación de objetos, por lo que tendremos que recurrir a métodos estáticos de la propia clase.

En el caso de LocalDate, estos son los métodos estáticos que nos permitirían crear un objeto LocalDate:

- **static LocalDate now()**. Crea un objeto asociado a la fecha actual del sistema.

- **static LocalDate of(int year, int month, int day)**. Crea un objeto asociado a una determinada fecha específica, cuyos valores se pasan como parámetro. Los valores de los meses van de 1 (enero) a 12 (diciembre). Si el valor de alguno de los parámetros está fuera de rango o el día del mes no se corresponde con el valor indicado, se producirá una DateTimeException:

```
LocalDate ld=LocalDate.of(2017,2,30);
```

- **static LocalDate of(int year, Month month, int day)**. Es igual que el anterior, solo que el mes se puede pasar como un objeto Month. Month es una enumeración con los nombres de los meses del año:

```
LocalDate ld=LocalDate.of(2017,Month.MAY,3); //3 de

                                              //mayo de 2017
```

Si mostramos por pantalla un objeto LocalDate se llamará al método toString(), que devuelve la fecha en formato: yyyy-mmm-dd:

```
LocalDate ld=LocalDate.of(2017,12, 30);

Systemout.println(ld);// muestra 2017-12-30
```

Una vez creado el objeto, podemos recurrir a los siguientes métodos para la manipulación de la fecha:

- **int getYear(), getMonth() y getDayOfMonth()**. Devuelven, respectivamente, el año, mes y día del mes asociado a la fecha.

- **LocalDate plusYears(long y), plusMonths(long m) y plusDays(long d). Devuelven una copia de la fecha**, correspondiente a los años, meses o días añadidos. Si se sobrepasa el número de días del mes o el número de meses del año, se incrementará el mes/año, según el caso:

```
LocalDate ld=LocalDate.of(2017,12, 30);

LocalDate ld2=ld.plusMonths(2);

System.out.println(ld2); //muestra 2018-1-1

System.out.println(ld); //muestra 2017-12-30
```

Observa en el código anterior que la fecha original no se ha modificado, se ha generado una nueva fecha resultante de la operación. Es habitual encontrar preguntas de examen en donde se realiza una operación sobre una variable de fecha y el resultado no se almacena en ningún sitio:

```
ld.plusYears(10);
```

En estos casos, debemos de tener en cuenta que si se imprimiese el valor de la variable ld, seguiría mostrando la **misma fecha** que antes de llamar al método.

- **LocalDate minusYears(long y), minusMonths(long m) y minusDays(long d)**. Devuelven una copia de la fecha, correspondiente a los años, meses o días restados. Si el resultado de la resta en el caso de los meses o días es inferior a 0, se reajustan los años/meses/días de la fecha.

- **LocalDate withYear(int y), withMonth(int m) y withDayOfMonth(int d).** Devuelven una copia de la fecha con el nuevo año, mes o día establecido, respectivamente. Se produce una DateTimeException si el mes o día del mes establecido no es válido.

CLASE LOCALTIME

Representa una hora en concreto. Para crear un objeto LocalTime, recurriremos a alguno de los siguientes métodos estáticos:

- **static LocalTime now()**. Devuelve un objeto LocalTime con la hora actual.

- **static LocalTime of(int hour, int minute, int second)**. Devuelve un objeto LocalTime con la hora indicada. El valor de hora estará comprendido entre 0 y 23, mientras que los de minutos y segundos deberán estar comprendidos entre 0 y 59. Si alguno de los valores está fuera de rango, se producirá un DateTimeException. Existen otras dos versiones del método of, una que recibe horas y minutos y otra que recibe horas, minutos, segundos y nanosegundos.

Si mostramos por pantalla un objeto LocalTime, se llamará al método toString(), que devuelve la hora en formato: hh:mm

Tras la creación del objeto, podemos utilizar los siguientes métodos para la manipulación de la hora:

- **int getHour(), getMinute() y getSecond()**. Devuelven, respectivamente, la hora, los minutos y los segundos.

- **LocalTime plusHours(long h), plusMinutes(long m) y plusSeconds(long s)**. Devuelven un nuevo objeto LocalTime resultante de la suma de hora, minutos y segundo realizada:

```
LocalTime lt=LocalTime.of(23, 50,40);

System.out.println(lt.plusSeconds(800)); // 00:04
```

  Como en el caso de LocalDate, recuerda que estos métodos no modifican el objeto original, devuelven un nuevo objeto resultante de la operación.

- **LocalTime minusHours(long h), minusMinutes(long m) y minusSeconds(long s)**. Devuelven un nuevo objeto LocalTime resultante de restar la hora, los minutos y los segundos indicados.

- **LocalTime withHour(int h), withMinute(int m) y withSecond(int s)**. Devuelven una copia de la hora con la nueva hora, minuto o segundo establecido. Se producirá una excepción DateTimeException si el valor de alguno de los campos está fuera de rango.

CLASE LOCALDATETIME

Representa una combinación de fecha y hora. Para crear un objeto de este tipo utilizaremos los métodos estáticos:

- **static LocalDateTime now()**. Crea un objeto LocalDateTime asociado a la fecha y hora actuales.

- **static LocalDateTime of(int year, int month, int dayOfMonth, int hour, int minute)**. Crea un objeto LocalDateTime con los datos de fecha y hora proporcionados. Si alguno está fuera de rango, se producirá una DateTimeException.

Esta clase incluye también todos los métodos *get*, *plus* y *with* que hemos visto en las dos clases anteriores. Y, aparte de estos, disponemos también de los métodos:

- **LocalDate toLocalDate()**. Devuelve un objeto LocalDate con la parte de la fecha asociada al objeto.

- **LocalTime toLocalTime()**. Devuelve un objeto LocalTime con la parte de la hora asociada al objeto.

CLASE INSTANT

Un objeto Instant representa un instante en la línea de tiempo, tomando como referencia las 00:00:00 horas del 1-1-1970.

Existen diversos métodos estáticos para crear un objeto Instant:

- **static Instant now().** Crea un objeto asociado al instante de tiempo actual.
- **static Instant ofEpochMillis(long milis)**. Crea un objeto del instante asociado a los milisegundos indicados.
- **static Instant ofEpochSecond(long sec)**. Igual que el anterior, pero indicando los segundos transcurridos desde la fecha de referencia.

Al presentar un objeto de este tipo, el formato devuelto por el método toString() es de la forma: yyyy-mm-ddThh:mm:ssZ:

```
Instant it=Instant.ofEpochSecond(70000);

System.out.println(it); //muestra 1970-01-01T19:26:40Z
```

La clase proporciona los siguientes métodos para la manipulación del instante de tiempo:

- **long getEpochSecond()**. Devuelve los segundos transcurridos desde la fecha de referencia.
- **long toEpochMillis()**. Devuelve los milisegundos transcurridos desde la fecha de referencia.
- **Instant plusMillis(long m) y plusSeconds(long s)**. Devuelven un nuevo objeto Instant resultante de añadir los milisegundos o segundos indicados, respectivamente.
- **Instant minusMillis(long m) y minusSeconds(long s).** Devuelven un nuevo objeto Instant resultante de restar los milisegundos o segundos indicados, respectivamente.

Formateado de fechas

La nueva clase DateTimeFormatter del paquete java.time.format permite aplicar un determinado formato a los objetos LocalDate, LocalTime y LocalDateTime estudiados anteriormente. Las tres clases disponen de un método format que tiene la siguiente firma:

String format(DateTimeFormatter format).

A partir del objeto DateTimeFormatter que se proporciona como parámetro devuelven una cadena de caracteres, resultante de aplicar el formato indicado sobre el objeto de fecha/hora correspondiente.

Por tanto, lo primero será crear un objeto DateTimeFormatter, para lo cual utilizaremos alguno de los siguientes métodos estáticos:

- **static DateTimeFormatter ofLocalizedDateTime(FormatStyle dateTimeStyle)**. Devuelve un objeto DateTimeFormatter según el estilo indicado por el objeto FormatStyle. FormatStyle, que se encuentra también en el paquete java.time.format, proporciona una serie de constantes de estilo predefinido, concretamente, las constantes FULL, LONG, MEDIUM y SHORT. La siguiente instrucción crea un objeto DateTimeFormatter para aplicar un formato de fecha hora largo:

```
DateTimeFormatter dtf;

dtf=DateTimeFormatter.ofLocalizedDateTime(

                        FormatStyle.FULL);
```

- **ofLocalizedDate(FormatStyle dateStyle) y ofLocalizedTime(FormatStyle timeStyle)**. Igual que el anterior, aunque solo proporcionan formato para fecha y hora, respectivamente.

- **static DateTimeFormatter ofPattern(String pattern)**. Crea un objeto DateTimeFormatter a partir del patrón de formato indicado como parámetro. En la ayuda oficial de Oracle sobre la clase DateTimeFormatter tienes una tabla con los diferentes caracteres de formato que se pueden utilizar para construir el patrón de formato. Por ejemplo, si quisiéramos formatear una fecha/hora de manera que se pueda presentar de la forma día/mes/año - horas:minutos:segundos, esta sería la instrucción para la creación del objeto:

```
DateTimeFormatter dtf;

dtf=DateTimeFormatter.ofPattern(

                "dd/MM/YYYY - HH:mm:ss");
```

En la ayuda oficial de Java SE:

https://docs.oracle.com/javase/8/docs/api/

Dentro de la página de ayuda de la clase DateTimeFormatter, puedes encontrar información sobre los diferentes formatos predefinidos y caracteres de formato a aplicar.

Una vez creado el DateTimeFormatter, si quisiéramos cambiar la localización, deberíamos utilizar el método:

- **DateTimeFormatter withLocale(Locale locale)**. Devuelve una copia del objeto DateTimeFormatter, asociado a la localización proporcionada como parámetro.

Creado el DateTimeFormatter podemos aplicarlo para formatear cualquier objeto fecha/hora de los estudiados. El siguiente código presenta la fecha y hora actuales formateadas de la forma indicada en el ejemplo anterior:

```
DateTimeFormatter dtf;

dtf=DateTimeFormatter.ofPattern("dd/MM/YYYY - HH:mm:ss");

LocalDateTime ldt=LocalDateTime.now();

System.out.println(ldt.format(dtf));
```

Tengamos en cuenta que en el examen podría aparecer un código como el siguiente y preguntarnos qué se mostraría si la fecha y hora actual es 2018-10-23 23:50:10:

```
DateTimeFormatter dtf;

dtf=DateTimeFormatter.ofPattern("dd/mm/YYYY");

LocalDateTime ldt=LocalDateTime.now();

System.out.println(ldt.format(dtf));
```

La respuesta sería 30/50/2018. Ten en cuenta que la m en minúscula son los minutos, y si decidimos que aparezcan los minutos entre medias del día y el año es perfectamente válido.

Parseado de fechas

Todas las clases de fecha/hora que hemos visto disponen de un método estático *parse(String s)* que, a partir de una representación de cadena del valor temporal, nos devuelve el objeto correspondiente. Por ejemplo, para crear un objeto LocalDate a partir de la cadena que representa la fecha sería:

```
LocalDate ld=LocalDate.parse("2017-10-23");
```

Sin embargo, Java es muy estricto en cuanto a la manera en la que deben estar representados los valores de fecha y hora para poder parsear. **Si no coincide exactamente con la forma esperada, se producirá una DateTimeParseException**. Veamos cómo tiene que ser la representación de cadena en cada una de las clases estudiadas para poder aplicar el método parse():

- LocalDate: yyyy-MM-DD
- LocalTime: HH-mm-ss
- LocalDateTime: yyyy-MM-DDT HH-mm-ss
- Instant: yyyy-MM-DDT HH-mm.milisZ

En el caso de LocalDate, LocalTime y LocalDateTime, disponen también de una versión de parse en la que se le puede proporcionar como segundo parámetro un objeto DateTimeFormatter con el formato de la cadena a parsear. Por ejemplo:

```
LocalDate ld2;

ld2=LocalDate.parse("18/11/23",DateTimeFormatter.ofPattern(

                    "yy/MM/dd"));

System.out.println(ld2); //imprime 2018-11-23
```

Clases para intervalos de tiempo

En el paquete java.time encontramos dos clases para manejar intervalos de tiempo, estas son Duration y Period. Al igual que las clases de fecha y hora, **los objetos de estas clases son inmutables**.

CLASE PERIOD

Un objeto Period representa un intervalo de tiempo basado en fecha, del tipo 2 años 4 meses y 10 días. Como el resto de clases que estamos estudiando, Period no dispone de constructores públicos por lo que para crear un objeto de esta clase recurriremos a los siguientes métodos estáticos:

- **static Period of(int years, int months, int days)**. Crea un objeto Period a partir de los años, meses y días indicados. Los valores pueden ser negativos.

- **static Period ofYears(int years).** Crea un objeto Period representado por los años indicados.

- **static Period ofMonths(int months)**. Crea un objeto Period representado por los meses indicados.

- **static Period ofDays(int days)**. Crea un objeto Period representado por los días indicados.

En el caso de la clase Period, el método toString(), llamado cuando se presenta un objeto por pantalla, devuelve una cadena de caracteres con el periodo de tiempo formateado de la siguiente manera: PañosYmesesMdiasD:

```
Period p=Period.of(3, 20, 5);

System.out.println(p); //muestra P3Y20M5D
```

En cuanto a los métodos que proporciona la clase Period para la manipulación de periodos de tiempo, tenemos los siguientes:

- **int getYears(), getMonths() y getDays()**. Devuelven los años, meses y días, respectivamente, asociados al periodo.

- **Period plusYears(long years), plusMonths(long months) y plusDays(long days)**. Devuelven un nuevo objeto Period resultante de la suma del número de años, meses o días indicados.

- **Period minusYears(long years), minusMonths(long months) y minusDays(long days)**. Devuelven un nuevo objeto Period resultante de la resta del número de años, meses o días indicados.

- **Period withYears(long years), withMonths(long months) y withDays(long days)**. Devuelven un nuevo objeto Period resultante de establecer el número de años, meses o días indicados.

- **Period normalized()**. Devuelve un nuevo objeto Period con los años y meses normalizados, es decir, reajustando los valores de años y meses de manera que el valor de estos últimos no sea superior a 11. Para comprender su funcionamiento, veamos el siguiente código de ejemplo:

```
Period p=Period.ofYears(10);

p=p.plusMonths(20);

System.out.println(p); //sin normalizar muestra P10Y20M

System.out.println(p.normalized()); //normalizado

                                    // muestra P11Y8M
```

CLASE DURATION

Representa un intervalo temporal medido con unidades horarias, al estilo 2 horas 20 minutos 40 segundos.

Para crear un objeto Duration emplearemos los siguientes métodos estáticos de la clase:

- **static Duration ofDays(long days)**. Crea un objeto Duration basado en la cantidad de días indicado, aunque internamente, cada día es considerado como 24 horas.

- **static Duration ofHours(long hours)**. Crea un objeto Duration basado en la cantidad de horas indicada.

- **static Duration ofMinutes(long minutes)**. Crea un objeto Duration basado en la cantidad de minutos indicado.

- **static Duration ofSeconds(long seconds)**. Crea un objeto Duration basado en la cantidad de segundos indicado.

El método toString() de Duration devuelve una representación del objeto en la forma: PThorasHminutosMsegundosS:

```
Duration d=Duration.ofDays(2);

System.out.println(d); //muestra PT48H
```

Los siguientes métodos de la clase Duration nos permitirán manipular los datos del objeto:

- **long getSeconds()**. Devuelve la duración del intervalo en segundos.

- **long toDays(), toHours() y toMinutes()**. Devuelve la duración del intervalo en días, horas o minutos, respectivamente.

```
Duration d=Duration.ofDays(2);

System.out.println(d.getSeconds());//muestra 172800

System.out.println(d.toDays()); //muestra 2

System.out.println(d.toHours()); //muestra 48

System.out.println(d.toMinutes()); //muestra 2880
```

- **Duration plusDays(long days), plusHours(long hours), plusMinutes(long minutes) y plusSeconds(long seconds)**. Devuelven una copia del objeto Duration con los días, horas, minutos o segundos añadidos.

- **Duration minusDays(long days), minusHours(long hours), minusMinutes(long minutes) y minusSeconds(long seconds)**. Devuelven una copia del objeto Duration con los días, horas, minutos o segundos restados.

- **Duration withSeconds(long seconds).** Devuelve una copia del objeto Duration con la cantidad de segundos establecidos:

```
Duration d=Duration.ofDays(2);

System.out.println(d.withSeconds(100)); //muestra PT1M40S
```

EXPRESIONES LAMBDA Y PREDICADOS

Las expresiones lambda son una de las principales novedades que se incorporaron en la versión Java 8 y representan el paradigma de la programación funcional.

Son muchos los ámbitos de aplicación de las expresiones lambda, aunque el examen de certificación iz0-808 se centra en el conocimiento de la sintaxis de las expresiones lambda y su utilización en la implementación de predicados, que son una de las principales aplicaciones de uso de estas estructuras.

Además de estos dos puntos, en este apartado también trataremos el uso de las expresiones lambda con los nuevos métodos de colecciones incorporados en la versión Java 8.

Interfaces funcionales

Antes de entrar en detalle sobre las expresiones lambda, vamos a definir el concepto de interfaz funcional, que es en lo que se basan dichas expresiones.

DEFINICIÓN

Una interfaz funcional es una interfaz Java que **cuenta con un único método abstracto**. La siguiente interfaz sería funcional, pues a pesar de incluir métodos default, contiene un único método abstracto:

```
interface InterA{

    default void m(){

        System.out.println("default InterA");

    }

    int test();

}
```

La siguiente interfaz, que hereda InterA, también sería funcional pues solo incorpora una método estático y, por tanto, contaría como único método abstracto el heredado de InterA:

```
interface InterB extends InterA{

    static void print(){

        System.out.println("static InterB");

    }

}
```

Debemos de tener en cuenta que **los métodos abstractos que coincidan con los definidos en la clase Object no son tenidos en cuenta de cara a la característica de la funcionalidad**. Por ejemplo, la siguiente interfaz, a pesar de contar con dos métodos abstractos, seguiría siendo una interfaz funcional puesto que uno de sus métodos abstractos coincide con uno de los métodos de Object:

```
interface InterC extends InterA{

    String toString(); //coincide con método de Object

}
```

La siguiente interfaz, por el contrario, no sería una interfaz funcional pues el único método abstracto que tiene coincide con uno de Object:

```
interface InterD{

    boolean equals(Object ob);

}
```

Esta interfaz, que hereda de una funcional InterA, tampoco se puede considerar funcional porque tendría dos métodos abstractos, aunque estén sobrecargados:

```
interface InterE extends InterA{

    int test(int p);

}
```

Como último ejemplo, esta interfaz tampoco puede considerarse funcional ya que sobrescribe como *default* el método heredado de InterA, lo que significa que no cuenta con ningún método abstracto:

```
interface InterF extends InterA{

    //ya no tiene método abstracto

    default int test(){

        return 10;

    }

}
```

ANOTACIÓN @FUNCTIONALINTERFACE

La anotación @FunctionalInterface se emplea para indicar al compilador que estamos ante una interfaz funcional. Su uso no es ni mucho menos obligatorio, pero en caso de que la interfaz no sea funcional, el compilador nos avisará con un error de compilación, aunque sintácticamente no hayamos cometido ningún error.

Por ejemplo, la siguiente interfaz es sintácticamente correcta y compilaría sin problemas:

```
interface Prueba{

    void run();

    int test();

}
```

Pero si la anotamos con @FunctionalInterface se producirá un error de compilación, dado que no es funcional:

```
@FunctionalInterface

 interface Prueba{

    void run();

    int test();

}
```

Expresiones lambda

A continuación, estudiaremos con detalle esta importantísima novedad incorporada en Java 8, haciendo especial hincapié en la sintaxis para la creación de este tipo de expresiones.

DEFINICIÓN

Una expresión lambda es una **implementación de una interfaz funcional**. Proporciona el código del único método abstracto de la interfaz, a la vez que genera un objeto que implementa la misma.

La estructura de una expresión lambda es la siguiente:

lista_parametros->implementación

Donde *lista_parametros* representa la lista de parámetros del único método abstracto de la interfaz, mientras que implementación constituye el código del método.

Como todo el conjunto devuelve un objeto que implementa la interfaz, la expresión lambda se asignará a una variable del tipo de la interfaz:

Tipo_interfaz variable= lista_parametros->implementación;

Por ejemplo, dada la siguiente interfaz:

```
interface Runnable{

    void run();

}
```

La siguiente instrucción proporcionaría una implementación de la interfaz anterior a través de una expresión lambda y se almacenaría la referencia al objeto en una variable Runnable:

```
Runnable r =()->System.out.println("lambda");
```

SINTAXIS PARA LA CONSTRUCCIÓN DE EXPRESIONES LAMBDA

De cara a la creación de expresiones lambda debemos tener en cuenta unas normas sintácticas, tanto a nivel de lista de parámetros como de implementación.

Lista de parámetros

A la hora de definir la lista de parámetros de una expresión lambda debemos de tener en cuenta las siguientes reglas:

- Si el método no tiene parámetros, se indican paréntesis vacíos ().
- Si dispone de un único parámetro, se puede indicar el parámetro con o sin el tipo. Si se indica tipo, habrá que ponerlo entre paréntesis, pero si no se indica el tipo del parámetro, los paréntesis son opcionales.
- Si el método tiene más de un parámetro, será obligatorio indicarlos entre paréntesis, aunque el tipo de los parámetros es opcional.

Veamos algunos ejemplos de lista de parámetros válidos para una expresión lambda:

```
()-> //el método no recibe parámetros

a-> //método de un parámetro

(int b)-> //método de un parámetro indicando el tipo

(a, b, c)-> //método de tres parámetros

(int x, int y)-> //dos parámetros indicando el tipo
```

Por otro lado, aquí presentamos algunos casos de listas de parámetros definidas de forma incorrecta:

```
-> //si no hay parámetros se debe indicar ()

int a-> //faltan los paréntesis

a,b-> //faltan los paréntesis

(int x, y)-> //se debe indicar el tipo de todos o de ninguno
```

Implementación

Para la implementación del método de la interfaz a través de una expresión lambda, debemos tener en cuenta las siguientes reglas:

- Si la implementación se compone solo de una instrucción, se pueden omitir las llaves, incluso se **puede omitir la palabra *return* cuando devuelve resultado**. Si se quiere indicar return, entonces serán obligatorias las llaves.

- Si hay más de una instrucción, las llaves son obligatorias y si devuelve resultado se deberá indicar return en la instrucción que realiza la devolución del resultado.

Seguidamente, presentamos algunos ejemplos de implementaciones válidas en expresiones lambda:

```
->System.out.println("hello");

//una única instrucción que devuelve resultado y, por tanto

//no hace falta indicar return

->5-3;

->{return true;}

->{a=a+2;return a;}
```

Por el contrario, las siguientes implementaciones serían incorrectas:

```
->return true; //faltan las llaves

->a=5;System.out.println(a); //faltan las llaves
```

EJEMPLO DE EXPRESIÓN LAMBDA

Veamos un ejemplo de implementación de una interfaz funcional concreta mediante expresiones lambda. Supongamos que tenemos la siguiente interfaz:

```
interface Prueba{

    String test(int a);
}
```

Las siguientes instrucciones representarían implementaciones correctas de la interfaz:

```
Prueba p1=n->"hola"+n;

Prueba p2= (int k)->{k++; return String.valueOf(k);};

System.out.println(p1.test(5)); //muestra "hola 5"

System.out.println(p2.test(10));//muestra "11"
```

Referencias a métodos

Cuando la implementación de una expresión lambda se compone de una única instrucción que realiza una llamada a un método, toda la expresión puede ser sustituida por una **referencia a método**. La sintaxis de la referencia a método es:

objeto::metodo

o

clase::metodo

donde los parámetros de llamada a método se determinan implícitamente a partir de los parámetros de llamada al método de la interfaz.

Por ejemplo, dada la expresión lambda:

```
a->System.out.println(a);
```

Toda ella puede ser sustituida por la siguiente referencia a método:

```
System.out::println
```

El parámetro que se pasará a println será el parámetro del método que se implementa.

Por ejemplo, dada la interfaz:

```
interface Test{

    void hacer(int k);

}
```

Una manera de crear un objeto que implemente esta interfaz y llamar al método *hacer()* con un determinado número, sería usando expresiones lambda:

```
Test t=n->System.out.println(n);

t.hacer(10);  //imprime el número 10
```

Lo mismo utilizando referencia a métodos sería:

```
Test t=System.out::prinln;

t.hacer(10);
```

El parámetro que se pasa al método hacer() sería el que se utilizaría en la llamada a *println()*

Veamos otro ejemplo, supongamos la siguiente interfaz:

```
interface Inter{

    int mayor(int a, int b);

}
```

Una implementación de la interfaz mediante expresión lambda sería:

```
Inter in=(a,b)->Math.max(a,b);
```

Empleando referencia a método:

```
Inter in=Math::max;
```

Implementación de predicados: Interfaz Predicate

Uno de los objetivos del examen de certificación es el conocimiento de esta interfaz y su implementación a través de expresiones lambda. Predicate es una interfaz funcional que forma parte del conjunto de nuevas interfaces incorporadas a partir de Java 8, que se encuentran en el paquete java.util.function.

Su método abstracto se llama test(), recibe un parámetro de tipo genérico y devuelve un boolean. Esta es su definición abreviada, omitiendo los métodos estáticos y default presentes en la interfaz:

```
interface Predicate<T>{

    :

    boolean test(T t);

}
```

A continuación, presentamos algunos ejemplos de posibles implementaciones de esta interfaz mediante expresiones lambda:

```
Predicate<Integer> p1=a->a>10;

Predicate<Integer> p2=n->{n=n+7;return n%2==0;};

Predicate<String> p3=s->s.equals("hello");
```

En una hipotética pregunta de examen en la que aparezca una posible implementación de esta interfaz, debemos de fijarnos tanto en la correcta sintaxis de la expresión lambda como en que corresponda a una implementación de la interfaz. Por ejemplo, las siguientes instrucciones generarían errores de compilación por los motivos que se indican en el comentario:

```
//La lista de parámetros no corresponde al método

//test de la interfaz

Predicate<Integer> pd1=()->true;

//la expresión lambda es incorrecta, si se

//usa return la sentencia debe

//escribirse entre llaves

Predicate<Integer> pd2=(n)->return n%2==0;

//la implementación debe devolver un boolean siempre

Predicate<String> pd3=(s)->s.length();
```

Nuevos métodos de colecciones

A partir de Java 8 se han incorporado nuevos métodos a las colecciones de tipo lista, conjuntos y tablas que se basan en la aplicación de la programación funcional para realizar determinadas tareas en la colección, como el recorrido de la misma, eliminación y búsqueda de elementos, etc.

Estos nuevos métodos se basan en la utilización de expresiones lambda como parámetros de los métodos para indicar las tareas a realizar, evitando la utilización de bucles y estructuras de control. Veamos algunos de los más interesantes.

MÉTODO REMOVEIF DE LA INTERFAZ COLLECTION

Este método, incluido en la interfaz Collection, puede ser utilizado tanto con listas (ArrayList) como con conjuntos (HashSet). A continuación se indica el formato del método:

boolean removeIf(Predicate<? super E> filter)

Este método elimina de la colección todos los elementos que cumplen el predicado que se le pasa como parámetro. A pesar de lo extraño que pueda parecer el parámetro del método, simplemente representa una implementación de la interfaz Predicate del tipo de dato de la colección. Habitualmente, esta implementación se proporcionará como una expresión lambda.

El método aplica el método test() del predicado a cada elemento de la colección, eliminando de la misma a todos aquellos elementos para los que la aplicación del método devuelve true.

Por ejemplo, supongamos que tenemos la siguiente lista de enteros.

```
ArrayList<Integer> lista=new ArrayList<>();

lista.add(20); lista.add(10);

lista.add(5); lista.add(12);
```

Si quisiéramos eliminar de la lista todos los números mayores de 10, sería:

```
lista.removeIf(n->n>10);
```

MÉTODO FOREACH DE LA INTERFAZ ITERABLE

Este método puede ser aplicado tanto a listas, como a conjuntos. Su formato es el siguiente:

void forEach(Consumer<? super E> action)

Lo que hace este método es aplicar el método *accept()* de la interfaz funcional Consumer a cada uno de los elementos de la colección. El método *accept()* tiene este formato:

void accept(E e);

Como vemos, recibe un elemento y realiza algún tipo de operación con el mismo, pero sin devolver ningún resultado. Por ejemplo, si quisiéramos mostrar el contenido de la lista del ejemplo anterior, en lugar de utilizar una instrucción for para recorrer cada elemento y mostrarlo, podemos hacerlo con el método *forEach*, pasándole una expresión lambda con la tarea a realizar:

```
lista.forEach(n->System.out.println(n););
```

Lo mismo utilizando referencia a métodos sería:

```
lista.forEach(System.out::println);
```

MÉTODO FOREACH DE HASHMAP

Una de las limitaciones de las tablas antes de la versión Java 8 era que no se podía recorrer directamente la colección completa, o se recorrían claves o valores. Con el método forEach se puede recorrer toda la colección completa y aplicar una acción sobre claves y valores. El formato del método es el siguiente:

forEach(BiConsumer<? super K,? super V> action)

Es similar al forEach de listas y conjuntos, solo que en este caso el parámetro es un BiConsumer, que es una interfaz variante de Consumer en la que el método abstracto acccept, recibe dos parámetros:

void accept(T t, U u);

El método forEach realizará una llamada a este método por cada entrada de la colección, pasando como primer parámetro la clave y como segundo el valor.

Por ejemplo, dada la siguiente tabla:

```
HashMap<Integer,String> datos=new HashMap<>();

datos.put(300,"Ana");

datos.put(4500,"Elena");

datos.put(100,"Juan");

datos.put(3000,"Manuel");
```

Si quisiéramos mostrar por pantalla el nombre unido a su clave, para cada uno de los elementos de la tabla sería:

```
datos.forEach((k,v)->System.out.println(v+"-"+k););
```

PREGUNTAS TIPO EXAMEN

Pregunta 1

Given the following:

```
public static void main(String[] args){
        String cad="  ";
        cad.trim();
        Sustem.out.println(cad.equals("")+" : "+cad.isEmpty());
}
```

What is the result?

A. true : true

B. true : false

C. false : true

D. false : false

Pregunta 2

Given the code fragment:

```
String[] cads=new String[2];
int i=0;
for(String s:cads){
        cads[i].concat(" index"+i);
        i++;
```

```
    }
    for(i=0;i<cads.length;i++){
            System.out.println(cads[i]);
    }
```

What is the result?

A. index 0

index 1

B. Null index 0

Null index 1

C. Null

Null

D. A NullPointerException is thrown at runtime

Pregunta 3

Given:

```
public static void main(String[] args) {
    String str="A  ";
    str=str.concat("B ");
    String str2="C  ";
    str=str.concat(str2);
    str.replace('C','D');
```

```
        str=str.concat(str2);
        System.out.println(str);
```

What is the result?

A. A B C D

B. A C D

C. A B C C

D. A B D

E. A B D C

Pregunta 4

Given the code fragment:

```
public class CCMask {
    public static String maskCC(String card){
            String x="XXXX-XXXX-XXXX-";
            //line 1
    }
    public static void main(String[] args) {
            System.out.println(maskCC("1234-5678-7241-5900"));
    }
}
```

You must ensure that the maskcc method returns a string that hides all digits of the credit card number except the four last digits (and the hyphens that separate each group of four digits).

Which two code fragments should you use at line 1, independently, to achieve this requirement?

A. StringBuilder sb=new StringBuilder(card);

```
sb.substring(15,19);

return x+sb;
```

B. return x+card.substring(15,19);

C. StringBuilder sb=new StringBuilder(x);

```
sb.append(card,15,19);

return sb.toString();
```

D. StringBuilder sb=new StringBuilder(card);

```
StringBuilder s=sb.insert(0,x);

return s.toString();
```

Pregunta 5

Given the code fragment:

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

    List<String> data=new ArrayList<>();

    data.add("one");

    data.add("two");

    data.add("three");

    data.add("two");

    if(data.remove("two")){

            data.remove("four");
```

```
        }
        System.out.println(data);
    }
```

What is the result?

A. [one, three, two]

B. [one, three]

C. [one, two, three, two]

D. An exception is thrown at runtime.

Pregunta 6

Given the following:

```
LocalDate d1=LocalDate.now();
LocalDate d2=LocalDate.of(2017,11,30);
LocalDate d3=LocalDate.parse("2017-11-30",DateTimeFormatter.ISO_DATE);
System.out.println("Date 1= "+d1);
System.out.println("Date 2= "+d2);
System.out.println("Date 3= "+d3);
```

Assume that the system date is November 30, 2017. What is the result?

A.Date 1 = 2017-11-30

Date 2 = 2017-11-30

Date 3 = 2017-11-30

B. Date 1 = 30/11/2017

Date 2 = 2017-11-30

Date 3 = Nov 30, 2017

C.Compilations fails

D.A DateParseException is thrown at runtime

Pregunta 7

Given the code fragment:

```
public static void main(String[] args){
        LocalDate ld=LocalDate.of(2017,02,28);
        ld.plusDays(4);
        System.out.println(ld);
}
```

What is the result?

A. 2017-03-4

B. 2017-02-28

C. Compilation fails

D. A Exception is thrown at runtime

Pregunta 8

Given:

Person.java:

```
public class Person[
    String name;
    int age;
    public Person(String m, int a){
```

```
            name=n;
            age=a;
        }
        public String getName(){
            return name;
        }
        public int getAge(){
            return age;
        }
    }
```

Test.java

```
    public static void checkAge(List<Person> list, Predicate<Person> predicate) {
        for(Person p:list){
            if(predicate.test(p)){
                System.out.println(p.name+"  ");
            }
        }
    }
    public static void main(String[] args){
        List<Person> lst=Arrays.asList(new Person("p1", 44),
                                       new Person("p2",40),
                                       new Person("p3", 35));
```

```
    //line 1
}
```

Which code fragment, when inserted at line 1, enables the code to print p1?

A. checkAge (lst, ( ) -> p. get Age () > 40);

B. checkAge(lst, Person p ->p.getAge() > 40);

C. checkAge (lst, p ->p.getAge() > 40);

D. checkAge(lst, (Person p) -> { p.getAge() > 40; });

Pregunta 9

Given the following:

```
interface Runnable{
        void run();
}
```

Which of the following will create instance of Runnable type (choose 2)?

A. Runnable run = () -> {System.out.println("Run");}

B. Runnable run =void -> System.outprintln("Run");

C. Runnable run = () -> System.out.println("Run");

D. Runnable run = -> System.out.println("Run"};

E. Runnable run = () -> {return System.out.println("Run");}

SOLUCIONES

1. D. La llamada a trim() devuelve una cadena sin los espacios, pero la cadena original no es alterada, por tanto, cad sigue apuntando a una cadena formada por un espacio, así pues, tanto la comparación con cadena vacía como la llamada a isEmpty() resultarán false.

2. D. Al crear el array de cadenas cada una de sus posiciones se inicializa a null, por tanto, la instrucción de llamada a concat: cads[i].concat() provocará un NullPointerException.

3. C. La llamada a str.replace('C','D'); no reemplaza el carácter 'C' por el 'D' en la cadena original, por lo que hasta ese momento, str sigue apuntando a "A B C", como después se le concatena 'C' de nuevo y el resultado se guarda en str, se mostrará "A B C C".

4. B y C. La A no es correcta porque el valor devuelto por el método substring, que serían los cuatro últimos dígitos de la cuenta, no son utilizados. La D tampoco es correcta porque lo que se devolvería es todo el número de cuenta, unido con los caracteres de la máscara.

5. A. La llamada a remove elimina únicamente la primera ocurrencia del elemento.

6. A. Los tres LocalDate representan el mismo valor de fecha y al mostrarlos, se realiza una llamada implícita al método toString() que devuelve la fecha formateada siempre de la forma año-mes-día.

7. B. Los objetos LocalDate son inmutables, la llamada a plusDays devuelve un nuevo objeto resultante de la operación. Como este objeto no se almacena en la variable ld, esta seguirá apuntando al objeto original.

8. C. El segundo parámetro tiene que ser una expresión válida de Predicate. La indicada en A no lo es porque le falta el parámetro, la indicada en B es incorrecta porque al definir el tipo del parámetro hay que ponerlo entre paréntesis y la D no es correcta porque, si se pone la instrucción entre llaves, se debe indicar el return.

9. A y C. B no es correcta porque la definición del parámetro en la expresión lambda es incorrecta, al igual que en D. La E no es correcta porque el método run() no devuelve ningún resultado y no se puede utilizar return.

Índice analítico

E

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W