Java 8 associate
C1 Declarations and Access Control

1Z0-808

Link

https://mylearn.oracle.com/ou/exam/java-se-8-programmer-i-1z0-808/105037/110679/170387
https://docs.oracle.com/javase/specs/jls/se8/html
https://docs.oracle.com/javase/tutorial
http://www.java2s.com

https://enthuware.com
https://github.com/carlosherrerah/OCA

JUAN CARLOS HERRERA HERNANDEZ
carlos.herrera@upa.edu.mx

Contenido

7 197/282 (dia 2: 5 al 7) 6 working with objects

8 172/1777 228

D 37/1535

Java Basics

The Java programming language is an object-oriented programming (OOP) language.

Key Concepts of the Java Programming Language:

- Object-oriented: Modularity, Information-hiding, Code re-use, Plug ability and debugging ease.
- Distributed: such as Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), and Universal Resource Locator (URL). Serialized objects can be sent to other machines, deserialized, and then used as a normal Java object.
- Simple: not allow programmers to directly manipulate pointers to memory locations.
- Multi-threaded: supports multithreading
- Secure: Prohibiting the manipulation of memory by using pointers. The JVM provides a secure "sandbox".
- Platform-independent: Java technology compiler (Java compiler) creating Java bytecode (.class).
- Write once, run anywhere (cross-platform execution).
- Strongly typed.

The programming language also uses a feature called a **garbage collector** to monitor and remove objects that are no longer referred to. Java boolean can have only a true or false value, not 0 or 1.

A **compiler** is an application that converts a program that you write into a CPU-specific code called machine code. These platform-specific files (binary files) are often combined with other files, such as libraries of prewritten code, using a linker to create a platform-dependent program, called an executable that can be executed by an end user.

Bytecodes: La JVM es el intérprete de Java. Ejecuta los bytecodes (archivos compilados con extensión *.class) creados por el compilador de Java.

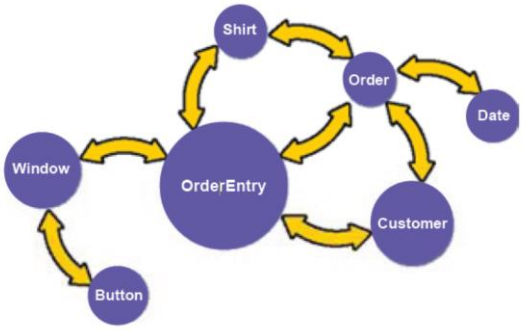
After the **bytecode** is created, it is interpreted (executed) by a bytecode interpreter called the virtual machine or VM. A virtual machine is a platform-specific program that understands platform-independent bytecode and can execute it on a particular platform. For this reason, the Java programming language is often referred to as an interpreted language. Java technology bytecode (Java bytecode) file executing on **several platforms** where a Java runtime environment exists.

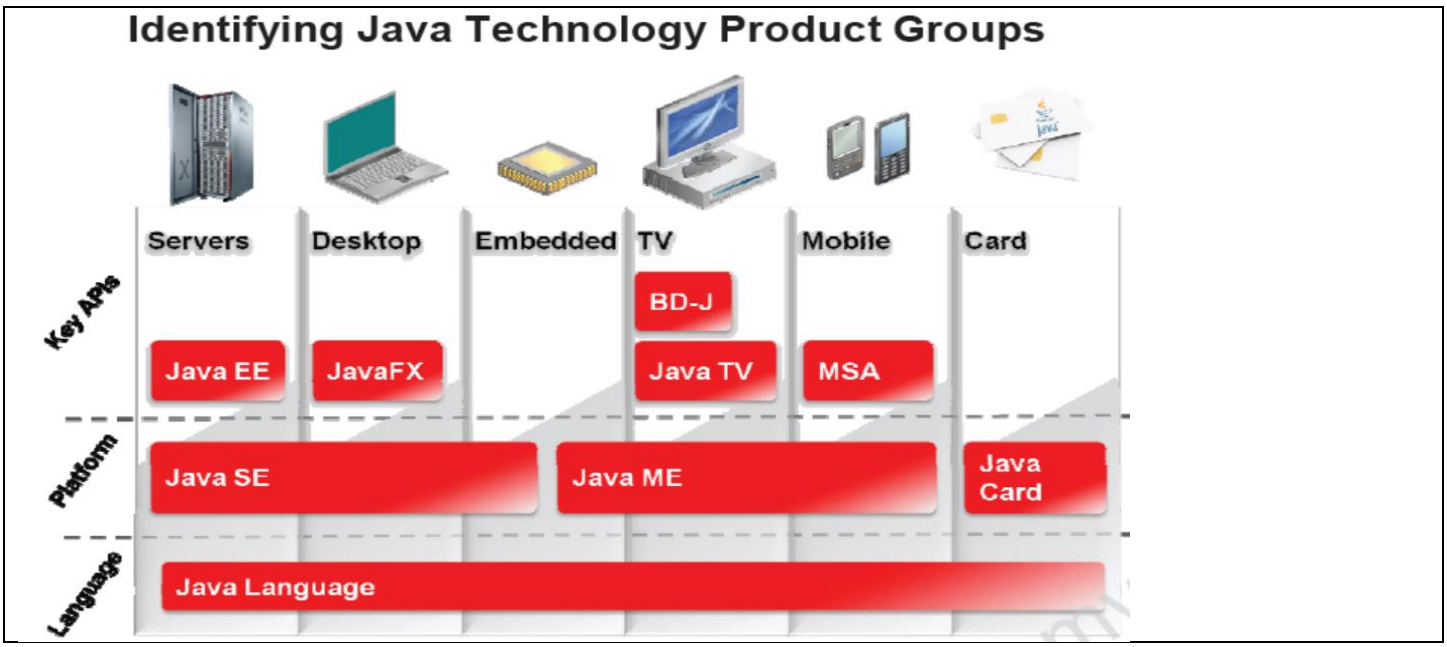
Java Virtual Machine (**JVM**) is required on every platform where your programming will run. The Java Virtual Machine is responsible for interpreting Java technology code, loading Java classes, and executing Java technology programs.

Combined, the JVM software and Java class libraries are referred to as the Java runtime environment (**JRE**).

Each edition includes a Java Development Kit (**JDK**) [also known as a Software Development Kit (**SDK**)] that allows programmers to create, compile, and execute Java technology programs on a particular platform.

Compiladores y Ensambladores: La traducción es fuera de línea, no en tiempo de ejecución.
Interpretes: La traducción es en línea, en tiempo de ejecución.

Procedural programming focuses on sequence	Object-Oriented
	



Java Platform, Standard Edition (**Java SE**) is used to develop applets and applications that run within web browsers and on desktop computers. Java Platform, Enterprise Edition (**Java EE**) is used to create large enterprise, server-side, and client-side distributed applications. Java Platform, Micro Edition (**Java ME**) is used to create applications for resource-constrained consumer devices.

Downloading and Installing the JDK

https://www.oracle.com/java/technologies/downloads/

jre java runtime environment (correr programas)
jdk java development kit (programar)

Variables de ambiente

Panel de control -> Sistema -> Configuracion avanzada del sistema
Opciones avanzadas -> Variables de entorno -> Variables de Usuario

JAVA_HOME C:\Program Files\Java\jdk-1.8	PATH %JAVA_HOME%\BIN
CLASSPATH (solo uno) . ; %JAVA_HOME%\LIB %JAVA_HOME%\LIB ; .	Probar Instalación desde CMD C:\>java -version C:\>javac -version

Working with IDEs There are several available IDEs: – NetBeans IDE from Oracle – JDeveloper from Oracle – Eclipse from IBM – SpringToolSuite4 – IntelliJ – VS Code	Product Life Cycle (PLC) Stages 1. Analysis 2. Design 3. Development 4. Testing 5. Implementation 6. Maintenance 7. End-of-life (EOL)

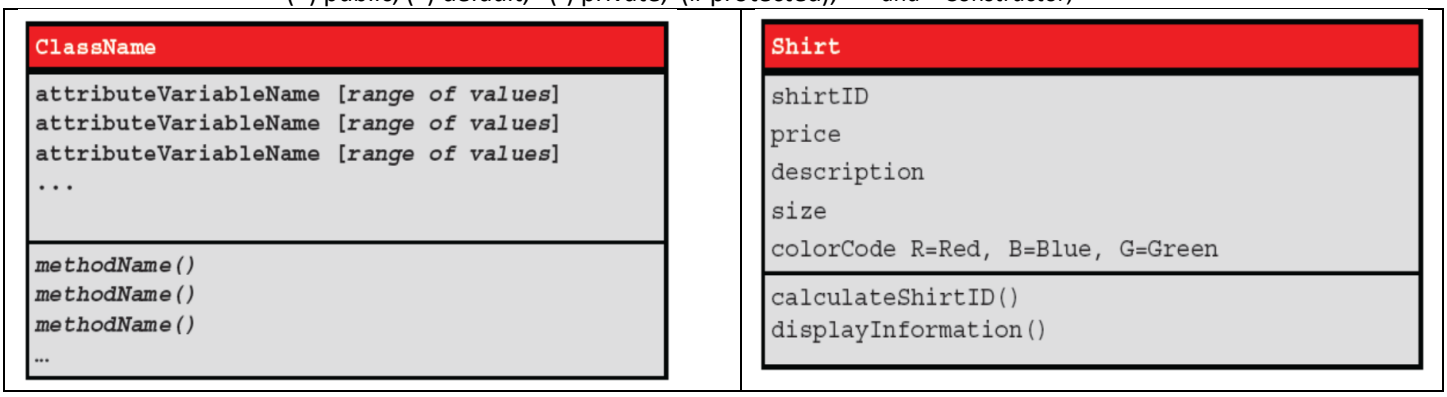
Thinking in Objects

Identifying Objects

- Objects can be physical or conceptual.
- Objects have attributes (characteristics) such as size, name, shape, and so on.
- Objects have operations (things that they can do) such as setting a value, or increasing speed.

Object names are often nouns, such as "account" or "shirt." **Object attributes** are often nouns, too, such as "color" or "size." **Object operations** are usually verbs or noun-verb combinations, such as "display" or "submit order." Your ability to recognize objects in the world around you will help you define objects better when you approach a problem using object-oriented analysis.

Modeling Classes: Class, Object, State(instance variables), Behavior(methods)
(+) public, ( ) default, (-) private, (# protected), « and » Constructor,



UML "Universal Modeling Language" is used to:

- Model the objects, attributes, operations, and relationships in object-oriented programs
- Model the dynamic behavior of the system by showing collaborations among objects and changes to the internal states of objects

Symbols Used in Defining a Java Source

Braces	{ }	llaves	block of code.
Brackets	[ ]	corchetes	optional modifier or Arrays
Parentheses	( )	Parentesis	arguments.
Semicolons	;	punto y coma	signify the end of a statement.
Commas	,	coma	can separate multiple arguments and values.
Single quotation marks	' '	comilla simple	define single characters.
Double quotation marks	" "	comilla doble	define a string of multiple characters.
Double forward slashes	//		indicate a single-line comment.
	/* abc */		comments in block.

There is a third type of comment called a documentation comment (Javadoc).
Must begin with a forward slash and two asterisks (/**) and end with an asterisk and a forward slash (*/).
Estar dentro de la carpeta "src" para poder compilar y correr.

Compiling a Program: javac Nombre_clase.java Generate: Nombre_clase.class
Executing (Testing) a Program: java Nombre_clase

Javac –cp <path> Le indica donde se encuentra el classpath
Javac –d <directory> Le indica donde dejar los .class

Java –cp <path> Le indica donde se encuentra el classpath

1. Declarations and Access Control

1.1. Class Definition

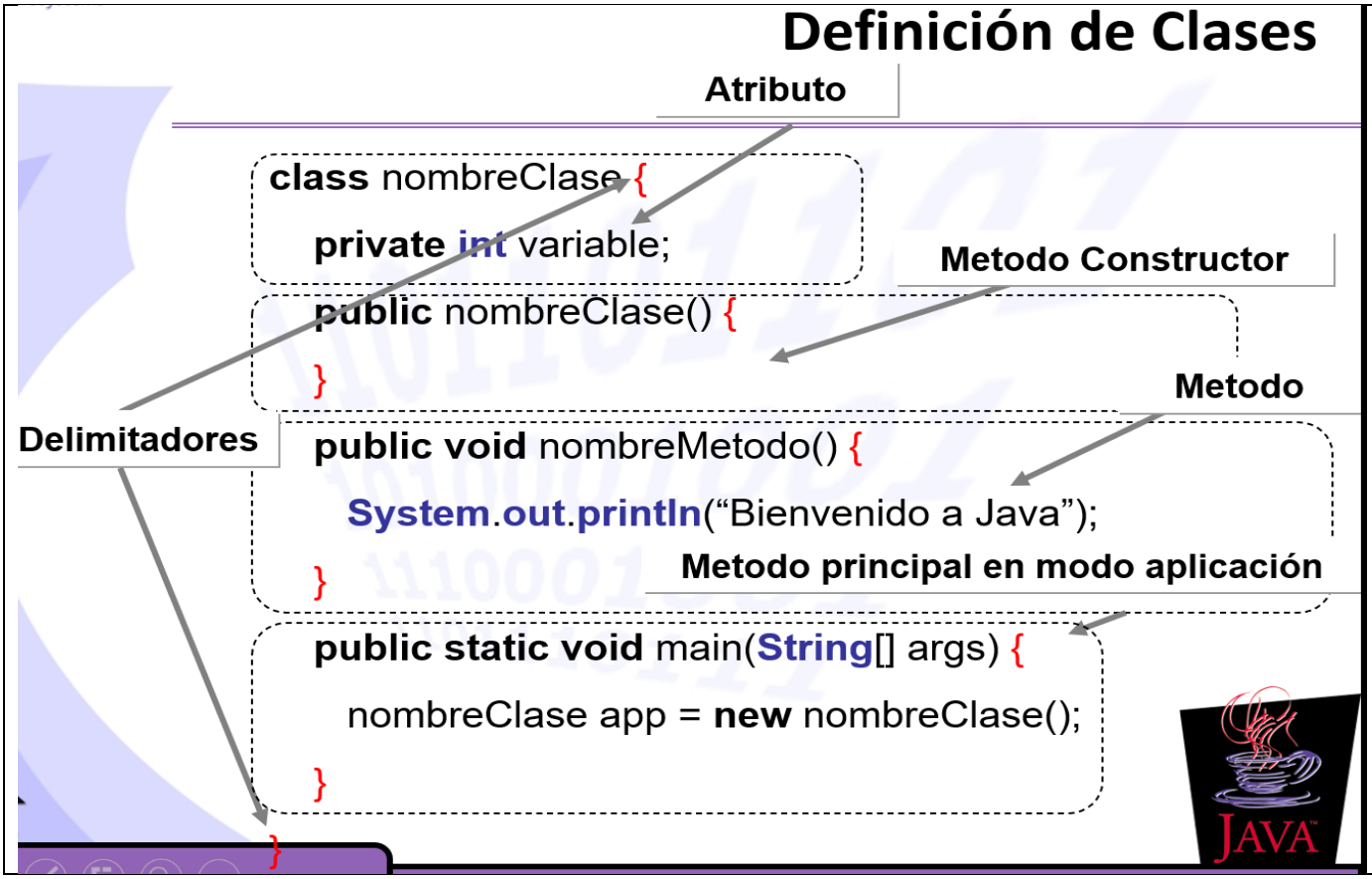
```
int studentGrade=75;
if (studentGrade >= 90)
  System.out.println("A");
else
  if (studentGrade >= 80)
    System.out.println("B");
  else
    if (studentGrade >= 70)
      System.out.println("C");
    else
      if (studentGrade >= 60)
        System.out.println("D");
      else
        System.out.println("F");
```

```
int studentGrade = 75;
if (studentGrade >= 90)
  System.out.println("A");
else if (studentGrade >= 80)
  System.out.println("B");
else if (studentGrade >= 70)
  System.out.println("C");
else if (studentGrade >= 60)
  System.out.println("D");
else
  System.out.println("F");
```

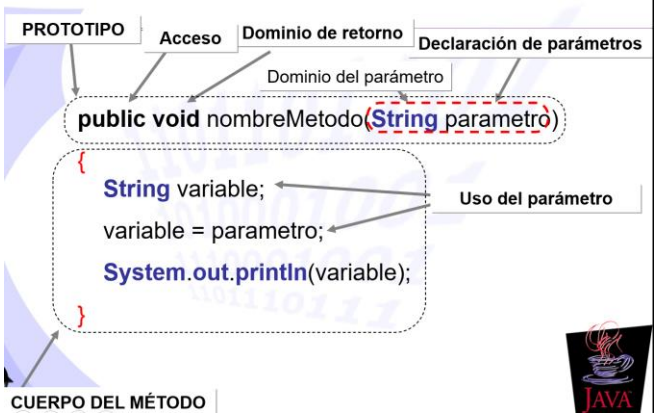
```
1. class Wombat implements Runnable {
2. private int i;
3. public synchronized void run() {
4. if (i%5 != 0) { i++; }
5. for(int x=0; x<5; x++, i++)
6. { if (x > 1) Thread.yield(); }
7. System.out.print(i + " ");
8. }
9. public static void main(String[] args) {
10. Wombat n = new Wombat();
11. for(int x=100; x>0; --x) { new Thread(n).start(); }
12. }}
```

```
class Wombat implements Runnable {
  private int i;

  public synchronized void run() {
    if (i % 5 != 0) {
      i++;
    }
    for (int x = 0; x < 5; x++, i++) {
      if (x > 1)
        Thread.yield();
    }
    System.out.print(i + " ");
  }

  public static void main(String[] args) {
    Wombat n = new Wombat();
    for (int x = 100; x > 0; --x) {
      new Thread(n).start();
    }
  }
}
```

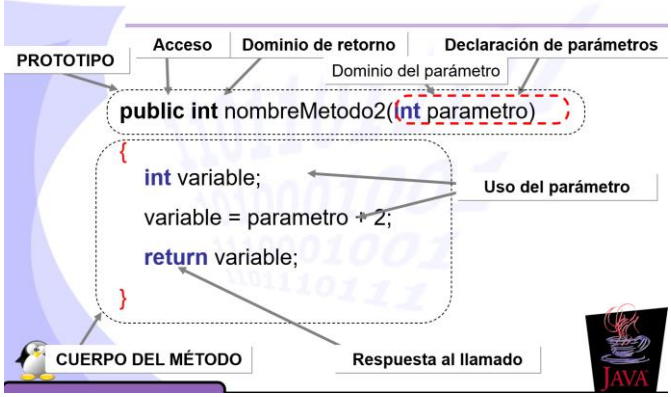
Definición de Clases



Definición de un método



Definición de un método (2)



```
package path.subpath;
import path.subpath.NombreClase;
import path.subpath.*;
```

Syntax for declaring a class:

```
[modificadorDeAcceso] class NombreClase [extends NombreSuperClase][implements Interface1, Interface2, … ]{

        //atributos de la clase (0 ó más atributos)
        [modificadorDeAcceso]  tipo nombreAtributo;

        // Constructores
         [modifier] <class_name> (<argument>*) {
            <statement>*
          }

        //métodos de la clase (o más métodos)
        [modificadorDeAcceso]   tipoDevuelto   nombreMetodo([lista   parámetros])   [throws listaExcepciones]{
                              // instrucciones del método
                              [return valor;]
        }
}
```

Syntax for main method: psvm or main

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

The main method accepts zero or more objects of type String (String[] args).

Any method can return at most one value. If the method returns nothing, the keyword **void** must be used as the return type.

`Modifiers for class:  public, abstract or final`

Modificador	Misma Clase	Misma Paquete	Subclase	Universo
private	Si			
default	Si	Si		
protected	Si	Si	Si	
public	Si	Si	Si	Si

`modificadorDeAcceso:`

Public:	Proyecto
Protected:	Paquete o Subclase (sin importar ubicación)
Default o Friendly:	Paquete
Private:	Clase

IDE Debugger: A debugger lets you place breakpoints in your source code, add field watches, step through your code, run into methods, take snapshots, and monitor execution as it occurs. This is helpful in solving logic problems.

Practice 4-3 : Exploring the Debugger

1.2. Java Refresher

Class
Object
State (instance variables)
Behavior (methods)
Inheritance
Interfaces

1.3. Identifiers and Keywords

1.3.1. Legal Identifiers

- Identifiers must start with a letter, a currency character ($), or a connecting character such as the underscore (_). Identifiers cannot start with a digit!
- After the first character, identifiers can contain any combination of letters, currency characters, connecting characters, or numbers.
- In practice, there is no limit to the number of characters an identifier can contain.
- You can't use a Java keyword as an identifier.
- Identifiers in Java are case sensitive; foo and FOO are two different identifiers.

1.3.2. Keyboards

abstract	default	for	package	synchronized
assert	do	if	private	this
boolean	double	implements	protected	throw
break	else	import	public	throws
byte	enum	instanceof	return	transient
case	extends	int	short	true
catch	false	interface	static	try
char	final	long	strictfp	void
class	finally	native	super	volatile
continue	float	new	switch	while

1.3.3. Oracle's Java Code Conventions

Java provides programmers the ability to create objects that are well encapsulated.
Java provides programmers the ability to send Java objects from one machine to another.

- **Classes and interfaces** the first letter should be capitalized, and if several words are linked together to form the name, the first letter of the inner words should be uppercase (a format that's sometimes called "CamelCase"). **For classes**, the names should typically be **nouns**. Here are some examples:
  Dog
  Account
  PrintWriter

For interfaces, the names should typically be **adjectives**, like these:

Runnable
Serializable

- **Methods** The first letter should be lowercase, and then normal CamelCase rules should be used. In addition, the names should typically be **verb-noun** pairs. For example:

  getBalance
  doCalculation
  setCustomerName

- **Variables** Like methods, the CamelCase format should be used, but starting with a lowercase letter. Oracle recommends short, meaningful names, which sounds good to us. Some examples:

  buttonWidth
  accountBalance
  myString

- **Constants** Java constants are created by marking variables `[static] final`. They should be named using uppercase letters with underscore characters as separators:

  MIN_HEIGHT

1.4. Define Classes

1.4.1. Source File Declaration Rules

- There can be only one public class per source code file.
- If there is a public class in a file, the name of the file must match the name of the public class. For example, a class declared as public class Dog { } must be in a source code file named Dog.java.
- A file can have more than one non-public class.
- Files with no public classes can have a name that does not match any of the classes in the file.
- Classes in the same package are implicitly imported into the source-code files of other classes in the same package.
- Class String is in package java.lang, which is imported implicitly into all source-code files.

Couple of rules for using static imports:

You must say `import static`; you can't say `static import`.

```
import java.util.*;
import java.util.ArrayList;
import java.util.Calendar;
import static java.lang.System.*;
```

The main() method can be overloaded:

```
public static void main(String[] args)         // default
static public void main(String…  x)
public static void main(String argumentos[])
```

System.out.println("\033[H\033[2J"); // limpiar pantalla	

1.4.2. Class Declarations and Modifiers

Access modifiers (`public`, default, `private`, `protected`)
Nonaccess modifiers (including `strictfp`, `final`, and `abstract`)

A **class** can be declared with only `public` or `default` access; the other two access control levels don't make sense for a class.

the `final` keyword means the class can't be subclassed. In other words, no other class can ever extend (inherit from) a final class, and any attempts to do so will result in a compiler error.

You must never, ever, ever mark a class as both `final` and `abstract`. An abstract class can never be instantiated. if even a single method is abstract, the whole class must be declared abstract.

The first concrete class to extend an `abstract` class must implement all of its `abstract` methods.

Coding with abstract class types (including interfaces) lets you take advantage of `polymorphism` and gives you the greatest degree of flexibility and extensibility.

TABLE 1-2 Determining Access to Class Members

Visibility	Public	Protected	*Default*	Private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes, *through inheritance*	No	No
From any nonsubclass class outside the package	Yes	No	No	No

1.5. Use Interfaces

1.5.1. Declaring an Interface

When you create an interface, you are defining a contract for what a class can do, without saying anything about how the class will do it.

An **interface** is `public and abstract`. When you implement the interface, all interface methods must be implemented and must be marked **`public`**. However, although an abstract class can define both abstract and non-abstract methods, an interface generally has only abstract methods.

These rules are strict:

- An interface must be declared with the keyword `interface`.
- If there is a public interface in a file, the name of the file must match the name of the public interface.
- Interface methods are implicitly `public` and `abstract`. In other words, you do not need to actually type the `public` or `abstract` modifiers in the method declaration, but the method is still always `public` and `abstract`.
- Usually interfaces have only `abstract` methods.
- All variables defined in an interface must be `public`, `static`, and `final` in other words, interfaces can declare only constants, not instance variables.
- Interface methods cannot be marked `final`, `strictfp` or `native`.
- An interface can *extend* one or more other interfaces.

  interface C extends A, B { void methodC(); }
- An interface cannot extend anything but another interface.

  interface MyInterface extends MyClass { // cause an error
- An interface cannot implement another interface or class.
- Interface types can be used polymorphically.
- As of Java 8, interfaces can have **concrete methods** declared as either default or static.

The following is a legal interface declaration:

```
public abstract interface Rollable { }
public interface Rollable { }
abstract interface Rollable { }
interface Rollable { }

public interface Bounceable {
    [ public static final ] int TOP = 20;
    void hello();
    public void bounce();
    abstract public int getBounceFactor();
    public abstract void setBounceFactor(int bf);
```

1.5.2. Declaring Interface Constants

You are allowed to put **constants in an interface**. You need to remember one key rule for interface constants. They must always be ***public static final***. But you don't actually have to declare them that way. You can't change the value of a constant! Once the value has been assigned, the value can never be modified.

1.5.3. Declaring `default` Interface Methods

- `default` methods are declared by using the `default` keyword. The `default` keyword can be used only with interface method signatures, not class method signatures.
- `default` methods are `public` by definition, and the `public` modifier is optional.
- `default` methods **cannot** be marked as `private`, `protected`, `static`, `final`, or `abstract`.
- `default` methods must have a concrete method body.

1.5.4. Declaring `static` Interface Methods

- `static` interface methods are declared by using the `static` keyword.
- `static` interface methods are public by default, and the `public` modifier is optional.
- `static` interface methods cannot be marked as `private`, `protected`, `default`, `final` or `abstract`.
- `static` interface methods must have a concrete method body.
- When invoking a `static` interface method, the method's type (interface name) MUST be included in the invocation. Ej: MyInterface.m1();

Diferencias clave de métodos en la interface:

Característica	`default`	`static`
Acceso	A través de instancias de clases que implementen la interfaz	A través de la interfaz directamente
Sobrescritura	Puede ser sobrescrito en la clase que implementa la interfaz	No puede ser sobrescrito
Propósito	Proporcionar implementación por defecto para clases	Métodos utilitarios relacionados con la interfaz, pero no ligados a instancias
Herencia	Se heredan por las clases que implementan la interfaz	No se heredan

En resumen, usas **métodos `default`** cuando deseas proporcionar una implementación común que las clases pueden heredar o sobrescribir, mientras que los **métodos `static`** son para definir métodos que no dependen de las instancias de las clases que implementan la interfaz y se usan principalmente para utilidades.

1.6. Declare Class Members

1.6.1. Access Modifiers

Methods and instance (nonlocal) variables are collectively known as *members*.
You can modify a member with both access and nonaccess modifiers.

Field declarations (class attributes are called "fields"), may also be initialized at declaration time.

Whereas a *class* can use just two of the four access control levels (`default` or `public`), members can use all four: `public` `protected` default `private`.

private > default > protected > public (where private is most restrictive and public is least restrictive).

Public < protected < default < private (from visual to restrictive)

If the class itself will not be visible to another class, then none of the members will be visible either, even if the member is declared `public`. Once you've confirmed that the class is visible, then it makes sense to look at access levels on individual members.

A method marked `private` cannot be overridden.
`private` members can be accessed only by code in the same class.
`private` members are not visible to subclasses, so `private` members cannot be inherited.
Declaring instance variables with access modifier private is known as data hiding.

The `protected` and `default` access control levels are almost identical, but with one critical difference. A *default* member may be accessed only if the class accessing the member belongs to the same package, whereas a `protected` member can be accessed (through inheritance) by a subclass ***even if the subclass is in a different package***.

For a subclass outside the package, the protected member can be accessed only through inheritance.

Can access modifiers be applied to local variables? NO!

Method arguments are essentially the same as local variables. They can also have the modifier `final`, means it can't be modified within the method.

```
public Record getRecord(int fileNumber, final int recNumber) { }
```

An `abstract` method is a method that's been *declared* (as `abstract`) but not *implemented*. In other words, *it has no method body*.

```
public abstract void showSample();
```

Concrete just means nonabstract.

Any class that extends an `abstract` class must implement all `abstract` methods of the superclass, unless the subclass is *also* `abstract`. The rule is this:
The first concrete subclass of an `abstract` class must implement *all* `abstract` methods of the superclass.

An overloaded method (a method using the same identifier, but different arguments),

A method can never, ever, ever be marked as both `abstract` and `final`, or both `abstract` and `private`.

The `abstract` modifier can never be combined with the `static` modifier.

The `synchronized` keyword indicates that a **method** can be accessed by only one thread at a time. Example `public synchronized Record retrieveUserInfo(int id) { }`

The `synchronized` modifier applies only to methods and code blocks.
`synchronized` methods can have any access control and can also be marked `final`. **OCP**

The `native` modifier indicates that a **method** is implemented in platform dependent code, often in C. **OCP**

You can declare a **method** as `strictfp`. Remember, `strictfp` forces floating points. **OCP**

"X extends Y" is correct if X and Y are either both classes or both interfaces

Methods with Variable Argument Lists (var-args)

arguments The things you specify between the parentheses when you're *invoking* a method.
Example: doStuff("a", 2);

parameters The things in the *method's signature* that indicate what the method must receive when it's invoked. Example: Void doStuff(string s, int a) { }

var-arg: Type . . .

Legal

```
void doStuff(int... x) { }              // expects from 0 to many ints
                                        // as parameters
void doStuff2(char c, int... x)  { }    // expects first a char,
                                        // then 0 to many ints
void doStuff3(Animal... animal) { }     // 0 to many Animals
```

Ilegal

```
void doStuff4(int x...) { }             // bad syntax
void doStuff5(int... x, char... y) { }  // too many var-args
void doStuff6(String... s, byte b) { }  // var-arg must be last
```

1.6.2. Constructor Declarations

Every time you make a new object, at least one constructor is invoked. Every class has a constructor, although if you don't create one explicitly, the compiler will build one for you.

- A constructor can't ever, ever, ever, have a return type.
- They must have the same name as the class in which they are declared.
- Have all of the normal access modifiers, and they can take arguments (including var-args), just like methods.
- Constructors can't be marked `static, final` or `abstract.`

If a class does not define constructors, the compiler provides a default constructor with no parameters, and the class's instance variables are initialized to their default values.

1.6.3. Variable Declarations

There are two types of variables in Java:
Primitives: one of eight types: `char`, `boolean`, `byte`, `short`, `int`, `long`, `float` or `double`.
Reference variables: is used to refer to (or access) an object. Example Dog myDog;

Type	Bits	Bytes	Minimum Range	Maximum Range
byte	8	1	-2^7	$2^7 - 1$
short	16	2	-2^{15}	$2^{15} - 1$
int	32	4	-2^{31}	$2^{31} - 1$
long	64	8	-2^{63}	$2^{63} - 1$
float	32	4	n/a	n/a
double	64	8	n/a	n/a

boolean		true or false	
[*Note:* A boolean's representation is specific to the Java Virtual Machine on each platform.]			
char	16	'\u0000' to '\uFFFF' (0 to 65535)	(ISO Unicode character set)

Primitive variables can be declared as:

- Class variables (statics)
- Instance variables (object)
- Method parameters
- Local variables: It is a variable declared within a method. It is not initialized, so you must do it.

Instance variables are defined inside the class, but outside of any method, and are initialized only when the class is instantiated. Instance variables are the fields that belong to each unique object. Only instance variables are initialized automatically, but a local variable does not.

- Can use any of the four access *levels*
- Can be marked `final, transient`
- Cannot be marked `abstract, synchronized, strictfp, native, static`

Primitive-type instance variables are initialized by default.
Variables of types byte, char, short, int, long, float and double are initialized to 0.
Variables of type boolean are initialized to false.
The default value for a field of type String (or any other reference type) is null.

Reference-type fields are initialized by default to the value null.

Local Variables	Variables (nonlocal)	Methods
final	final public protected private static transient volatile	final public protected private static abstract synchronized strictfp native

Local variables are always on the stack, not the heap. If the variable is an object reference, the object itself will still be created on the heap. It must be *initialized* with a value. Local variables don't get default values. A local variable can't be referenced in any code outside the method in which it's declared.

- Local (method, automatic, or stack) variable declarations cannot have access modifiers.
- `final` is the only modifier available to local variables.
- Local variables don't get default values, so they must be initialized before use.

It is possible to declare a local variable with the same name as an instance variable. It's known as *shadowing. to distinguish it is necessary to use the reserved word "this.*

Most operations result in `int` or `long`:
• `byte`, `char`, and `short` values are promoted to `int` before the operation.
• If either argument is of the `long` type, the other is also promoted to `long`, and the result is of the `long` type.

Final Variables

Declaring a variable with the `final` keyword makes it impossible to reassign that variable once it has been initialized with an explicit value. The data within the object can be modified, but the reference variable cannot be changed. Effect of `final` on variables, methods, and clases.

Transient Variables

If you mark an instance variable as `transient`, you're telling the JVM to skip (ignore) this variable when you attempt to **serialize** the object containing it. Serialization lets you save (sometimes called "flatten") an object by writing its state (in other words, the value of its instance variables) to a special type of I/O stream.

The `volatile` modifier applies only to instance variables. **OCP**

Cuando una variable es declarada como `volatile`, el valor de esa variable no se almacenará en cachés de los hilos individuales, sino que siempre será leído directamente desde la **memoria principal**. Esto garantiza que si un hilo cambia el valor de la variable, ese cambio será visible para otros hilos inmediatamente.

Static Variables and Methods

The `static` modifier is used to create variables and methods that will exist independently of any instances created for the class. All `static` members exist before you ever make a new instance of a class. In other words, all instances of a given class share the same value for any given `static` variable. `static` methods do not have direct access to nonstatic members.

Things you can mark as `static`:	**Things you can't mark as `static`:**
Methods	Constructors
Variables	Local variables
Initialization blocks	

Array Declarations

It is never legal to include the size of an array in the declaration.
Arrays can hold either primitives or object references, but an array itself will always be an object on the heap, even if the array is declared to hold primitive elements.

Arrays are efficient, but many times you'll want to use one of the Collection types from java.util (including `HashMap, ArrayList, and TreeSet`).
Collection classes offer more flexible ways to access an object (for insertion, deletion, reading, and so on) and, unlike arrays, can expand or contract dynamically as you add or remove elements.

```
int[] dato          int dato[]
int[][] dato        int dato[][]        int[] dato[]
```

It is never legal to include the size of the array in your declaration.

```
int[5] scores;   // Compilation Error
int[] scores1 = new int[5];
int[] scores2;
scores2 = new int[5];
```

The preceding code won't compile. Remember, the JVM doesn't allocate space until you actually instantiate the array object. That's when size matters.

1.7. Declare and Use enums

An `enum` specifies a list of constant values assigned to a type.
An `enum` can be declared outside or inside a class, but NOT in a method.
The semicolon at the end of an `enum` declaration is optional.

Java lets you restrict a variable to having one of only a few predefined values—in other words, one value from an enumerated list. (The items in the enumerated list are called, surprisingly, `enum`s.). Using `enum`s can help reduce the bugs in your code.

Every `enum` has a static method, `values()`, that returns an array of the `enum`'s values in the order they're declared.

You can NEVER invoke an `enum` constructor directly.

`enum`s can be declared very simply, or they can be quite complex—including such attributes as methods, variables, constructors.

```
enum CoffeeSize { BIG, HUGE, OVERWHELMING };

public class Enums {


      public static void main(String[] args) {
            CoffeeSize myCoffee = CoffeeSize.BIG;
            myCoffee = myCoffee.values()[1];
            System.out.println(myCoffee);
            System.out.println(myCoffee.ordinal());
      }
}
```