The **marker interface pattern** is a [design pattern](http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29) in [computer science](http://en.wikipedia.org/wiki/Computer_science), used with languages that provide run-time type information about objects. It provides a means to associate metadata with a class where the language does not have explicit support for such metadata.

To use this pattern, a [class](http://en.wikipedia.org/wiki/Class_%28computer_science%29) implements a **marker interface**[[1]](http://en.wikipedia.org/wiki/Marker_interface_pattern" \l "cite_note-EffectiveJava-1) (also called **tagging interface**), and methods that interact with instances of that class test for the existence of the interface. Whereas a typical [interface](http://en.wikipedia.org/wiki/Interface_%28computer_science%29) specifies functionality (in the form of method declarations) that an implementing class must support, a marker interface need not do so. The mere presence of such an interface indicates specific behavior on the part of the implementing class. Hybrid interfaces, which both act as markers and specify required methods, are possible but may prove confusing if improperly used.

An example of the application of marker interfaces from the [Java programming language](http://en.wikipedia.org/wiki/Java_%28programming_language%29) is the [Serializable](http://docs.oracle.com/javase/8/docs/api/java/io/Serializable.html) interface. A class implements this interface to indicate that its non-[transient](http://en.wikipedia.org/wiki/Transient_%28computer_programming%29) data members can be written to an [ObjectOutputStream](http://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html). The ObjectOutputStream private method writeObject() contains a series of instanceof tests to determine writeability, one of which looks for the Serializable interface. If any of these tests fails, the method throws a NotSerializableException.

**Critique**

A major problem with marker interfaces is that an interface defines a contract for implementing classes, and that contract is inherited by all subclasses. This means that you cannot "unimplement" a marker. In the example given, if you create a subclass that you do not want to serialize (perhaps because it depends on transient state), you must resort to explicitly throwing NotSerializableException (per ObjectOutputStream docs)

Another solution is for the language to support [metadata](http://en.wikipedia.org/wiki/Metadata) directly:

* Both the [.NET framework](http://en.wikipedia.org/wiki/.NET_Framework) and [Java](http://en.wikipedia.org/wiki/Java_%28software_platform%29) (as of Java 5 (1.5)) provide support for such metadata. In .NET, they are called *"custom attributes"*, in Java they are called *"annotations"*. Despite the different name, they are conceptually the same thing. They can be defined on classes, member variables, methods, and method parameters and may be accessed using [reflection](http://en.wikipedia.org/wiki/Reflection_%28computer_science%29).
* In [Python](http://en.wikipedia.org/wiki/Python_%28programming_language%29), the term "marker interface" is common in [Zope](http://en.wikipedia.org/wiki/Zope) and [Plone](http://en.wikipedia.org/wiki/Plone_%28software%29). Interfaces are declared as metadata and subclasses can use implementsOnly to declare they do not implement everything from their super classes.
* **What are Marker Interfaces in Java?**  
    
  An empty interface having no methods or fields/constants is called a marker interface or a tag interface. This of course means if the interface is extending other interfaces (directly or indirectly) then the super interfaces must not have any inheritable member (method or field/constant) as otherwise the definition of the marker interface (an entirely empty interface) would not be met. Since members of any interface are by default 'public' so all members will be inheritable and hence we can say for an interface to be a marker interface, all of its direct or indirect super interfaces should also be marker. (Thanks **marco** for raising the point. I thought it was obvious, but mentioning all this explicitly would probably help our readers.)  
    
  There are few Java supplied marker interfaces like Cloneable, Serializable, etc. One can create their own marker interfaces the same way as they create any other interface in Java.  
    
  **Purpose of having marker interfaces in Java i.e., why to have marker interfaces?**  
    
  The main purpose to have marker interfaces is to create special types in those cases where the types themselves have no behavior particular to them. If there is no behavior then why to have an interface? Because the implementor of the class might only need to flag that it belongs to that particular type and everything else is handled/done by some other unit - either internal to Java (as in the case of Java supplied standard marker interfaces) or an app specific external unit.  
    
  Let's understand this by two examples - one in which we will discuss the purpose of a standard Java interface (Cloneable) and then another user-created marker interface.  
    
  **What purpose does the Cloneable interface serve?**  
    
  When JVM sees a clone() method being invoked on an object, it first verifies if the underlying class has implemented the 'Cloneable' interface or not. If not, then it throws the exception CloneNotSupportedException. Assuming the underlying class has implemented the 'Cloneable' interface, JVM does some internal work (maybe by calling some method) to facilitate the cloning operation. Cloneable is a marker interface and having no behavior declared in it for the implementing class to define because the behavior is to be supported by JVM and not the implementing classes (maybe because it's too tricky, generic, or low-level at the implementing class level). So, effectively marker interfaces kind of send out a signal to the corresponding external/internal entity (JVM in case of Cloneable) for them to arrange for the necessary functionality.  
    
  How does JVM support the 'cloning' functionality - probably by using a native method call as cloning mechanism involves some low-level tasks which are probably not possible with using a direct Java method. So, a possible 'Object.clone' implementation would be something like this:-
* public Object clone() throws CloneNotSupportedException {  
    
   if (this implements Cloneable)  
    
   return nativeCloneImpl();  
    
   else  
    
   throw new CloneNotSupportedException();  
    
  }
* Anyone wondered as to why and when do we get 'CloneNotSupportedException' exception at compile-time itself? Well... that's no trick. If you see the signature of the 'Object.clone()' method carefully, you will see a throws clause associated with it. I'm sure how can you get rid of it: (i) by wrapping the clone-invocation code within appropriate try-catch (ii) throwing the CloneNotSupportedException from the calling method.  
    
  What purpose does a user-defined marker interface serve? It can well serve the same purpose as by any standard marker interface, but in that case the container (the module controlling the execution of the app) has to take the onus of making sure that whenever a class implements that interface it does the required work to support the underlying behavior - the way JVM does for Cloneable or any other standard marker interface for that matter.  
    
  **Defining an user-defined marker interface in Java**  
    
  Let's define a user-defined marker interface. Let's say there is an app suporting a medical store inventory and suppose you need a reporting showing the sale, revenue, profit, etc. of three types of medicines - allopathic, homeopathic, and ayurvedic separately. Now all you need is to define three marker interfaces and make your products (medicines) implement the corresponding ones.
* public interface Allopathic{}  
  public interface Homeopathic{}  
  public interface Ayurvedic{}
* In your reporting modules, you can probably get the segregation using something similar to below:-
* for (Medicine medicine : allMedicines) {  
  if (medicine instanceof Allopathic) {  
  //... update stats accordingly  
  }  
  else if (medicine instanceof Homeopathic) {  
  //... update stats accordingly  
  }  
  else if (medicine instanceofAyurvedic) {  
  //... update stats accordingly  
  }  
  else {  
  //... handle stats for general items  
  }  
  }
* As you can see the medicines themselves don't need to implement any specific behavior based on whether they are allopathic, homeopathic, or ayurvedic. All they need is to have a way of reflecting which category they belong to, which will in turn help the reporting modules to prepare the stats accordingly.  
    
  Now this can be done by having a flag as well... yeah, sure it can be. But, don't you think tagging a class makes it more readable than having a flag indicating the same. You kind of make it an implementation-independent stuff for the consumers of your classes. If your class implements an interface, it becomes part of the class signature of the published API. Otherwise, you would probably handle the situation by having a public final field having the flag set up at the time of instantiation - final because you would not like others to change it. I guess going the marker interface way would probably make more sense in many such situations.  
    
  Another advantage of going via marker interface way is that at any point of time you can easily cast the objects of the implementing classes. Again it's not that if you go via public final approach, you can't do that. You can very well do, but casting might look a cleaner approach in many situations.  
    
  The bottom-line is there will hardly be any enforced need for a designer/developer to go via that way as there can be possible alternatives, but marker interfaces can surely be a preferred choice for some in some cases.  
    
  **Note:** Annotations are considered as another possible (quite popular as well) alternative to marker interfaces. Read more about them in this article - [**Annotations in Java >>**](http://geekexplains.blogspot.com/2008/06/what-are-annotations-in-java-what-are.html)