# https://dzone.com/articles/composition-vs-inheritance

**Inheritance**

Inheritance is a reusability mechanism in object-oriented programming. With inheritance, the common properties of various objects are exploited to form relationships with each other. The abstract and common properties are provided in the superclass , which is available to the more specialized subclasses.

For example, a color printer and a black-and-white printer are kinds of a printer (single inheritance); an all-in-one printer is a printer, scanner, and photocopier (multiple inheritance).

**Object composition**

Individual abstractions offer certain functionalities that need to be combined with other objects to represent a bigger abstraction: a composite object that is made up of other smaller objects. You need to make such composite objects to solve real-life programming problems. In such cases, the composite object shares HAS-A relationships with the containing objects, and the underlying concept is referred to as object composition.

By way of analogy, a computer is a composite object containing other objects such as CPU, memory, and a hard disk. In other words, the computer object shares a HAS-A relationship with other objects.

**Composition vs. Inheritance**

In some situations, it’s difficult to choose between the two. It’s important to remember  
that nothing is a silver bullet—you cannot solve all problems with one construct. You need to analyze each situation carefully and decide which construct is best suited for it.

    A rule of thumb is to use HAS-A and IS-A phrases for composition and inheritance, respectively. For instance,

A computer HAS-A CPU.  
A circle IS-A shape.  
A circle HAS-A point.  
A laptop IS-A computer.  
A vector IS-A list.

    This rule can be useful for identifying wrong relationships. For instance, the relationship of car IS-A tire is completely wrong, which means you cannot have an inheritance relationship between the classes Car and Tire . However, the car HAS-A tire (meaning car has one or more tires) relationship is correct—you can compose a Car object containing Tire objects.

    In real scenarios, the relationship distinctions can be nontrivial. Many people ignore a big caution sign suspended over this practice—always check whether the IS-A relationship exists between the derived classes and the base class. If the IS-A relationship does not hold, it’s better to use composition instead of inheritance.

    For example, take a set of classes DynamicDataSet and SnapShotDataSet that require a common  
functionality—say, sorting. Now, one could derive these data set classes from a sorting implementation, as

//Sorting.java

import java.awt.List;

public class Sorting {

public List sort(List list) {

// sort implementation

return list;

}

}

class DynamicDataSet extends Sorting {

// DynamicDataSet implementation

}

class SnapshotDataSet extends Sorting {

// SnapshotDataSet implementation

}

Do you think this is a good solution? No, it’s not a good solution for the following reasons:

* The rule of thumb does not hold here. DynamicDataSet is not a Sorting type. If you make such mistakes in class design, it can be very costly—and you might not be able to fix them later if a lot of code has accumulated that makes the wrong use of inheritance relationships. For example, Stack extends Vector in the Java library. Yet a stack clearly is not a vector, so it could not only create comprehension problems but also lead to bugs. When you create an object of Stack class provided by the Java library, you can add or delete items from anywhere in the container because the base class is Vector , which allows you to delete from anywhere in the vector.
* There is another challenging issue: what if one DataSet class wants to use one sorting algorithm (say, MergeSort) and another data set class wants to use a different sorting algorithm (say, QuickSort)? Will you inherit from two classes implementing two different sorting algorithms?First, you cannot directly inherit from multiple classes, since Java does not support multiple class inheritance. Second, even if you were able to somehow inherit from two different sorting classes (MergeSort extends QuickSort , QuickSort extends DataSet), that would be an even worse design.

    In this case it is best to use composition—in other words, use a HAS-A relationship instead of an IS-A relationship. The resultant code is

//Sorting.java

import java.awt.List;

interface Sorting {

List sort(List list);

}

class MergeSort implements Sorting {

public List sort(List list) {

// sort implementation

return list;

}

}

class QuickSort implements Sorting {

public List sort(List list) {

// sort implementation

return list;

}

}

class DynamicDataSet {

Sorting sorting;

public DynamicDataSet() {

sorting = new MergeSort();

}

// DynamicDataSet implementation

}

class SnapshotDataSet {

Sorting sorting;

public SnapshotDataSet() {

sorting = new QuickSort();

}

// SnapshotDataSet implementation

}

Use inheritance when a subclass specifies a base class, so that you can exploit dynamic polymorphism.In other cases, use composition to get code that is easy to change and loosely coupled. In summary, **favor composition over inheritance.**