Composition over Inheritance

The reasons to prefer composition are profound.

**Inheritance**

* Use inheritance when you have a clear hierarchical relationship between classes, and you can say one class is a specialized form of another.
* **Inheritance is sometimes useful when hierarchy is more important than the formation of objects. It relates to Open-Closed Principle, which states that classes should be closed for modification but open to extension. That way you can have polymorphism**
* It’s suitable when you want to reuse code from the base class and when changes to the base class should propagate to derived classes. Inheritance is a good choice when you need to override or extend the behavior of the base class.
* There are two benefits of inheritance: **subtyping** and **subclassing**. Subtyping means conforming to a type (interface) signature, i.e. a set of APIs, and one can override part of the signature to achieve subtyping polymorphism. **Subclassing means implicit reuse of method implementations.**
* As a rule of thumb, I tend to choose inheritance over composition when polymorphic use is expected. For example, having a polymorphic class Widget in GUI frameworks, or a polymorphic class Node in XML libraries allows to have an API which is much more readable and intuitive to use
* You can't change the implementation inherited from super classes at runtime (obviously because inheritance is defined at compile time).

**Why Inheritance Breaks Encapsulation**

**Inheritance exposes protected members**. **This breaks encapsulation of the parent class**, and if used by subclass.

**Composition**

* If your object needs to appear as a different object or behave differently depending on an object state or conditions, then use Composition: Refer to State and Strategy Design Patterns. One example of this: **You want to create a Stack out of a List. Stack only has pop, push and peek. You shouldn't use inheritance given that you don't want push\_back, push\_front, removeAt, et al.-kind of functionality in a Stack**.
* **Use composition to combine simple objects into more complex ones**. It’s ideal when you want to model a has-a relationship rather than an is-a relationship.
* **Composition is preferable when you want to change the parts of a whole dynamically at runtime**, or when you want to limit the visibility of the composed objects’ methods and data.
* **Composition avoids a proliferation of classes.**
* **Delegation is one example of a way to use composition instead of inheritance. Delegation lets you modify the behavior of a class without subclassing**. It can be achieved using Decorator pattern. Composition provides pluggability.
* **Composition is a very simple and tactical way of building objects**. Using composition you can always choose to define your own behavior or simply expose that part of your composed parts

When you want to "Copy"/Expose the base class' API, you use inheritance. When you only want to "copy" functionality, use delegation.

**Inheritance creates tight, compile-time coupling between the classes whereas Composition in contrast is loose coupling, which among others enables clear separation of concerns, the possibility of switching dependencies at runtime and easier, more isolated dependency testability.**