Inheritance

- Is great for showing a class that is always a special variant of its parent class.
- Explicitly captures commonality, taking a class definition (what's the same: attributes, method signatures and methods) and extending it with a new class definition (what's different: attributes, method signatures and methods)

Inheritance Rules

- Inheritance should be used only for genuine “is a kind of” relationships
  - It should always be possible to substitute a subclass object for a superclass object
  - All methods in the superclass should make sense in the subclass
  - Is not a “role played by a”
- Inheritance for short-term convenience may lead to problems in the future
- Subtypes should be mutually exclusive
- Be careful of switching back and forth between types – lose relationship history: An object, once classified, will forever remain an object in that class.

Risks of Inheritance

- Weak Encapsulation Within
  - Strong encapsulation with other classes, but weak encapsulation between superclass and its subclasses
  - Subclasses are not well shielded from the potential ripple effects of changes to superclasses

- The “Fragile Base Class” problem:
  - A fragile base class is a common problem with inheritance, which applies to Java and any other language which supports inheritance.
  - In a nutshell, the base class is the class you are inheriting from, and it is often called fragile because changes to this class can have unexpected results in the classes that inherit from it.
  - There are few methods of mitigating this; but no straightforward method to entirely avoid it while still using inheritance.
  - Therefore, if you change a superclass (base class) , you must check all subclasses to correct any rippling change effects.

How to mitigate Risks of Inheritance

- Make sure the subclass is indeed a special kind-of the superclass and not just:
  - A factoring out of common method implementations
  - A role played by the superclass : Person -> Employee & Customer
    - *Composition* is more flexible
- If you find you need to nullify or override (not extend with `super`) the inherited responsibilities:
  - Introduce a new superclass (if you can)
  - Define the class elsewhere and use composition to invoke the needed responsibilities.

Composition

- Composition extends the responsibilities of an object by *delegating* work to additional objects.
- Composition is *the* major mechanism for extending the responsibilities of an object
- Nearly every object in an object model is composed of, knows of, or works with other objects (composition).

  - Peter Coad, *Java Design*

- **Note**: Containment is special case of *strong composition*, where the inner, contain object is only accessed through its container: e.g. Order and Order line item, vs. Customer and Order.

Alternative to Inheritance: Composition

- Composition isn't preferred because of some unseen benefit of composition, it's preferred because it *avoids the drawbacks of inheritance*.
- Inheritance adds the significant benefit of substitutability.
- However, that benefit comes with the significant cost of *tight coupling*.
- Inheritance is the *strongest coupling relationship* possible.
- If you don't need the substitutability, the coupling cost isn't worth it.

Composition over inheritance

- Composition over inheritance (or composite reuse principle) in object-oriented programming is the principle that classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement the desired functionality) rather than inheritance from a base or parent class.
- This is an often-stated principle of OOP, such as in the influential book *Design Patterns*.
- An implementation of composition over inheritance typically begins with the creation of various interfaces representing the behaviors that the system must exhibit.
- The use of interfaces allows this technique to support the Polymorphic behavior that is so valuable in object-oriented programming.
- Classes implementing the identified interfaces are built and added to business domain classes as needed. Thus, system behaviors are realized without inheritance.
- In fact, business domain classes may all be base classes without any inheritance at all. Alternative implementation of system behaviors is accomplished by providing another class that implements the desired behavior interface.
- Any business domain class that contains a reference to the interface can easily support any implementation of that interface and the choice can even be delayed until run time.

- Wikipedia

Benefits and Drawbacks

- To favor composition over inheritance is a design principle that gives the design *higher flexibility*.
- It is more natural to build business-domain classes out of various components than trying to find commonality between them and creating a family tree.
- For example, a gas pedal and a wheel share very few common traits, yet are both vital components in a car.
- What they can do and how they can be used to benefit the car is easily defined.
- Composition also provides a more stable business domain in the long term as it is less prone to the quirks of the family members.
- In other words, it is better to compose what an object can do (*HAS-A*) than extend what it is (*IS-A*).
- Initial design is simplified by identifying system object behaviors in separate interfaces instead of creating a hierarchical relationship to distribute behaviors among business-domain classes via inheritance.

Empirical studies

A 2013 study of 93 open source Java programs (of varying size) found that:

- While there is [no] huge opportunity to replace inheritance with composition (...), the opportunity is significant (median of 2% of uses [of inheritance] are only internal reuse, and a further 22% are only external or internal reuse).
- Our results suggest there is no need for concern regarding abuse of inheritance (at least in open-source Java software), but they do highlight the question regarding use of composition versus inheritance.
- ***If there are significant costs associated with using inheritance when composition could be used, then our results suggest there is some cause for concern.***

— *Tempero et al., "What programmers do with inheritance in Java"*