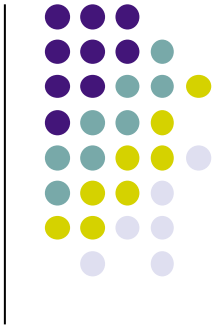
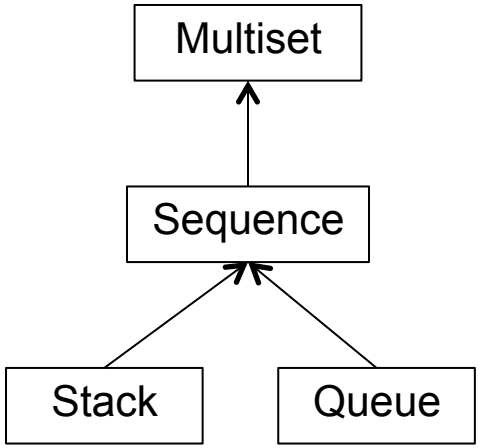
Similar data abstractions

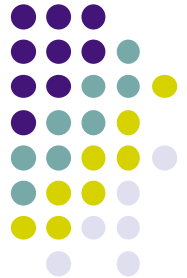


- Data abstractions are often very similar
  - Especially if the entities they represent are similar (such as “person” versus “employee”, “car part” versus “airplane part”, and so forth)
- A simple example is the concept “collection of elements”
  - Multiset: a collection with no defined order
  - Sequence: a multiset with a total order
    - Sequence = multiset + total order
  - Stack: a sequence where adding and removing are done on the same side
    - Stack = sequence + add/remove constraint
  - Queue: a sequence where adding is done on one side and removing on the other side
    - Queue = sequence + add/remove constraint



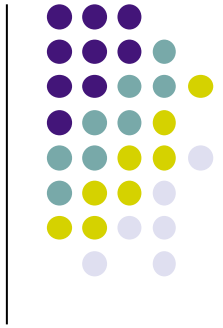
- Language support for similar data abstractions is important

Incremental definition with inheritance



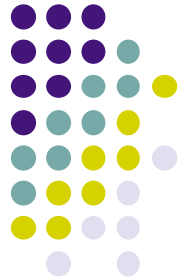
- It is important to avoid duplicated code in a program
  - Duplicated code is problematic at two levels
    - Different copies tend to diverge slightly with time (low-level bugs)
    - The same idea is expressed twice (high-level bugs)
  - It is much better, for program structure and maintenance, to express the same idea exactly once
- Inheritance achieves this for similar data abstractions
  - Definition A can “inherit” from definition B
  - This means that A uses B as a base, possibly with modifications and extensions
- The incremental definition A is also called a class
  - A class can either be a complete or incremental definition
  - The resulting definition (A + the classes it inherits from directly or indirectly) is always complete

Dangers of inheritance



- Inheritance can be very useful, but its use is fraught with dangers
- The ability to extend A with inheritance is another interface to A
  - An additional interface to A's usual interface
  - This interface is extremely difficult to make correct and maintain correct throughout the lifetime of the abstraction
- So we must be very careful when using inheritance – two general rules:
  1. Prefer composition over inheritance
  2. When using inheritance, always follow the substitution principle

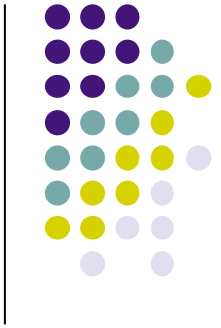
(1) Prefer composition over inheritance

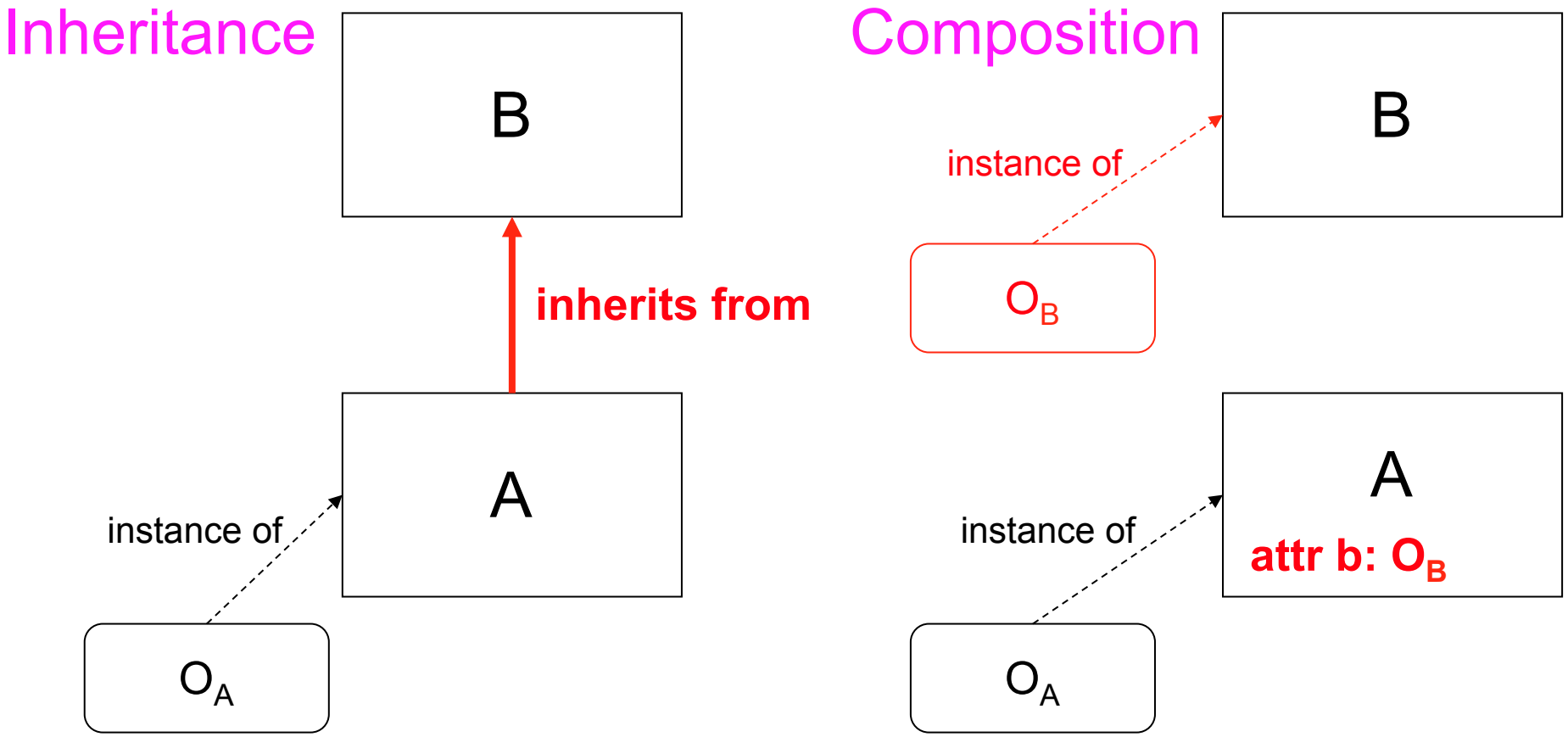


- It is important to use inheritance as little as possible
  - Only use it in well-defined ways, for example in well-established “programming patterns”
  - When defining a class, it should be declared “final” (not extensible by inheritance) by default

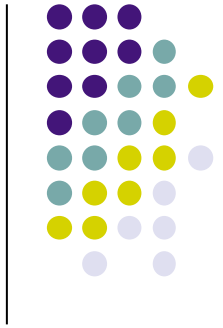
- Composition is much easier to use than inheritance and is often sufficient
  - Composition = an object refers to another object in one of its attributes (such as attribute figlist in CompoundFigure)
  - Composition does not add another interface: the object referred to is always accessed through its usual interface

Inheritance versus composition





(2) Always follow the substitution principle



- The use of inheritance is much easier if the substitution principle is followed
- Suppose that A inherits from B with objects O_A et O_B
  - Substitution principle: Every procedure that accepts O_B must accept O_A
  - If this principle is followed, then inheritance does not break anything! We say that A is a conservative extension of B.
- This is also called LSP (Liskov Substitution Principle)