



# Principles of Object-Orientation II

- Inheritance
- Polymorphism
- Composition



## 5) Inheritance

### What is Inheritance?

Inheritance is the concept that when a class of objects is defined, any subclass that is defined can inherit the definitions of one or more general classes. This means for the programmer that an object in a subclass need not carry its own definition of data and methods that are generic to the class (or classes) of which it is a part. This not only speeds up program development; it also ensures an inherent validity to the defined subclass object (what works and is consistent about the class will also work for the subclass).

- *Tech Target Network*

In object technology, the ability of one class of objects to inherit properties from a higher class.

- *Computing Dictionary*

Inheritance is the incremental construction of a new definition in terms of existing definitions without disturbing the original definitions and their clients.

- *Dictionary of Object Technology*

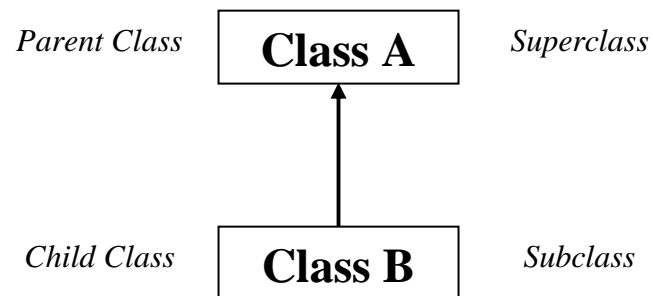


## Inheritance (cont.)

### Inheritance

Inheritance is an object-oriented principle that allows you to take the implementation of any given class and build a new class based on that implementation. This new subclass starts by inheriting all of the data and operations defined in the superclass. It can then extend the behavior by adding additional data and new methods to operate on it.

The subclass can also extend or replace behavior in the superclass by “overriding” methods that were already implemented in the superclass. The inheritance hierarchy can be many levels deep. At the root of the entire class hierarchy is the Object class, from which all classes eventually derive in Java.



# Principles of Object Orientation



## Inheritance (cont.)

### Inheritance Terminology

The terms "subclass" and "superclass" are the broadly accepted in object orientation. A superclass is sometimes referred to as a "base" class, and a subclass a "derived" class.

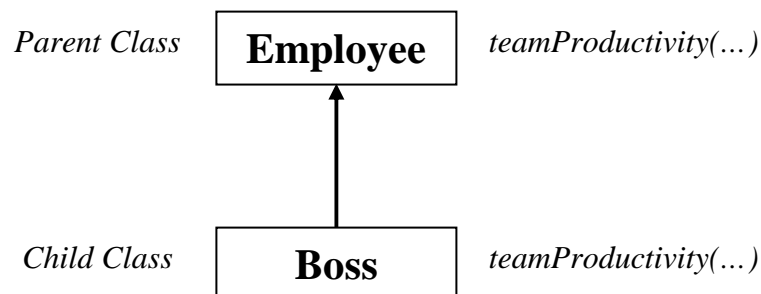
To create a subclass is called “specialization”. To factor out common parts of derived classes into a common base class (or parent) is called “generalization”. “Overriding” is the term for redefining a parent class method in a derived class, thus providing specialized behavior.

### Inheritance Example

Assume a class “Employee” exists as a superclass, and a class “Boss” that exists as a subclass class.

The Boss (derived) class is a more specialized version of the Employee (base) class. The Boss class adds an attribute of “staff” (i.e., a list of other employees). Note that the Boss class represents an “is a” relationship with Employee. A Boss “is a” type of Employee, so it has all the same properties as Employee - plus a few more.

The Boss class can override the Employee class’ “teamProductivity(...)” method to incorporate the work that is done by its “staff” and adds a few new methods of its own dealing behaviors particular to bosses.



# Principles of Object Orientation



## Inheritance (cont.)

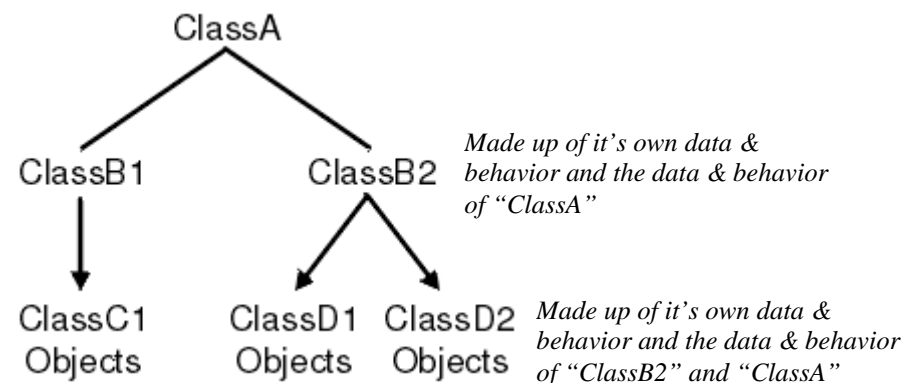
### Inheritance Details

Some object classes can be categorized and divided into more specific classes, just as a generic category can be divided into more specific types.

As we saw previously, the more generic class is commonly known as the superclass of the more specific classes. Likewise, each of the more specific classes is commonly known as a subclass of the more generic class. A given subclass, in turn, can be divided into additional, more specific subclasses.

Each subclass definition is built on the definition of its superclasses, and can also specify a set of additional attributes and methods unique to the subclass. An object, which is an instance of a subclass, not only contains or inherits the attributes and methods defined for the subclass, but also inherits the attributes and methods defined for each of the more generic classes (superclasses) as well.

You can best illustrate how different objects inherit attributes from different classes by drawing an inheritance tree, which is a diagram that shows how different classes and their objects are related. Consider the following example of an inheritance tree:

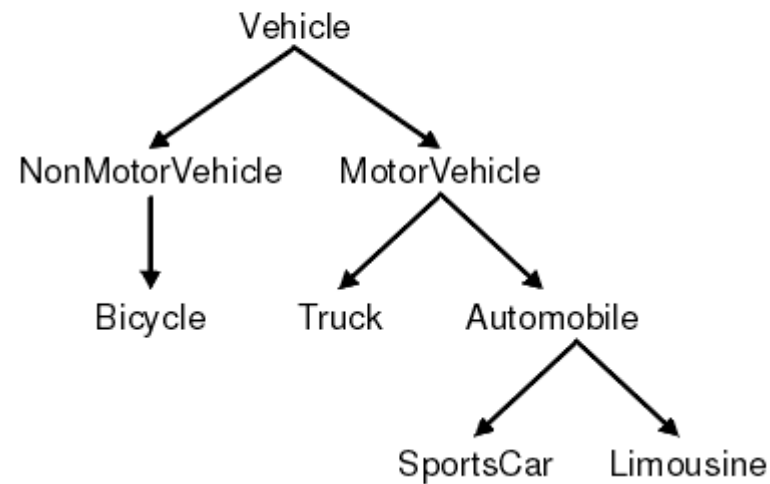




## Inheritance (cont.)

### Inheritance Details

You can best understand how class inheritance works by considering a concrete example. Suppose that you were designing an object-oriented system to process information about vehicles. An object-oriented design of such a database might be shown as follows:





## Inheritance (cont.)

### Inheritance Details

The Vehicle class from our previous example inheritance hierarchy is divided into two more specific subclasses: the NonMotorVehicle and the MotorVehicle classes. Notice that the NonMotorVehicle class only contains one subclass: the Bicycle class. On the other hand, the MotorVehicle class contains two subclasses: the Truck class and the Automobile class. The Automobile class, in turn, contains several subclasses of its own; the only two shown are the SportsCar and the Limousine classes.

Assume that all of the classes shown in the diagram are “abstract” except for the Bicycle class, the Truck class, and the subclasses of Automobile – Limousine and SportsCar. This means that there are no objects in our hierarchy that are any more specific. The only objects that we can instantiate are of the “concrete” classes - Bicycle, Truck, Limousine and SportsCar. The abstract classes Vehicle, NonMotorvehicle, MotorVehicle, and Automobile can never be instantiated.

There are no Vehicle objects in our system that are not additionally classified as a specific type of Vehicle. Furthermore, there is no Automobile object that is not a specific type of Automobile: for example, a specific instance of the Automobile class must either be a SportsCar, a Limousine, or an instance of some other Automobile subclasses not shown.



## Inheritance (cont.)

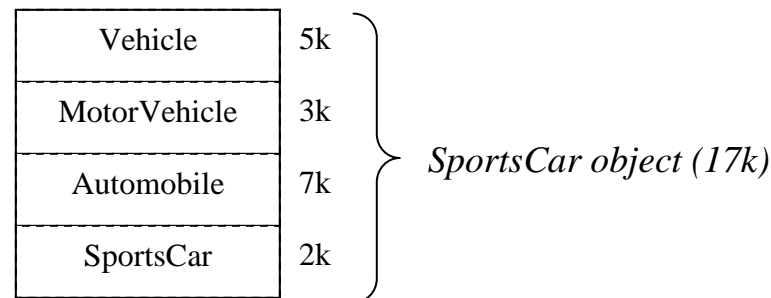
### Inheritance Details

Using the same inheritance hierarchy, assume that we have objects that represent specific vehicles, including “Frank's” sports car and “Jackie's” limousine.

Frank's sports car is an instance of the SportsCar class. It is a SportsCar. We can also say that Frank's sports car is an Automobile, which it is a MotorVehicle, and that it is a Vehicle. After all, any sports car can be classified as an automobile, as well as a motor-powered vehicle, or as a generic vehicle.

Similarly, Jackie's limousine is a Limousine, an Automobile, a MotorVehicle, and a Vehicle.

In inheritance, all data and behavior are located in the same namespace. No special access coding is required – all data and behaviors that make up the class itself, and all parent classes can be considered “all in the same class”.







## Inheritance (cont.)

### Method Overriding

Method Overriding involved defining a method whose signature exactly matches (i.e., same name, same argument types, and same return type) a method defined in a superclass. When an overridden method is invoked, the JVM (in Java) uses "dynamic method lookup" to determine which method definition is applicable to the current object.

A derived class can use the methods of its base class, or it can override them. The method in the derived class must have the same signature and return type as the base class method to override. The signature is number and type of arguments to the method.

When an object of the base class is used, the base class method is called. When an object of the subclass is used, its version of the method is used if the method is overridden.

Note that overriding is different from overloading. With overloading, many methods of the same name with different signatures (different number and/or types of arguments) are created. With overriding, the method in the subclass has the identical signature to the method in the base class. With overriding, a subclass implements its own version of a base class method. The subclass can selectively use some base class methods as they are, and override others.

# Principles of Object Orientation



## Inheritance (cont.)

### "super"

To access the parent version of a method from a child method that overrides it, we use “super” as the receiver when sending a message (think of “super” as a reference attribute that allows an object to refer to itself as it’s parent type).

The implicit reference attribute “super” is used within child class methods to invoke the parent class version of the method being defined.

For example, a subclass overrides the “update(...)” method, but makes a call to “super.update(...)” within that method definition to include the parent’s processing along with the subclass’s extensions.

```
public void update(int xyz)
{
    update(xyz);
    [...]
}
```

*This version results in an infinite recursive loop wherein “update(...)” calls “update(...)” which again calls “update(...)”, etc.*

```
public void update(int xyz)
{
    super.update(xyz);
    [...]
}
```

*This version results in call to the parent class version of “update(...)” method. When that call returns, the remaining code of the child-class “update(...)” method.*

Using “super” always starts method lookup from the *parent* class and stops at the first one found.

You can actually use super to call the parent version of any method, not just the one you are currently defining. For example, the subclass’s “update(...)” method could also make a call to “super.evaluate(...)” if desired.



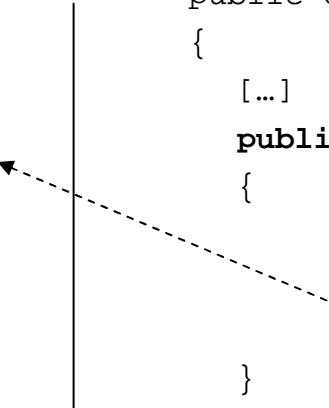
## Inheritance (cont.)

### Inheritance and Constructors

Constructors are considered tightly coupled to the class they are defined in, and thus are not directly inherited and therefore need to be specifically invoked. For example, assume the following:

```
public class Employee
{
    [...]
    public Employee(int a, double b)
    {
        [...]
    }
}

public class Boss extends Employee
{
    [...]
    public Boss(int a, double b, String c)
    {
        // Invoke the parent constructor
        super(a, b); // This MUST be called
        [...]
    }
}
```



If a given child class has a parent class defined with a constructor that requires parameters, that parent constructor *must* be called from within the child class constructor, and must be provided with parameters. This is compiler-enforced.

Since the “Employee” class above defines a constructor with 2 parameters, any “Boss” child class must invoke the Employee constructor from within its own constructor.



## Inheritance (cont.)

### Inheritance and Constructors

If we define no constructors in a class, then the compiler supplies the default constructor that takes no arguments.

This results in an exception to the previous parent constructor invocation rule:

If a parent class defines no constructor, or it uses the default constructor (the one with no parameters), then that constructor need *not* be manually invoked because the compiler will insert that parent constructor invocation automatically.

(Note that manually invoking a parent's default constructor will not harm anything in these cases).



## 6) Polymorphism

### What is Polymorphism?

Polymorphism is the ability of a generalized request (message) to produce different results based on the object that it is sent to.

- *Computing Dictionary*

Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure. Ad-hoc polymorphism is obtained when a function works, or appears to work, on several different types (which may not exhibit a common structure) and may behave in unrelated ways for each type.

- *Strachey*

A concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass; thus, any object denoted by this name is able to respond to some common set of operations in different ways.

- *Booch*

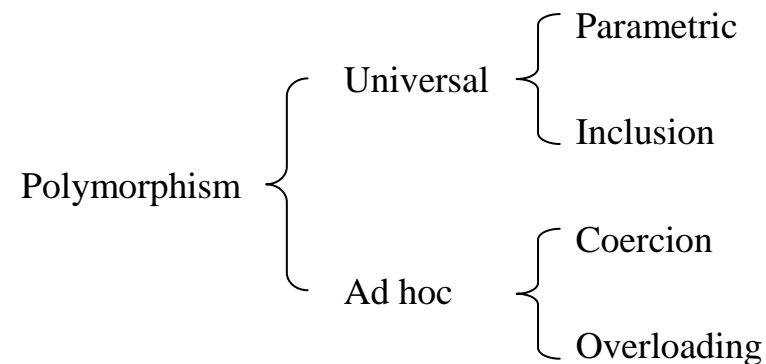


## Polymorphism

### Polymorphism

Polymorphism is the idea of allowing the same code to be used with different classes of data, resulting in more general and abstract implementations. The concept of polymorphism applies to functions as well as types. A function that can evaluate to and be applied to values of different types is known as a polymorphic function. A data attribute that contains elements of an unspecified type is known as a polymorphic data attribute.

Polymorphism can be divided into two major categories -- ad hoc and universal -- and four varieties: coercion, overloading, parametric, and inclusion. The classification structure is:



*Universal* polymorphism refers to a uniformity of type structure, in which the polymorphism acts over an infinite number of types that have a common feature. The less structured *ad hoc* polymorphism acts over a finite number of possibly unrelated types. The four varieties may be described as:

- Coercion: a single abstraction serves several types through implicit type conversion
- Overloading: a single identifier denotes several abstractions
- Parametric: an abstraction operates uniformly across different types
- Inclusion: an abstraction operates through an inclusion relation



## Polymorphism

### Ad hoc - Coercion

Coercion represents implicit parameter type conversion to the type expected by a method or an operator, thereby avoiding type errors.

For the following expressions, the compiler must determine whether an appropriate binary + operator exists for the types of operands:

```
2.0 + 2.0  
2.0 + 2  
2.0 + "2"
```

The first expression adds two double operands; the Java language specifically defines such an operator.

The second expression adds a "double" and an "int"; Java does not define an operator that accepts those operand types. Fortunately, the compiler implicitly converts the second operand to double and uses the operator defined for two double operands. That is tremendously convenient for the developer; without the implicit conversion, a compile-time error would result or the programmer would have to explicitly cast the "int" to "double".

The third expression adds a double and a String. Once again, the Java language does not define such an operator. So the compiler coerces the double operand to a String, and the plus operator performs string concatenation.



## Polymorphism

### Ad hoc - Coercion

Coercion also occurs at method invocation. Suppose class “Derived” extends class “Base”, and class “C” has a method with signature “doIt(Base b)”. For the method invocation in the code below, the compiler implicitly converts the derived reference variable, which has type Derived, to the Base type prescribed by the method signature. That implicit conversion allows the m(Base) method's implementation code to use only the type operations defined by Base:

```
C myC = new C();  
Derived derived = new Derived();  
myC.doit(derived);
```

Again, implicit coercion during method invocation obviates a cumbersome type cast or an unnecessary compile-time error. Of course, the compiler still verifies that all type conversions conform to the defined type hierarchy.





## Polymorphism

### Ad hoc - Overloading

Overloading permits the use of the same method name to denote multiple, distinct program meanings.

A class may possess multiple methods with the same name, provided that the method signatures are distinct. That means either the number of parameters must differ or at least one parameter position must have a different type.

Unique signatures allow the compiler to distinguish between methods that have the same name. The compiler distinguishes between the method names using the unique signatures, effectively creating unique names.

In light of that, any apparent polymorphic behavior related to method overloading actually evaporates upon closer inspection.

Both coercion and overloading are classified as ad hoc because each provides polymorphic behavior only in a limited sense. Though they fall under a broad definition of polymorphism, these varieties are primarily developer conveniences.

Coercion obviates cumbersome explicit type casts or unnecessary compiler type errors.

Overloading, on the other hand, provides syntactic sugar, allowing a developer to use the same name for distinct methods.



## Polymorphism

### Universal - Parametric

Parametric polymorphism allows the use of a single abstraction across many types.

For example, a List abstraction, representing a list of homogeneous objects, could be provided as a generic module. You would reuse the abstraction by specifying the types of objects contained in the list. Since the parameterized type can be any user-defined data type, there are a potentially infinite number of uses for the generic abstraction, making this arguably the most powerful type of polymorphism.

At first glance, the above List abstraction may seem to be the utility of the class `java.util.ArrayList`. However, pre-5.0 Java (also known as Java 1.5) does not support *true* parametric polymorphism in a type-safe manner. The `java.util.ArrayList` and `java.util`'s other collection classes are written in terms of the primordial Java class, `java.lang.Object`. Java's single-rooted implementation inheritance offers a partial solution, but not the true power of parametric polymorphism.



## Polymorphism

### Universal - Parametric

Using Java 5.0 “generics”, a collection is no longer treated as a list of Object references, but you can differentiate between a collection of references to Vehicles and collection of references to Integers, etc. A collection with a generic type has a type parameter that specifies the element type to be stored in the collection

As an example, consider the following segment of code that creates a linked list and adds an element to the list using pre-5.0 Java:

```
LinkedList list = new LinkedList();  
list.add(new Integer(1));  
Integer num = (Integer) list.get(0);
```

As you can see, when an element is extracted from the list, it must be cast. The casting is safe as it will be checked at runtime, but if you cast to a type that is different from, and not a supertype of, the extracted type then a runtime `ClassCastException` will be thrown.



## Polymorphism

### Universal - Parametric

Using Java 5.0 generic types, the previous segment of code can be written as follows:

```
LinkedList<Integer> list = new LinkedList<Integer>();  
list.add(new Integer(1));  
Integer num = list.get(0);
```

As you can see, you no longer need to cast to an Integer since the “get(...)” method would return a reference to an object of a specific type (Integer in this case). If you were to assign an extracted element to a different type, the error would be at compile-time instead of run-time. This early static checking increases the type safety of the Java language.

To reduce the clutter, the above example can be rewritten as follows...using autoboxing – the automatic conversion of data of a primitive type to the corresponding wrapper type, for example from "int" to Integer:

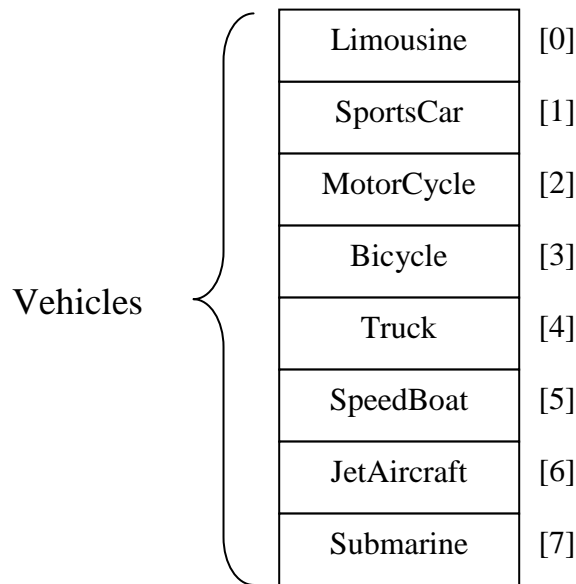
```
LinkedList<Integer> list = new LinkedList<Integer>();  
list.add(1); // Autoboxed  
int num = list.get(0); // Unautoboxed
```



## Polymorphism

### Universal - Parametric

With Java generics, we can create a list of objects that on the surface appear heterogeneous, but in reality all share a common ancestor class. As such, we can treat each of the objects in the list generically, as Vehicles – and can use them to invoke Vehicle methods without regard for the actual object type.



```
ArrayList<Vehicle> myVehicles = new ArrayList<Vehicle>();

// Here, we'll skip the code that would add
// elements to the ArrayList
[...]

// Now, we know any element in this list must be a
// Vehicle so we can generically call "move(...)" on
// element in the list without actually knowing which
// actual class type the element is.
// Polymorphism & dynamic binding in Java insure that
// the appropriate "move(...)" method is executed.

for (int i = 0; i < myVehicles.size(); i++)
    myVehicles.get(i).move(x,y);

// Each call will potentially result in a different "move"
// method invocation. 1st = Limousine's "move", 2nd = SportsCar's
// "move", 3rd = MotorCycle's "move", 4th = Bicycle's "move",
// 5th = Truck's "move", etc.
```



## Polymorphism

### Universal - Inclusion

Inclusion polymorphism achieves polymorphic behavior through an inclusion relation between types or sets of values. For many object-oriented languages, including Java, the inclusion relation is a subtype relation. So in Java, inclusion polymorphism is subtype polymorphism.

Inclusion polymorphism means that a derived class object can appear wherever a base class object is expected.

As noted earlier, when Java developers generically refer to polymorphism, they invariably mean subtype polymorphism. Gaining a solid appreciation of subtype polymorphism's power requires viewing the mechanisms yielding polymorphic behavior from a type-oriented perspective. The rest of this article examines that perspective closely. For brevity and clarity, I use the term polymorphism to mean subtype polymorphism.

Inclusion/subtype polymorphism is a natural part of Java. The underlying dynamic method-binding mechanism that initiates this form of polymorphism kicks into action whenever you call an instance method, regardless of whether an object reference variable has class or interface type. Nevertheless, to achieve inclusion/subtype polymorphism's full benefits, you must work with class hierarchies, where subclasses introduce methods that override superclass methods.

# Principles of Object Orientation



## Polymorphism

### More on Polymorphism

```
public class A
{
    [...]
    public void doIt()
    {
        System.out.println("This is the A doIt");
    }
    [...]
}

public class B extends A
{
    [...]
    public void doIt()
    {
        System.out.println("This is the B doIt");
    }
    [...]
}
```

Remember that a superclass reference can be used to refer to any subclass objects. This is because subclass objects are actually superclass objects as well. In the above example, "B" objects are also "A" objects.

When a method is invoked via a superclass reference, dynamic method lookup begins with the actual object's class type, not at the reference variable's class type.

```
public class Driver
{
    public static void main(String args[])
    {
        A myA = null;

        myA.doIt(); // Causes NullPointerException!!

        // The following calls the "A" class "doIt()" method. The "A"
        // reference refers to an "A" object – therefore method lookup
        // starts with the "A" class. A suitable "doIt()" method is
        // found in the "A" class and is therefore executed.

        myA = new A();
        myA.doIt(); // Calls the "A" doIt() method

        // The following calls the "B" class "doIt()" method. The "A"
        // reference refers to a "B" object which is legal because "B"
        // objects derive from "A" objects, and are therefore actually "A"
        // objects as well.
        // Method lookup starts with the "A" class. A suitable "doIt()"
        // method is found in the "A" class and is therefore executed.

        myA = new B();
        myA.doIt(); // Calls the "B" doIt() method
    }
}
```



## 7) Composition

### What is Composition?

Composition is a way and practice to combine simple objects or data types into more complex ones. Compositions are a critical building block of many basic data structures, including the tagged union, the linked list, the binary tree, and the object used in object-oriented programming. Sometimes an issue of ownership arises: when a composition is destroyed, should objects belonging to it be destroyed as well? If not, the case is sometimes called aggregation.

- *Wikipedia*

Composition is an organized collection of components interacting to achieve a coherent, common behavior.

- *Kafura*

A technique for building a new object from one or more existing objects that support some or all of the new object's required interfaces.

- *Free Online Dictionary of Computing*





## Composition

### Composition

Composition can be defined as an organized collection of components (i.e., objects) interacting to achieve a coherent, common behavior.

The "part-whole" relationship is often expressed as a "has-a" relationship. For example, in referring to the relationship between an automobile and the automobile's windshield, the statement "the automobile has a windshield" suggests why the term "has-a" is used to describe part-whole compositions.

There are two forms of composition named *association* (also acquaintance) and *aggregation* (also containment). These two forms of composition are similar in that they are both part-whole constructions. What distinguishes aggregation from association is the visibility of the parts. In an aggregation only the whole is visible and accessible. In association the interacting parts are externally visible and, in fact, may be shared between different compositions.

- A soda machine is an example of aggregation. The machine is a whole composed of several internal parts (a cooling system, a coin acceptor, a change maker, a soda supply). These internal parts are not visible or accessible to the normal user of the soda machine.
- A computer workstation is an example of composition using association. The workstation consists of a keyboard, a mouse, a monitor, a printer, a modem and a processor. Each of these interacting parts are visible to the user and directly manipulate-able by the user.



## Composition

### Composition

In some cases the more generic term "composition" will be used in favor of the more precise terms "association" or "aggregation". This occurs when the statement applies to both forms, when the difference between the two forms is much less important than the general idea of forming a whole from parts, or when the precise form of composition can be inferred from context.

Aggregation offers greater security because its structure is usually defined in advance and cannot be altered at run-time. The implementer of the aggregation is secure in the knowledge that the integrity of the aggregation and its proper functioning cannot be adversely affected by direct interference with its internal mechanisms.

Association offers greater flexibility because the relationships among the visible parts can be redefined at run-time. An association can, therefore, be made to adapt to changing conditions in its execution environment by replacing one or more of its components. Interesting design decisions can revolve around which form of composition to use, balancing a need for security against a need for greater flexibility at run-time.



## Composition

### Composition Using Association

Association is a parts-whole relationship in which the "whole" is defined by the "parts" and the relationships among the parts. The parts of the composition maintain their identity, their external visibility, and their autonomy in the composition. The parts are often viewed as peers, collaborators or acquaintances, such terms reflecting the primacy of the parts in the part-whole composition. In some sense, the whole is the sum of its parts.

Association is a composition of independently constructed and externally visible parts.

A computer workstation is a typical example of a real-world association.

Each of the parts externally visible and can be manipulated in its own right (keyboard, mouse, monitor, etc.). The notion of "computer workstation" refers to the particular assembly of these parts in a way that gives rise to the functionality expected of a computer workstation.

The expectations of a computer workstation would not be met by an assembly of fewer parts (i.e., no monitor), extraneous parts (i.e., two keyboards), or the correct parts associated differently (i.e., the mouse connected to the modem).



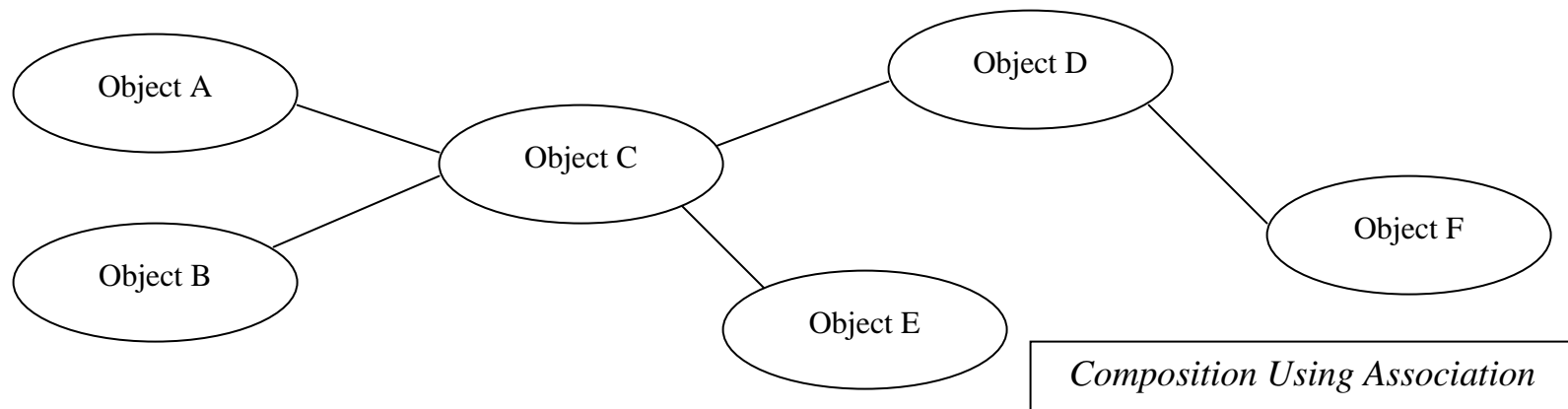
## Composition

### Composition Using Association

An association among objects is created when an object contains references or pointers to other objects.

One advantage of the association form of composition is that the parts may be shared between different compositions. This is easily accomplished by having the same object be connected to (pointed to) from two objects each of which is in a different composition. Using the computer workstation example mentioned above, it is possible to have a single printer shared between two different workstations.

A second advantage of an association is that the parts in an association can be dynamically changed. This can be accomplished simply by having a member in the composition connect to (point to) a different object. This change is dynamic in that it can be done at run-time. Again using the computer workstation example, it is possible to change the keyboard or mouse or to change the printer connected to the system.





## Composition

### Composition Using Aggregation

Aggregation is a composition in which the "whole" subsumes and conceals its constituent "parts". In contrast to an association, the parts are not visible externally, they do not have an identity as far as a user of the composition is concerned, and they do not possess autonomy to the same degree as parts of an association. The "whole" is the single visible entity.

Aggregation is a composition that encapsulates (hides) the parts of the composition. The outer objects contain inner (encapsulated) sub-objects which themselves may have hidden internal objects.

An aggregation of objects is created when one object (the "whole") contains in its encapsulated data one or more other objects (the "parts"). The aggregating class or object allows the entire assembly encapsulated sub-objects to be referred to as a single unit. This makes it easier to construct and manage multiple, independent instances of the system of sub-objects.

Through encapsulation, sub-objects of the system are protected from accidental misuse by elements outside the system itself. The existence of the encapsulating boundary captures the intent that the components of the system are designed to function as a unit.

### Indirect Control

When using aggregation, an important design issue is the degree of indirect control over the encapsulated objects that is reflected in the public interface of the aggregating class. Indirect control refers to the ability of an external object to affect the detailed organization or operation of the sub-objects through the public interface of the aggregating class. Ideally, no indirect control over the encapsulated objects should be allowed.



## Composition

### Composition Using Aggregation (cont.)

Providing excessive indirect control over the encapsulated objects begins to weaken the advantages of aggregation. In the extreme case, the aggregating class relinquishes complete control of its sub-objects, becoming little more than a weak wrapper to hold the sub-objects together as a group.

The designer of a good class must strike a balance between providing sufficient indirect control of the encapsulated objects so as to be usable in different applications and yet not so much indirect control as to lose the benefits of aggregation.

### Object Creation

The constructor of an aggregating class must insure that its sub-objects are properly initialized. It is expected, for example, that when a Vehicle object is constructed, all of its sub-objects (Engine, Radio, Transmission, etc.) are also properly constructed and ready for use.

The constructor for a sub-object may be related to the constructor of the aggregating class in one of three ways:

- Independent - the sub-object constructor is fixed and independent of the arguments of the aggregating class.
- Direct - the sub-object constructor depends directly on one or more arguments of the aggregating class's constructor.
- Indirect - the sub-object constructor depends on one or more values computed from the aggregating class's constructor arguments.

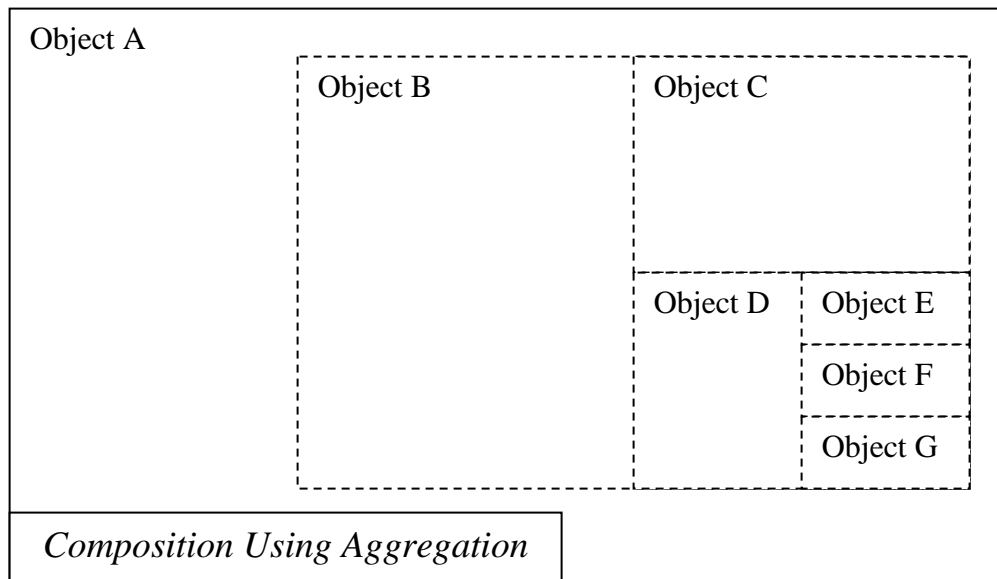


## Composition

### Composition Using Aggregation (cont.)

An advantage of aggregation is that the outer object may be used without much, if any, concern for the operation, or even the existence, of the internal sub-objects. When driving a car we are rarely concerned about the thousands of parts that are composed together to realize the car. The ability to ignore the finer structure of an object greatly simplifies the task of understanding how a system works or building a system that works in a particular way.

The second advantage of aggregation is that internal parts may be changed without affecting the user's view of the external whole. The internal structure of the parts may be completely changed or only individual parts may be replaced. Improvement in efficiency, reliability, or cost may motivate the replacement of parts.





# Principles of Object Orientation



## Summary:

- Inheritance is an object-oriented principle that allows you to take the implementation of any given class and build a new class based on that implementation. This new subclass starts by inheriting all of the data and operations defined in the superclass. It can then extend the behavior by adding additional data and new methods to operate on it.
- Polymorphism is the idea of allowing the same code to be used with different classes of data, resulting in more general and abstract implementations. The concept of polymorphism applies to functions as well as types. A function that can evaluate to and be applied to values of different types is known as a polymorphic function. A data attribute that contains elements of an unspecified type is known as a polymorphic data attribute.
- Composition can be defined as an organized collection of components (i.e., objects) interacting to achieve a coherent, common behavior.
  - Association is a parts-whole relationship in which the "whole" is defined by the "parts" and the relationships among the parts. The parts of the composition maintain their identity, their external visibility, and their autonomy in the composition. The parts are often viewed as peers, collaborators or acquaintances, such terms reflecting the primacy of the parts in the part-whole composition. In some sense, the whole is the sum of its parts.
  - Aggregation is a composition in which the "whole" subsumes and conceals its constituent "parts". In contrast to an association, the parts are not visible externally, they do not have an identity as far as a user of the composition is concerned, and they do not possess autonomy to the same degree as parts of an association.