# Overview:

* The composite design pattern composes objects into tree structures to represent part whole hierarchies.
  + Let clients treat individual objects and compositions of objects uniformly.
* A composite is an object designed as a composition of one-or-more similar objects that all exhibit similar functionality:
  + I.e. a group of object that is treated the same way as a single instance of the same type object.
* When we have many objects with common functionalities we create a composite object:
  + Creates a class that contains a group of its own objects.
* Allows us to build structures of objects in the form of trees:
  + Contains both compositions of objects and individual objects as nodes/leaves.
  + We ask each node in the tree structure to perform a task.
* Using a composite structure, we can apply the same operations over both composites and individual objects.
  + We can ignore the differences between compositions of objects and individual objects.
* Programmers often have to understand the difference between a leaf-node and a branch when dealing with tree-structured data:
  + Makes code more complex, and therefore, more error prone.
  + The composite pattern solves this by providing an interface that allows treating complex and primitive objects uniformly.
* The main purpose of the composite is to allow you to manipulate a single instance of the object just as you would manipulate a group of them:
  + The operations you can perform on all the composite objects often have a least common denominator relationship.

# Examples:

* Any organization that has many departments with each department having many employees to serve would utilize a composite pattern.
  + All are employees of the organization.
  + Groupings of employees could create a department.
    - Those departments ultimately can be grouped together to build the whole organization (hierarchy).
* Any tree structure in computer science follows the same concept as the composite.
* Graphics applications like drawing editors and schematic capture systems.
  + Let users build complex diagrams out of simple components.
  + User can group components to form larger components.
* Nested groups of menus and menu items in a GUI:
  + By putting menus and items in the same structure we create a part-whole hierarchy
    - A tree of objects that is made of parts (menus and menu items).
      * Can be treated as whole, like one big huge menu.
* We can use this pattern to treat this “huge” menu as “individual objects and compositions uniformly”.
* We have a tree structure of menus, submenus, and perhaps sub submenus along with menu items:
  + Any menu is a “composition” because it can contain both other menus and menu items.
  + Individual objects are just the menu items.
* Using a design that follows the Composite Pattern would allow you to write the same operation (like printing!) over the entire menu structure.

# When to use the Composite:

* When clients need to ignore the difference between compositions of objects and individual objects:
  + Using multiple objects in the same way with nearly identical code to handle them.
  + Less complex in this situation to treat primitives and composites as the same.
* When you are worried about memory usage:
  + Less number of objects reduces the memory usage.
  + Keep you away from errors related to memory like java.lang.OutOfMemoryError.
* When efficiency is a concern:
  + Creating an object in Java is really fast, however, we can still reduce the execution time of our program by sharing objects.
* When you are forced to maintain child ordering:
  + Parse trees as components:
    - We need to take special care to maintain that order.

# Benefits:

* Makes it easy to add new kinds of components.
* Makes clients simpler:
  + They do not have to know if they are dealing with a leaf or a composite component.
* One drawback is that it makes it harder to restrict the type of components of a composite:
  + You cannot rely on the type system to enforce constraints for you.
    - Have to use run-time checks instead.

# Implementations:

# Participants:

* **Component:**
  + Declares the interface for objects in the composition.
  + Implements default behavior for the interface common to all classes.
  + Declares an interface for accessing and managing its child components.
* **Leaf:**
  + Represents leaf objects in the composition.
    - A leaf has no children.
  + Defines behavior for primitive objects in the composition.
* **Composite:**
  + Defines behavior for components having children (add, remove, etc.).
  + Stores child components (some data structure, list?).
  + Implements child-related operations in the Component interface.
* **Client:**
  + Manipulates objects in the composition through the Component interface (does the grouping).

# Usage:

* A client uses the component class interface to interact with objects in the composition structure.
* If recipient is, a leaf then request is handled directly.
* If recipient is a composite, then it usually forwards request to its child components:
  + Also may perform additional operations before and after forwarding.

# Tradeoff:

* This pattern focuses on transparency and does not strictly follow **The Single Responsibility Design Principle.**
* Whether an element is a composite or leaf node is transparent to the client:
  + The Component interface is allowed to contain the child management operations and the leaf operations (add, remove, and shared operation)
    - Allows a client to treat both composites and leaf nodes uniformly.
    - Violates **The Single Responsibility Design Principle.**
      * We have both types of operations in the Component class.
        + We lose a bit of safety because a client might try to do something inappropriate or meaningless on an element (add or remove on a leaf objet itself instead of the composite).
* This is a design decision.
* We could take the design in the other direction and separate out the responsibilities into interfaces.
  + Makes our design safer.
    - Any inappropriate calls on elements would be caught at compile time or runtime.
      * We would lose transparency and our code would have to use conditionals and the **instanceof** operator.
* This is a classic case of a tradeoff.
* We are guided by design principles, but we always need to observe the effect they have on our designs.
* Sometimes we purposely do things in a way that seems to violate a principle.
  + However, this is a matter of perspective.
    - It might seem incorrect to have child management operations in the leaf nodes (like add (), remove () and getChild ()).
    - Then again, you can always shift your perspective and see a leaf as a node with zero children.

# Summary:

* Defines class hierarchies consisting of primitive objects and composite objects.
* Primitive objects can be composed into more complex objects.
  + Wherever client code expects a primitive object, it can also take a composite object.
* Clients can treat composite structures and individual objects uniformly:
  + Clients do not know whether they are dealing with a leaf or a composite component.
  + Simplifies client code, because it avoids having to write case statement style functions over the classes that define the composition.
* Newly defined composite or leaf subclasses work automatically with existing structures and client code:
  + Clients do not have to be changed for new component classes.