# Overview:

* The Iterator design pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
* An aggregate object is an object that contains other objects for the purpose of grouping those objects as a unit:
  + Also called a container or a collection.
  + Examples are linked list and a hash table.
* Iterators are generally used to traverse a container to access its elements.
* It is a very commonly used design pattern in Java with the collection framework:
  + Used to access the elements of a collection object.
* The pattern hides the actual implementation of traversal through the collection:
  + Client programs just use iterator methods.
* As mentioned, this pattern gives you a way to step through the elements of an aggregate without having to know how things are represented under the covers.
* An iterator object is responsible for keeping track of the current element.
  + It knows which elements have been traversed already.
* Once you have a uniform way of accessing the elements of all your aggregate objects, you can write polymorphic code that works with any of these aggregates.
* The iterator allows different traversal methods (forwards and backwards).
* Allows multiple traversals to be in progress concurrently.
* Places the task of traversal on the iterator object, not on the aggregate
  + Simplifies the aggregate interface and implementation.
  + Places the responsibility where it should be.
    - Keeps the aggregate focused on the things it should be focused on (managing a collection of objects), not on iteration.

# Examples:

* Suppose there are two companies, company A and company B.
* Company A stores its employee records (name, etc.) in a linked list.
* Company B stores its employee data in a big array.
* One day the two companies decide to work together:
  + The iterator pattern will allow us to have a common interface through which we can access data for both companies.
  + We will simply call the same methods without rewriting any code.
* Another example would be in a collage:
  + The arts department may use an array data structure.
  + The science department may use a linked list data structure to store their student’s records.
* The main administrative department will access those data through common methods using iterator:
  + It does not care which data structure is used by individual departments.

# When to use the iterator pattern?

* When you want, provide a standard way to iterator over a collection and hide the implementation logic from client program.
  + Logic for iteration is embedded in the collection itself and it helps client program to iterate over them easily.
* Use the pattern to support multiple traversals of aggregate objects.
* Use the pattern to support polymorphic iteration.

# Implementation:

# Participants:

* **Iterator:**
  + Defines an interface for accessing and traversing elements.
* **ConcreteIterator:**
  + Implements the Iterator interface.
  + Keeps track of the current position in the traversal of the aggregate.
* **Aggregate:**
  + Defines an interface for creating an Iterator object.
* **ConcreteAggregate:**
  + Implements the iterator creation interface to return an instance of the proper **ConcreteIterator**.
  + A ConcreteIterator keeps track of the current object taken together and can compute the succeeding object in the traversal.

# Implementation consequences:

* The implementation supports variations in the traversal of an aggregate
  + May traverse the parse tree in order or preorder.
* Iterators make it easy to change the traversal algorithm
  + Just replace the iterator instance with a different one.
  + You can also define Iterator subclasses to support new traversals.
* More than one traversal can be pending on an aggregate:
  + An iterator keeps track of its own traversal state.
  + You can have more than one traversal in progress at once.