**GENERAL TEST INFO**

* LEAVE SPACE FOR JAVADOC + WRITE CODE OUT FIRST (except for Q2 – Programming by Contract)
* COME BACK TO UML DIAGRAMS Q1 Q4 AT THE END
* COME BACK TO JAVADOC FOR Q2 Q3 Q4 AT THE END.
* For UMLs: Draw boxes and lines first, leave arrows until last

**Programming by Contract**

**Programming by Contract** is a concept which creates a contract between the software developer and the software user.

Every method starts with a **Pre-condition** that must be satisfied by the user of the feature.

* They don’t have to be handled by the developer
* If they hold, error checking is redundant

Every method ends with a **Post-condition** which the developer guarantees to be true (IFF pre-conditions were met when the method is called).

* The developer guarantees expected behaviour for the end user

**/\*  
 \* Method to deposit money into a bank account  
 \* @param amount is the amount to be deposited  
 \* @pre amount > 0  
 \* @post balance = balance + amount  
 \*/  
Public void deposit(float amount) { code inside method }**

Each class has an **Invariant** which is a logical condition that must be satisfied after any changes are made to the object represented by the class. It guarantees that the object is in a valid state.

* The condition is true after the constructor has completed execution (guarantees no invalid objects are created)
* The condition is true before and after a method call, but it can be temporarily violated during execution of a method.

**/\*   
 \* Bank Account handles the deposit and withdrawal of funds  
 \* @invariant balance >= 0  
 \*/  
Public class BankAccount() { variables and methods inside class }**

Other notes

* Getters don’t really need @pre @post as no state is being changed in the object. The user is simply viewing the object.
* **Calendar.getInstance()** returns a calendar object whose fields have been initialised with the current date and time.
* **currDate.get(Calendar.DATE)** returns the DATE field from the Calendar object.

**Generic Types / Polymorphism**

**Generics** allow you to customise a “generic” method or class to whatever type you’re working with.

Set<E>, List<E>, Map<E>, Graph<E> are interfaces. You can implement it using any underlying structure you want. It just has to support the standard interface methods + distinct element property.

Interfaces

|  |  |
| --- | --- |
| **Set<E>**  NO DUPLICATES  UN-ORDERED | **void add(E e)**  **void remove(E e)**  **boolean contains(E e)**  **Set<E> union(Set<E> other)**  **Set<E> intersection(Set<E> other)**  **boolean isSubset(Set<E> other)**  **boolean equals(Object o)** |
| **List<E>**  DUPLICATES  ORDERED | **void add(E e)** 🡪 adds to end of list **void add(int index, E e)** 🡪 adds to specified index in list  **void remove(E e)** 🡪 removes element  **void remove(int index)** 🡪 removes at specified index  **List<E> subList(int from, int to)** 🡪 returns view of sublist (from = inclusive, to = exclusive)  **E get(int index)** 🡪 returns element  **boolean contains(E e)** 🡪 T/F check for element  **boolean isEmpty()** 🡪 T/F check for empty  **boolean equals(Object o)** |
| **Map<K,V>** | **void put(V val)** 🡪 put value into map  **void remove(K key)** 🡪 remove value from map  **void replace(K key, V value)** 🡪 replace an entry of a specified key with the specified value  **V get(K key)** 🡪 get value from map  **boolean containsKey(K key)** 🡪 T/F check for key  **boolean containsValue(V vak)** 🡪 T/F check for value  **boolean isEmpty()** 🡪 T/F check for empty  **boolean equals(Object o)** |

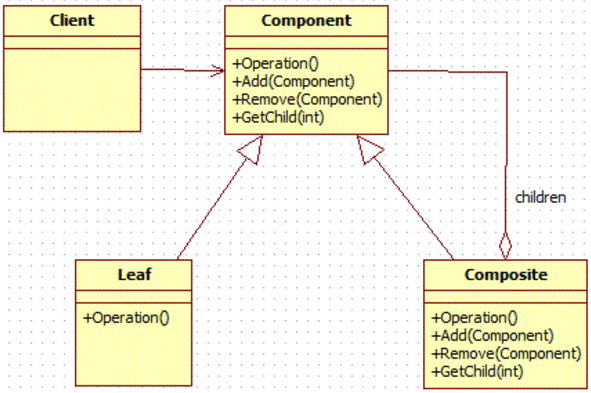
**public class Set<E> extends Iterable<E> {** /\* interface code \*/ **}**

* Extends = making a subclass

**public class ArraySet<E> implements Set<E> {** /\* class code \*/ **}**

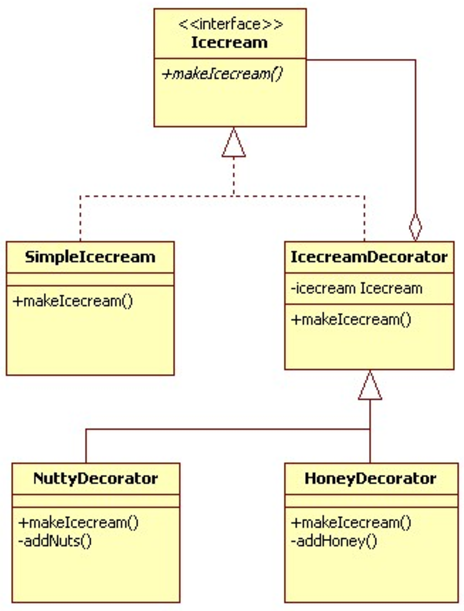
* Implements = implementing an interface 🡪 Animal implements AnimalInterface

**Design Patterns**



**Composite Pattern** is used when creating a class that contains a group of its own objects. This class provides ways to modify its group of same objects. (A tree-like structure)

* E.g. A composite object COMPUTER  
  🡪 composite object MOTHERBOARD  
  🡪 leaf object RAM + leaf object GRAPHICS CARD
* Each of these composite objects / leaf objects will have the same operations.  
  E.g. getPrice() for each component.



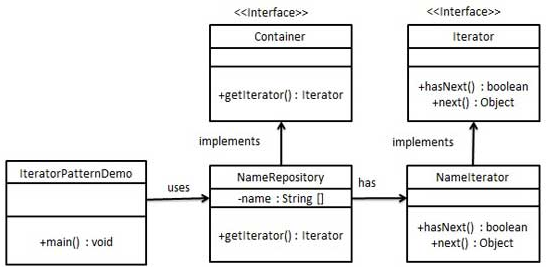
**Decorator Pattern** allows a user to add new functionality to an existing object without altering its structure.

* Using the example above, you can include DISCOUNTED PRICES to both your composite / leaf objects.

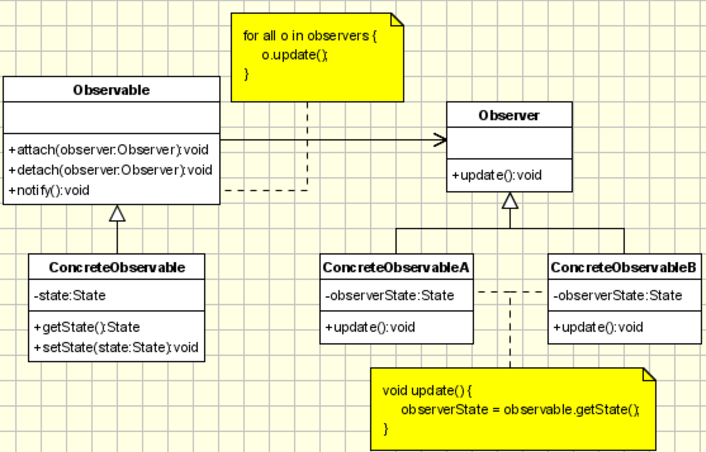
**Super** is used to call a method or constructor defined in the super-class.

* E.g. if **DiscountComponent extends DecoratorComponent** , super = DecoratorComponent
* Calling super methods: **super.method();**
* Calling super constructor: **super(constructor arguments);**

**Iterator Pattern** is used to access elements of a collection of objects in a sequential order, without need to know its underlying representation (i.e. list, set, map etc.)

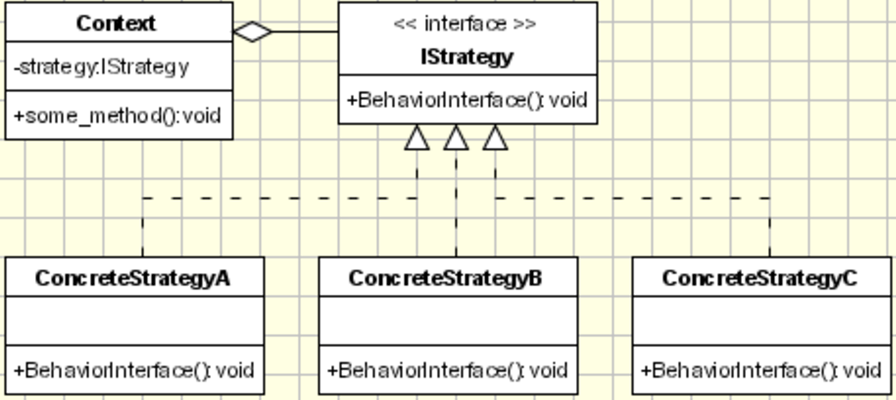


**Observer Pattern** is used when there is a one-to-many relationship between objects, such as when one is modified, its dependent objects are to be notified automatically.



**Strategy Pattern** is used when there are multiple possible algorithms / class behaviours to select from.

* Define a family of algorithms, encapsulate each one, make them interchangeable.
* Strategy lets the algorithm vary independently from clients that use it.



(JUST IN CASE) **Model View Controller (MVC) Pattern**

1. **Model** represents the data and does nothing else.
2. **View** displays the model data and sends users actions (e.g. button clicks) to the controller.
3. **Controller** provides model data to the view, interprets user actions such as btn clicks.

Example: Address Book Program

* Model = Person objects / data
  + Name, picture, address
* View = GUI
  + Window that displays the list of people
  + Detect mouse hover over events
  + Detect mouse click events
  + Detect keyboard press events
* Controller = Handles actions
  + Delete person
  + Add person
  + Email person