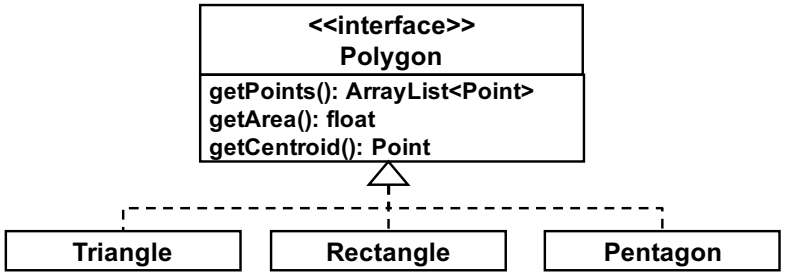
Strategy Design Pattern

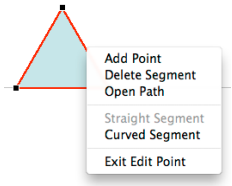
Strategy Design Pattern

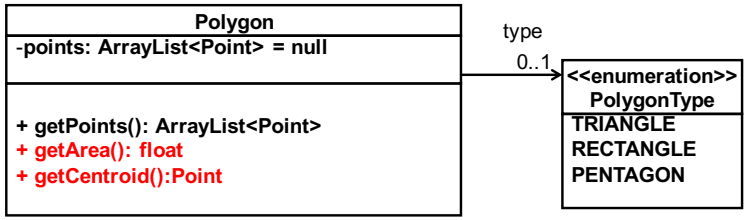
- Intent
  - Define a family of algorithms.
  - Encapsulate each algorithm in a class.
  - Make them interchangeable.
- a.k.a.
  - Policy

Recap



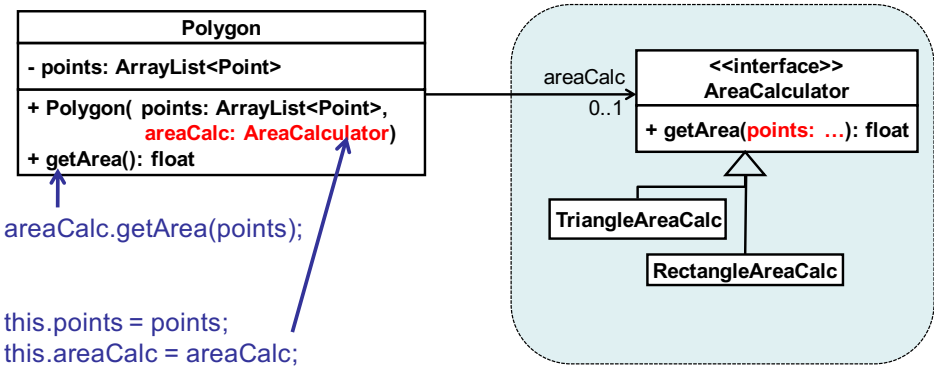
- Can a triangle become a rectangle dynamically?
- Do we allow that?
  - Maybe.
- Eliminate class inheritances





- Avoid conditional statements, particularly long and complicated ones.

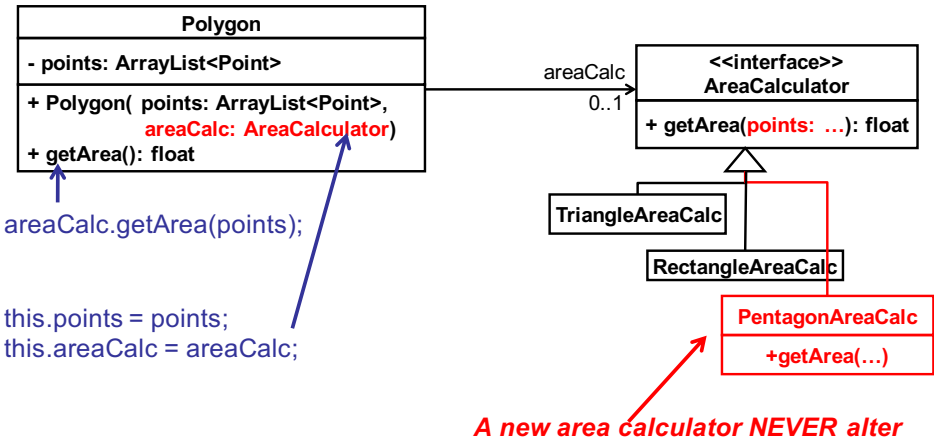
An Example of *Strategy*



Strategy Pattern:
Area calculation is "strategized."

User/client of Polygon:

```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add( new Point(…) ); al.add( new Point(…) );
Polygon p = new Polygon( al, new TriangleAreaCalc() );
p.getArea();
```

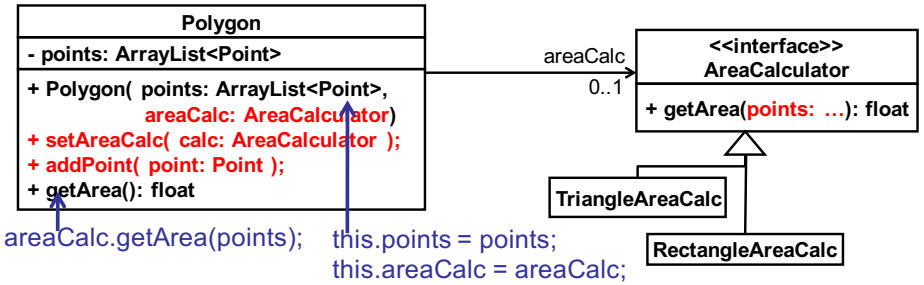



A new area calculator NEVER alter existing code (i.e., Polygon, area calcurators, and clients of Polygon).

User/client of Polygon:

```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add( new Point(…) ); al.add( new Point(…) );
Polygon p = new Polygon( al, new TriangleAreaCalc() );
p.getArea();
```


Polygon Transformation

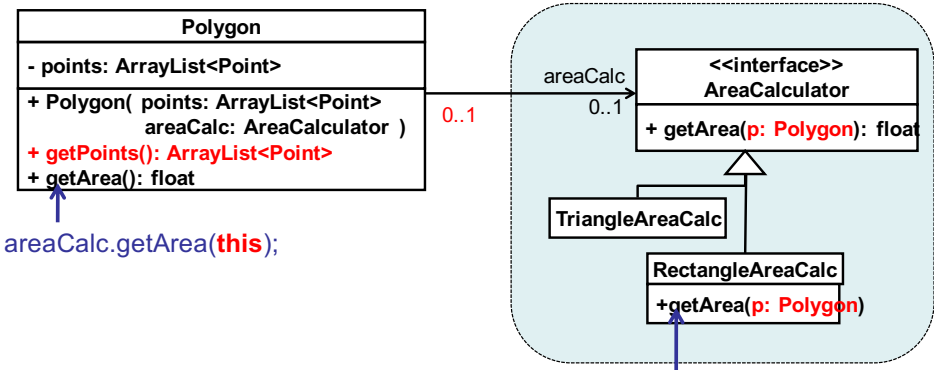


User/client of Polygon:

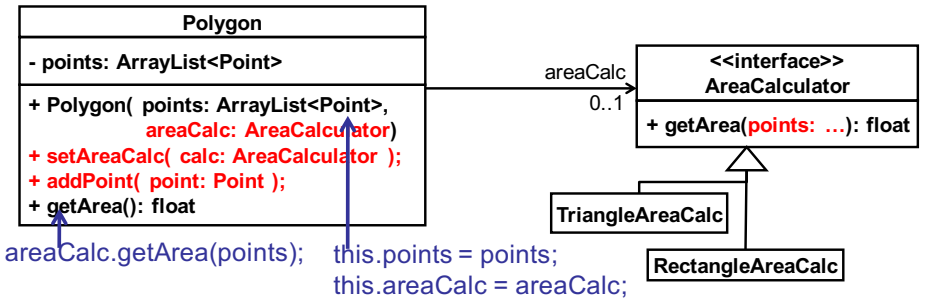
```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add( new Point(…) ); al.add( new Point(…) );
Polygon p = new Polygon( al, new TriangleAreaCalc() );
p.getArea(); // triangle's area
p.addPoint( new Point(…) );
p.setAreaCalc( new RectangleAreaCalc() );
p.getArea(); // rectangle's area
```

No changes in existing code. Dynamic polygon transformation. Dynamic replacement of area calculators

An Alternative



HW11-1: Implement this

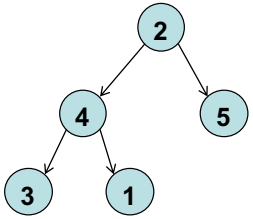


User/client of Polygon:

```
ArrayList<Point> al = new ArrayList<Point>();
al.add( new Point(…) ); al.add( new Point(…) ); al.add( new Point(…) );
Polygon p = new Polygon( al, new TriangleAreaCalc() );
p.getArea(); // triangle's area
p.addPoint( new Point(…) );
p.setAreaCalc( new RectangleAreaCalc() );
p.getArea(); // rectangle's area
```


- Extend your previous HW solution/program.
  - [optional] getCentroid(), CentroidCalculator
- Transform a triangle to a rectangle dynamically
- Transformation from a rectangle to a triangle dynamically

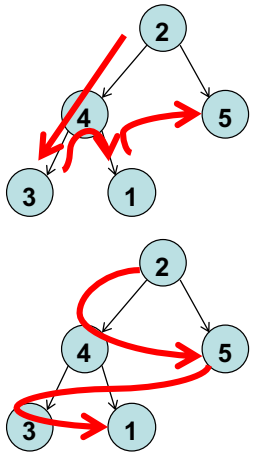
Tree Traversal with *Strategy*



In a tree, each node contains a number.
Search a particular number from the tree.
Traverse the tree (i.e., visit all nodes one by one in the tree) and sum up the numbers associated with individual nodes.

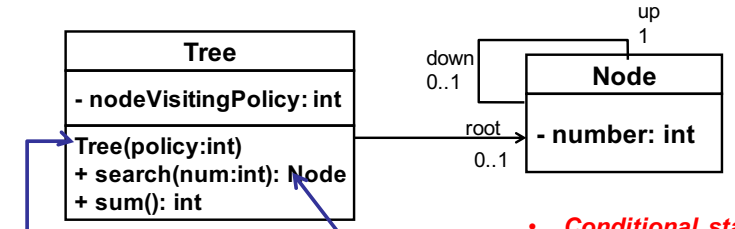
- Two major (well-known) algorithms
  - Depth-first
  - Width-first

Tree Traversal Policies/Algorithms



- Two major (well-known) algorithms
  - Depth-first
  - Width-first
- Assume
  - Different trees may have different search/traversal policies.
  - You may need to dynamically change one policy to another.

Not Good

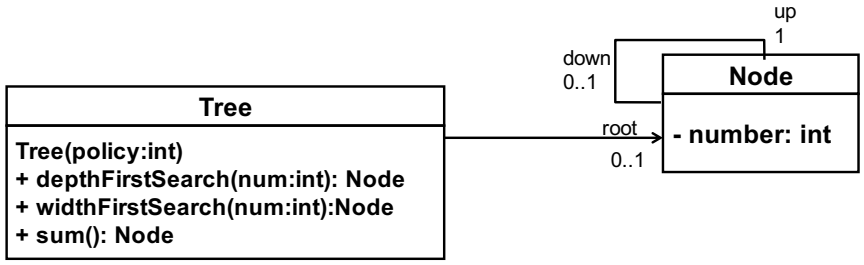


- *Conditional statements*
- *Need to modify existing code (search()) when new visiting policies are introduced.*

```
nodeVisitingPolicy = policy;
```

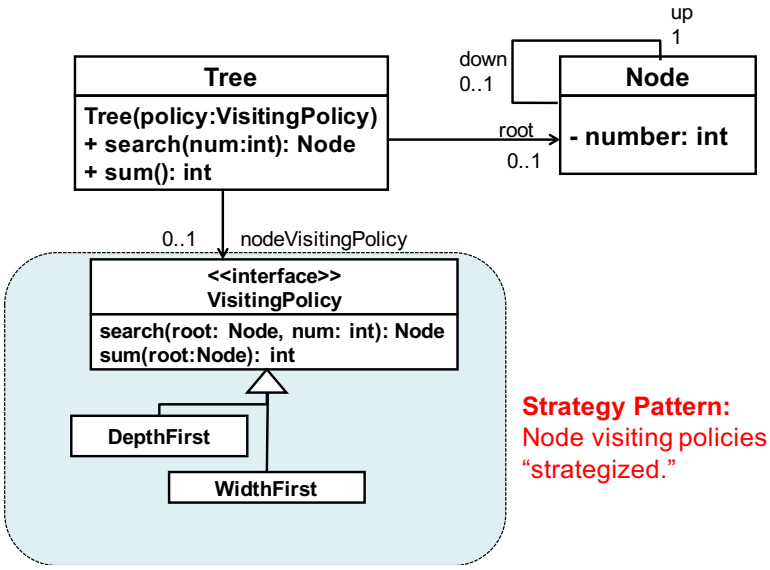
```
if(nodeVisitingPolicy==1){
  // do depth-first search
}
else if(nodeVisitingPolicy==0){
  // do width-first search
}
```


An Alternative

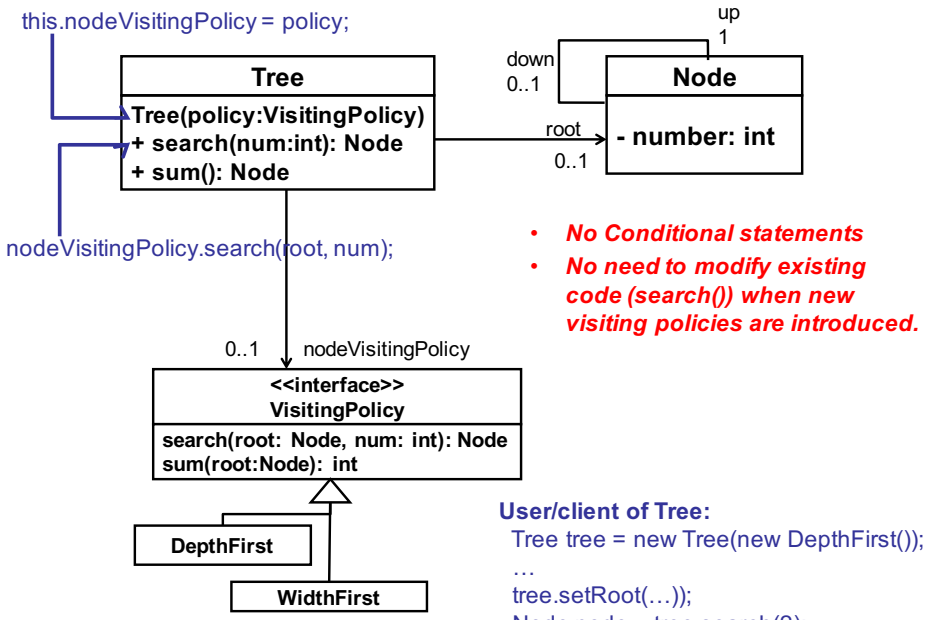


- *No Conditional statements in search()*
- *Still need to modify existing code (Tree) when new visiting policies are introduced.*

With Strategy Classes…



Strategy Pattern:
Node visiting policies are "strategized."



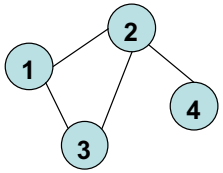
- *No Conditional statements*
- *No need to modify existing code (search()) when new visiting policies are introduced.*

User/client of Tree:

```
Tree tree = new Tree(new DepthFirst());
…
tree.setRoot(…));
Node node = tree.search(2);
```

Graph Traversal with *Strategy*

- A graph consists of nodes and links.
- Requirement: Find the shortest path between given two nodes.
  - 1 -> 2 -> 4: 2 hops between Node 1 and Node 4
  - 2 -> 3: 1 hop between Node 2 and Node 3

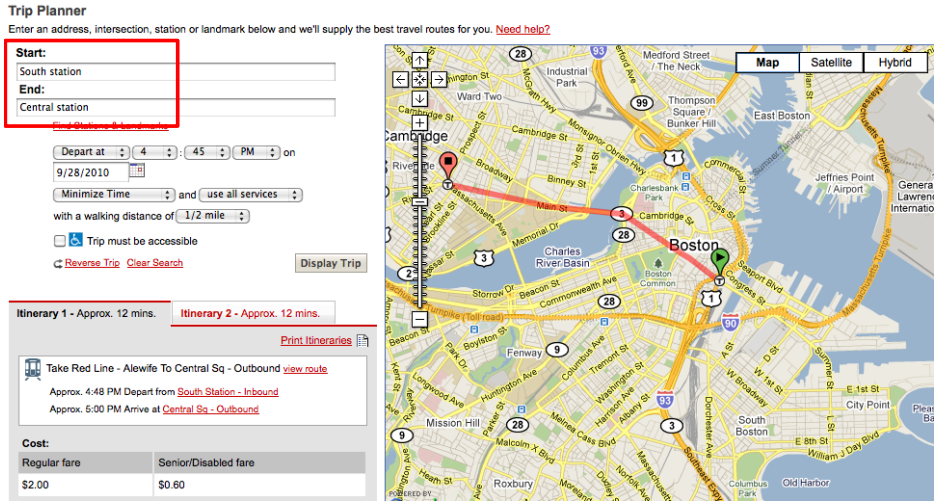


Graph
- graph: ArrayList<ArrayList<Boolean>>
+ Tree(graph:ArrayList<ArrayList<Boolean>>) + getShortestPath(n1:int, n2:int): ArrayList<Integer>

- 1 1 1 0
  1 1 1 1
  1 1 1 0
  0 1 0 1

  Connectivity matrix

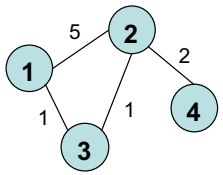
Trip Planner at mbta.org



- Directions from one place to another via T (e.g. South Station to Central Sq.)
- This is a shortest path search problem.

Weighted Graphs

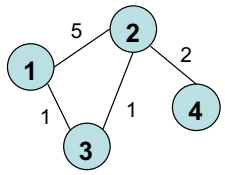
- What if you need to consider the *weighed* shortest path between two nodes?
  - 1 -> 3-> 2-> 4: total weight = 4, between Node 1 and Node 4
  - 1 -> 3 -> 2: total weight = 2, between Node 1 and Node 2



Graph
- graph: ArrayList<ArrayList<Integer>>
Tree(graph:ArrayList<ArrayList<Integer>>) + getShortestPath(n1:int, n2:int): ArrayList<Integer>

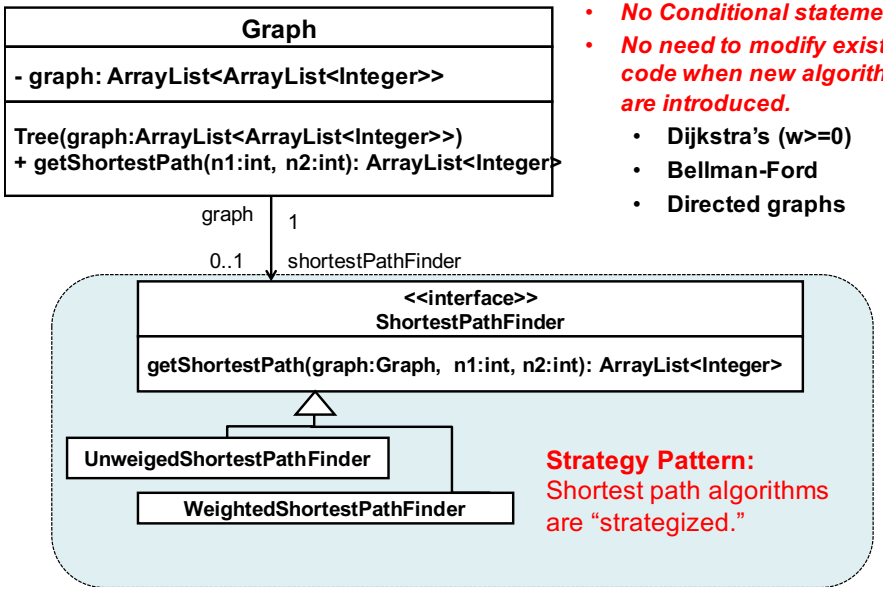
- 0 5 1 -1
  5 0 1 2
  1 1 0 -1
  -1 2 -1 0

  Weighted connectivity matrix



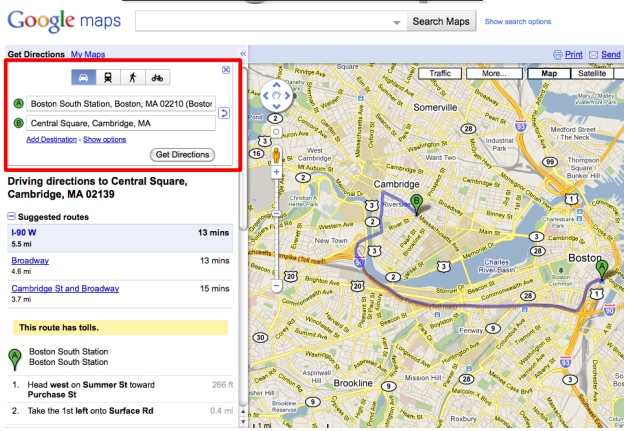
Graph
- graph: ArrayList<ArrayList<Boolean>>
Tree(graph:ArrayList<ArrayList<Boolean>>) + getShortestPath(n1:int, n2:int): ArrayList<Integer>

- ***Add conditional statements in getShortestPath()***
  - ***Conditional statements***
  - ***Need to modify existing code when new algorithms are introduced to compute the shortest path.***
- ***Add getWeightedShortestPath(...)***
  - ***No conditional statements***
  - ***Still need to modify existing code when new algorithms are introduced to compute the shortest path.***



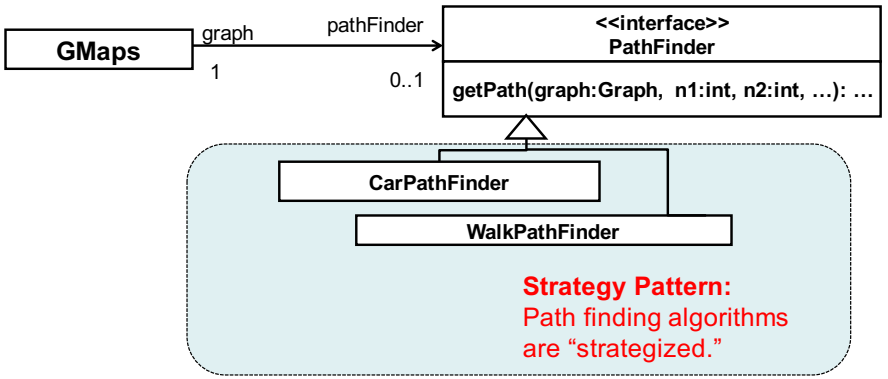
- ***No Conditional statements***
- ***No need to modify existing code when new algorithms are introduced.***
  - **Dijkstra's (w>=0)**
  - **Bellman-Ford**
  - **Directed graphs**

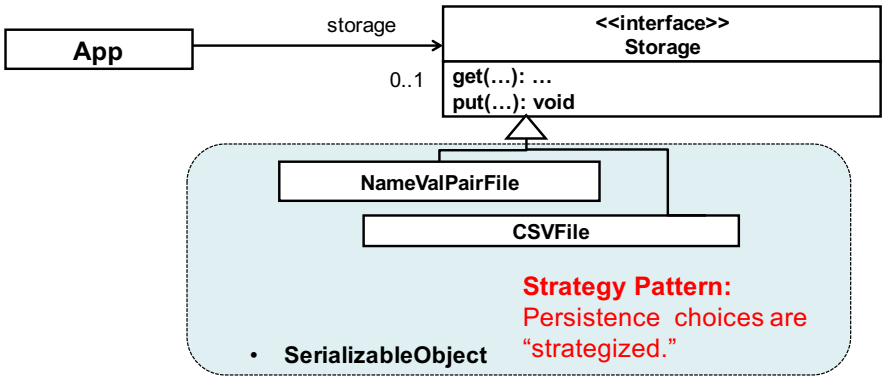
Google Maps



- Directions from one place to another (e.g. South Station to Central Sq.)
  - By car
  - By T
  - By walk
  - By bicycle
  - … extra transportation means in the future?

Data Storage (Persistence)





- **SerializableObject**
- **XML file**
- **YAML file**
- **JASON**

Object Pooling

- Strategy can be applied well to design various object pooling mechanisms.
  - Caching
  - Thread pooling
  - DB connection pooling
  - Network connection pooling

- c.f. a lecure note on *Factory Method*

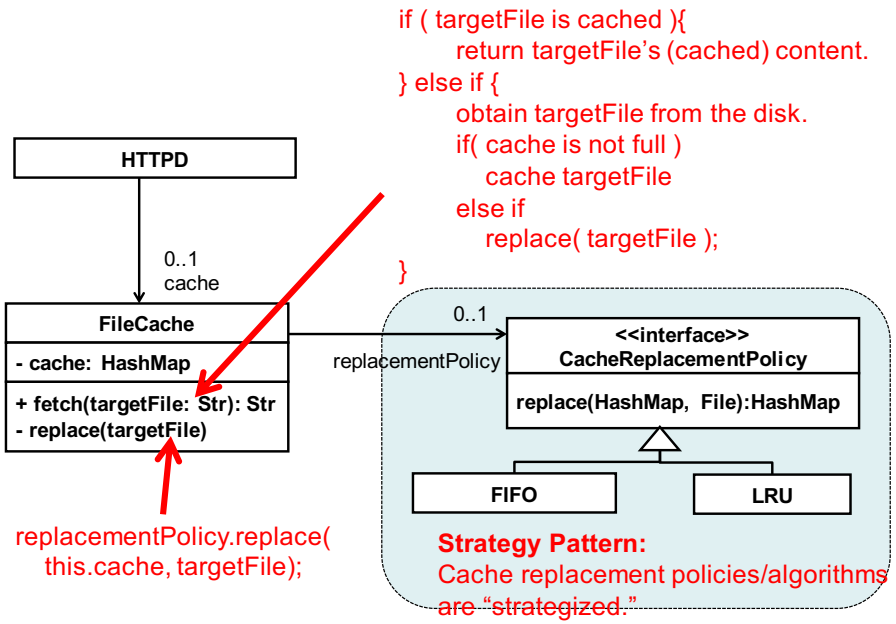
Web Page Caching

- A web server
  - Receives a connection establishment request from a client (browser).
  - Receives an HTTP command
  - Retrieves a target HTML file from the local disk and returns it to the client.

  - May cache a set of HTML files that have been accessed in the past.
    - Keep them in the memory
    - Faster response to the clients that request cached files
      - Memory access is much faster than disk access.

- However, the cache size is limited
  - Memory size is limited.
  - Need to determine which files to keep cached and which ones to be removed
    - When the size of cached files reaches the maximum cache size.

- Different cache replacement policies/algorithms
  - FIFO (First In First Out)
  - LRU (Least Recently Used)
  - LFU (Least Frequently Used)
  - … etc.



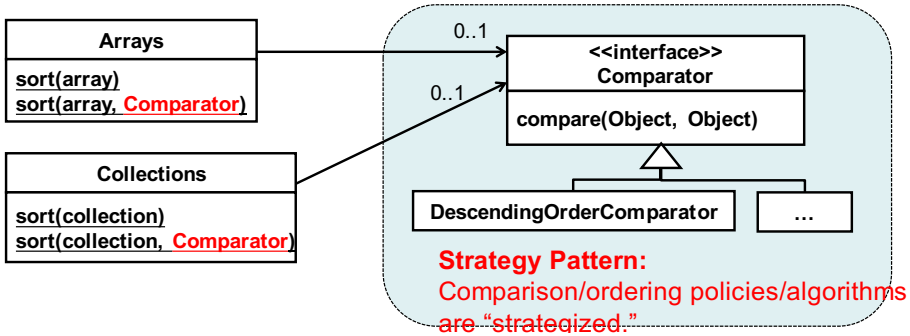
Comparators

- Sorting array elements:
  - ```
    int years[] = {2010, 2000, 1997, 2006};
    Arrays.sort(years);
    for(int y: years)
        System.out.println(y);
    ```
  - java.util.Arrays: a utility class (a collection of static methods) to process arrays and array elements
  - sort() sorts array elements in an ascending order.
    - 1997 -> 2000 -> 2006 -> 2010

- Sorting collection elements:
  - ```
    ArrayList<Integer> years2 = new ArrayList<Integer>();
    years2.add(new Integer(2010));
    years2.add(new Integer(2000));
    years2.add(new Integer(1997));
    years2.add(new Integer(2006));
    Collections.sort(years2);
    for(Integer y: years2)
        System.out.println(y);
    ```
  - java.util.Collections: a utility class (a collection of static methods) to process collections and collection elements
  - sort() sorts array elements in an ascending order.
    - 1997 -> 2000 -> 2006 -> 2010

Comparison/Ordering Policies

- What if you want to sort array/collection elements in a descending order or any specialized (user-defined) order?
  - Arrays.sort() and Collections.sort() implement ascending ordering only.
    - They do not implement any other policies.
- Define a custom comparator by implementing java.util.Comparator



- Arrays.sort() and Collections.sort() are defined to sort array/collection elements from "smaller" to "bigger" elements.
  - By default, "smaller" elements mean the elements that have *lower* numbers.
- A descending ordering can be implemented by treating "smaller" elements as the elements that have *higher* numbers.
- compare() in comparator classes can define (or re-define) what "small" means and what's "big" means.
  - Returns a negative integer, zero, or a positive integer as the first argument is "smaller" than, "equal to," or "bigger" than the second.

```
public class DescendingOrderComparator implements Comparator{
  public int compare(Object o1, Object o2){
    return ((Integer)o2).intValue()-((Integer) o1).intValue();
  }
}
```

Sorting Collection Elements with a Custom Comparator

```
ArrayList<Integer> years = new ArrayList<Integer>();
years.add(new Integer(2010)); years.add(new Integer(2000));
years.add(new Integer(1997)); years.add(new Integer(2006));
Collections.sort(years);
for(Integer y: years)
    System.out.println(y);
Collections.sort(years, new DescendingOrderComparator());
for(Integer y: years)
    System.out.println(y);
```

– 1997 -> 2000 -> 2006 -> 2010

– 2010 -> 2006 -> 2000 -> 1997


```
public class DescendingOrderComparator implements Comparator{
  public int compare(Object o1, Object o2){
    return ((Integer)o2).intValue()-((Integer) o1).intValue();
  }
}
```

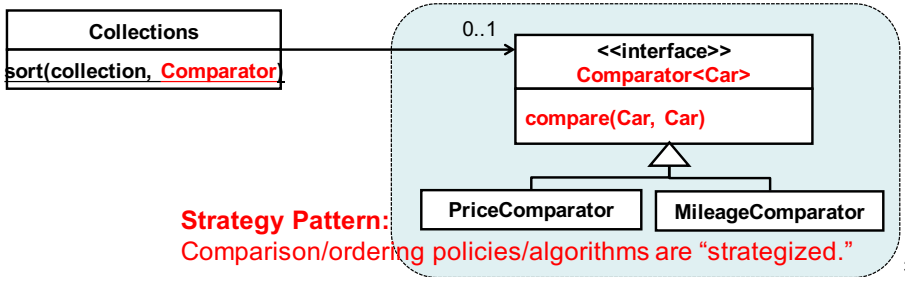
- A more type-safe option is available/recommended:

```
public class DescendingOrderComparator<Integer>{
  implements Comparator<Integer>{
    public int compare(Integer o1, Integer o2){
      return o2.intValue()-o1.intValue();
    }
}
```


- What if you want to sort a collection of your own (i.e., user-defined) objects?

```
class Car{
  private int getPrice();
  private int getYear();
  private float getMileage();
}
ArrayList<Car> usedCars= new ArrayList<Car>();
usedCars.add(new Car(…)); usedCars.add(…); usedCars.add(…);
Collections.sort(usedCars);
```

- Need to define and use a car-ordering policy in a custom comparator class.



Strategy Pattern:
Comparison/ordering policies/algorithms are "strategized."

- Assume "bigger" (or better) elements as the elements that have a
  - Lower price
  - Higher (more recent) year
  - Lower mileage

```
public class PriceComparator<Car>
  implements Comparator<Car>{
    public int compare(Car car1, Car car2){
      return car2.getPrice() - car1.getPrice();
    }
}
public class YearComparator<Car>
  implements Comparator<Car>{
    public int compare(Car car1, Car car2){
      return car1.getYear() - car2.getYear();
    }
}
```

Thanks to *Strategy*…

- You can define any extra ordering/comparison policies without changing existing code
  - e.g., Car, Collections.sort()
  - No conditional statements to shift ordering/comparison policies.
- You can dynamically change one ordering/comparison policy to another.

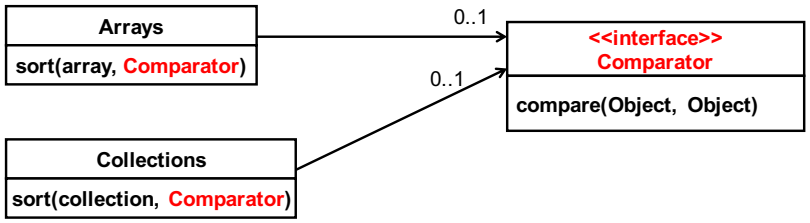
```
Collections.sort(usedCars, new PriceComparator());
// printing a list of cars
Collection.sort(usedCars, new YearComparator());
// printing a list of cars
```


Used Car Listings

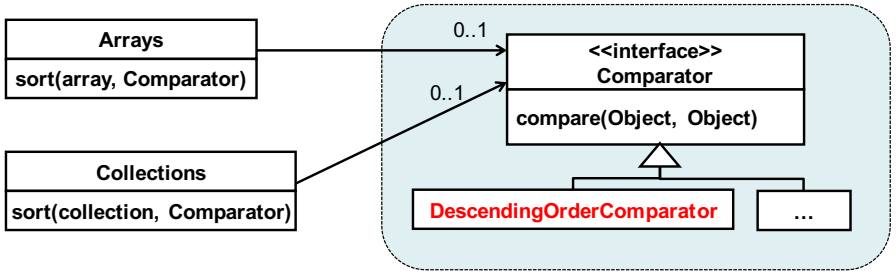
Year/Model	Information	Mileage	Seller/Distance	Price
2000 Audi A4 5dr Wgn 1.8T Avant Auto Quattro AWD				$4,880
	Used MPG: 19 Cty / 28 Hwy Automatic Gray	136,636	Dedham Auto Mall (7.4 Miles) Search Dealer Inventory	Free CARFAX Report
2001 Audi A4				$7,995
	Used	84,297	Herb Connolly Hyundai (18.8 Miles) Search Dealer Inventory	Free CARFAX Report
2002 Audi A6 4dr Sdn quattro AWD Auto				$7,998
	Used MPG: 17 Cty / 25 Hwy Automatic Blue	84,272	Dedham Auto Mall (7.4 Miles) Search Dealer Inventory	Free CARFAX Report
2003 Audi A4 1.8T				$10,697
	Used MPG: 20 Cty / 28 Hwy Automatic Blue	78,321	Direct Auto Mall (18.8 Miles) Search Dealer Inventory	Free CARFAX Report
2002 Audi allroad 5dr quattro AWD Auto				$10,900
	Used MPG: 15 Cty / 21 Hwy Automatic Green	98,362	Lux Auto Plus (8.6 Miles) Search Dealer Inventory	Get a CARFAX Record Check

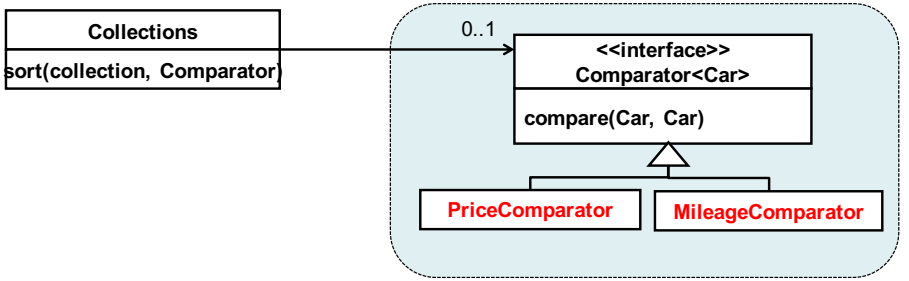
Year/Model	Information	Mileage	Seller/Distance	Price
2008 Audi A6				$37,897
	Certified Pre-Owned MPG: 17 Cty / 25 Hwy Automatic	0	Audi Burlington & Porsche of Burlington (14.9 Miles) Search Dealer Inventory	Free CARFAX Report
2007 Audi A4				$24,995
	Used MPG: 22 Cty / NA Hwy Brilliant Black	6,822	(19.3 Miles)	Get a CARFAX Record Check
2009 Audi A4				$33,497
	Certified Pre-Owned White	10,120	Audi Burlington & Porsche of Burlington (14.9 Miles) Search Dealer Inventory	Free CARFAX Report
2009 Audi A4 3.2L Prestige				$39,877
	Certified Pre-Owned MPG: 17 Cty / 26 Hwy Automatic White	12,118	Audi Burlington & Porsche of Burlington (14.9 Miles) Search Dealer Inventory	Free CARFAX Report
2008 Audi S5				$44,995
	Used Brilliant Black	16,492	(19.3 Miles)	Get a CARFAX Record Check

Programming to an Interface

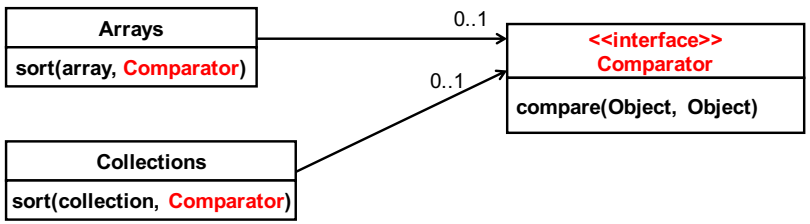


- One of the most important points in *Strategy*
- What Java API designers did was to…
  - Define the interface Comparator only.
    - Users of Arrays/Collections will define their comparator implementations.





Programming to an Interface



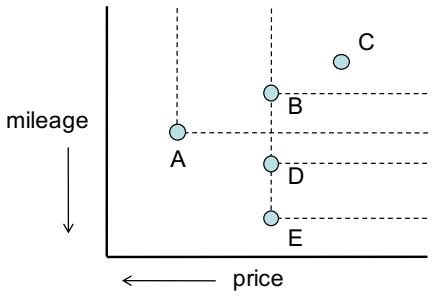
- What Java API designers intended was to…
  - make Arrays.sort() and Collections.sort() intact even if changes occur in Comparator implementations
  - make Arrays and Collections loosely-coupled from Comparator implementations.

HW11-2

- Implement the Car class and three comparator classes
  - [Optional] Implement an extra comparator, `ParetoComparator<Car>`, which performs the *Pareto comparison*.
- main() makes several cars and sort them with each of three (or four) comparators.

Pareto Comparison

- Given multiple objectives,
  - e.g., price, year and mileage
- Car A is said to ***dominate*** (or outperform) Car B iif:
  - A's objective values are superior than, or equal to, B's in all objectives, and
  - A's objective values are superior than B's in at least one objective.



  - Count the number of cars that dominate a car in question
    - A: 0
    - B: 3 (A, D, E)
    - C: 4 (A, B, D, E)
    - D: 1 (E)
    - E: 0
  - Better cars have lower "domination counts."

- Implement getDominationCount() in Car.
- When to compute domination counts for individual cars?
  - Before calling sort()

```
// finish up computing domination counts for all cars.
Collections.sort(usedCars, new PriceComparator<Car>());
```

  - When calling sort()

```
Collections.sort(usedCars,
                 new PriceComparator<Car>(usedCars));
```