Implementing Aggregation and Composition

Ahmed Sabbir Arif

Copy Constructor

- Shallow and deep copy.

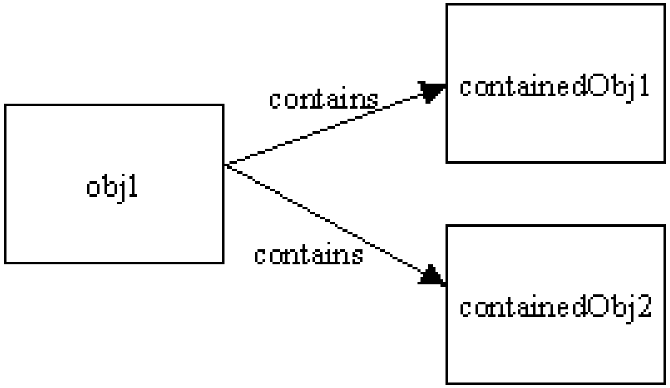


Image: javaworld.com

Shallow Copy

- A *copy* is attached to the same memory block as the *original*.
  - Hence, also known as *address copy*.
- Same data will be shared between the original and the copy.
  - Hence, modifying the one will alter the other.

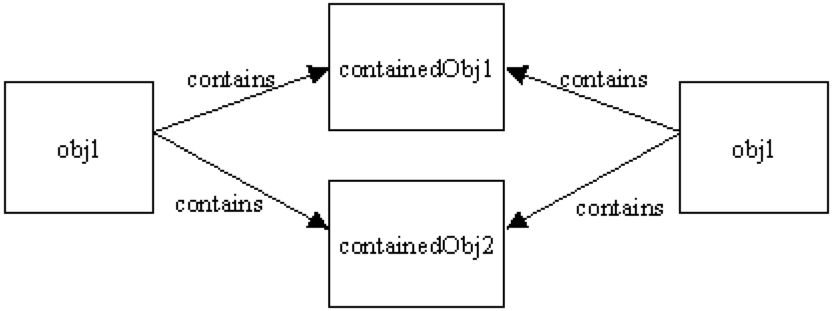


Image: javaworld.com

Deep Copy

- The data are actually copied over.
- The original and the copy do not depend on each other
  - But it is slower more expensive copy
- If a class is immutable, then there's no point making a deep copy.

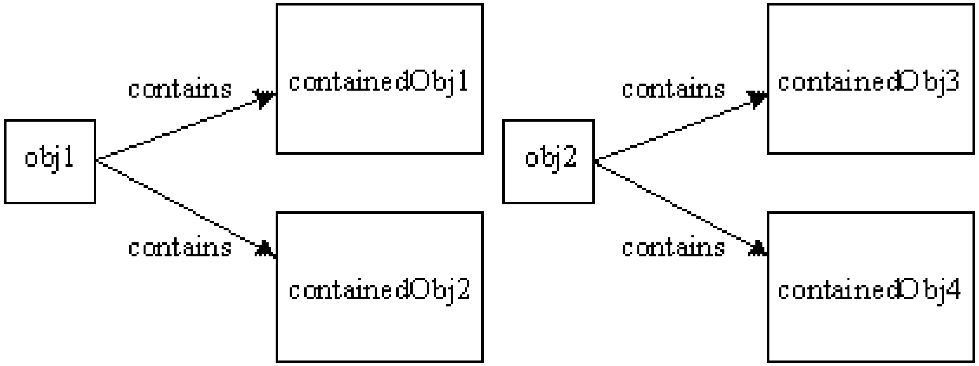


Image: javaworld.com

Composition

- The terms aggregation and composition are used to describe a relationship between objects.
- Both terms describe the ***has-a*** relationship:
  - The university *has-a* collection of departments.
  - Each department *has-a* collection of professors.
- Composition implies ownership:
  - If the university disappears then all of its departments disappear.
  - A university is a ***composition*** of departments.

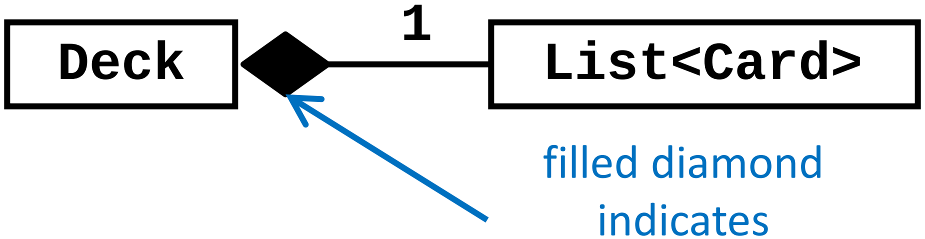
Example: A Deck of Cards

- A class to represent a deck of cards:
  - A deck has-a collection of 52 cards.
- The deck should own the cards:
  - A deck is a ***composition*** of cards.

- A client can ask the deck to deal a card:
  - The deck gives up ownership of the card.
  - The client takes ownership of the card.
- A client can try to give a dealt card back to the deck:
  - The deck takes ownership of the card *if and only if* it does not have the same card already.
  - Class invariant: **the cards in a deck are unique**.

- A client can ask for the deck to be shuffled.
- A client can ask to see all of the cards without the deck giving up ownership of any of the cards
  - We will implement the **`Iterable`** interface for the deck
  - Lets clients write code like:

```
Deck theDeck = new Deck();
for(Card c : theDeck)    // for each Card c in theDeck
{
   System.out.println(c);
}
```



filled diamond indicates composition

Iterators

- An iterator is an object that provides access to each element of a collection in sequential order.
- An iterator must implement the **`Iterator`** interface found in **`java.util`**.

```
public interface Iterator<E>
{
   public boolean hasNext();
   public E next();
   public void remove();     // optional
}
```

Iterable Interface

- Every **Collection** type supplies an iterator by implementing the **Iterable** interface

```
public interface Iterable<T>
{
  public Iterator<T> iterator();
}
```

Iterating with hasNext

- Prior to Java 1.5 you would iterate over a Collection like so:

```
// ...
ArrayList<String> lst = new ArrayList<String>();
lst.add("apple");
lst.add("banana");
lst.add("mango");
for(Iterator<String> iter = lst.iterator(); iter.hasNext(); )
{
  String s = iter.next();
  System.out.println(s.replace('a', 'A') + " ");
}
// prints Apple bAnAnA mAngo
```

Iterating with *for-each*

- The preferred method is to use a *for-each* loop:

```
// ...
ArrayList<String> lst = new ArrayList<String>();
lst.add("apple");
lst.add("banana");
lst.add("mango");
for(String s : lst)    // for each String s in lst
{
  System.out.println(s.replace('a', 'A') + " ");
}
// prints Apple bAnAnA mAngo
```

Deck

```
package playingcard;

import java.util.Collections;
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;

public class Deck implements Iterable<Card>
{
  private final List<Card> cards;
```

Deck

```
public Deck() {
  this.cards = new ArrayList<Card>();
  this.reset();
}
```

The Deck owns its Cards; thus it must create and own its own List of Cards.

```
public void reset() {
  this.cards.clear();
  final Set<String> keys = CardUtil.RANKS.keySet();
  for(String rank : keys) {
    for(String suit : CardUtil.SUITS) {
      cards.add(new Card(rank, suit));
    }
  }
}
```

Map does not supply an iterator to its keys or values, but it can return its keys as a Set.

Deck

```
public void shuffle()
{
  Collections.shuffle(this.cards);
}


public int getNumberOfCards()
{
  return this.cards.size();
}
```

Deck

```
public Card deal()
{
  Card c = null;
  if(this.getNumberOfCards() > 0)
  {
    c = this.cards.remove(0);
  }
  return c;
}
```

When a Deck deals a Card, it is giving up ownership of the Card; therefore, the Deck needs to remove the Card from its internal state. In this case, we always deal from the top of the Deck, so we remove the first Card stored in the List this.cards (the zeroth card).

Deck

```
public boolean take(Card c)
{
  boolean ok = false;
  if (!this.cards.contains(c))
  {
    ok = this.cards.add(c);
  }
  return ok;
}
```

A client can ask a Deck to take a Card from the client; in this case, the client is asking the Deck to take ownership of the Card. Because of the class invariant (Deck holds unique Cards), the take implementation must check that the Card c is not already in this.cards. The Deck takes ownership of the Card c if and only if it does not already have a card identical to c.

Implementing equals for Deck

- You might be tempted to write something like the following in **`equals()`**

```
eq = this.cards.equals(other.cards);
```

- Unfortunately, this does not work because the order of the list elements matters for **`List.equals()`**
  - A list with elements { 1, 2, 3, 4 } is not equals to a list with elements { 4, 3, 2, 1 }

- We can use **`List.containsAll()`**
  - A list with elements { 1, 2, 3, 4 } contains all of the elements in the list { 4, 3, 2, 1 }
  - But a list with elements { 1, 2, 3, 4 } also contains all of the elements in the list { 4, 3 }

- We will say that two Decks are equal if they have the same number of cards and they contain the same cards.

Deck equals

```
@Override public boolean equals(Object obj)  {
  boolean eq = false;
  if (this == obj) {
    eq = true;
  }
  else if( obj != null && this.getClass() == obj.getClass() )
  {
    Deck other = (Deck) obj;
    eq = this.cards.size() == other.cards.size() &&
         this.cards.containsAll(other.cards);
  }
  return eq;
}
```

Deck iterator

```
@Override public Iterator<Card> iterator()
{
  return this.cards.iterator();
}
```

Because Deck implements Iterable<Card>, we must provide a method that returns an iterator to a Card. We could implement our own Card iterator class, but the List contained by the Deck already supplies an iterator for us. In this case, we can just delegate to this.cards to get a suitable iterator.

Exercises

- Add a constructor that constructs a Deck given a Collection of Cards

```
public Deck(Collection<Card> cards)
```

  – Hint: use the Collections utility

- Add a method that sorts the Deck by rank

```
public void sort()
```

  – Hint: use the Collections utility

- Implement **`toString()`**