Aggregation and Composition

[notes Chapter 4]

Aggregation and 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
- aggregation does not imply ownership
  - if a department disappears then the professors do not disappear
  - a department is an *aggregation* of professors

Aggregation

- suppose a **`Person`** has a name and a date of birth

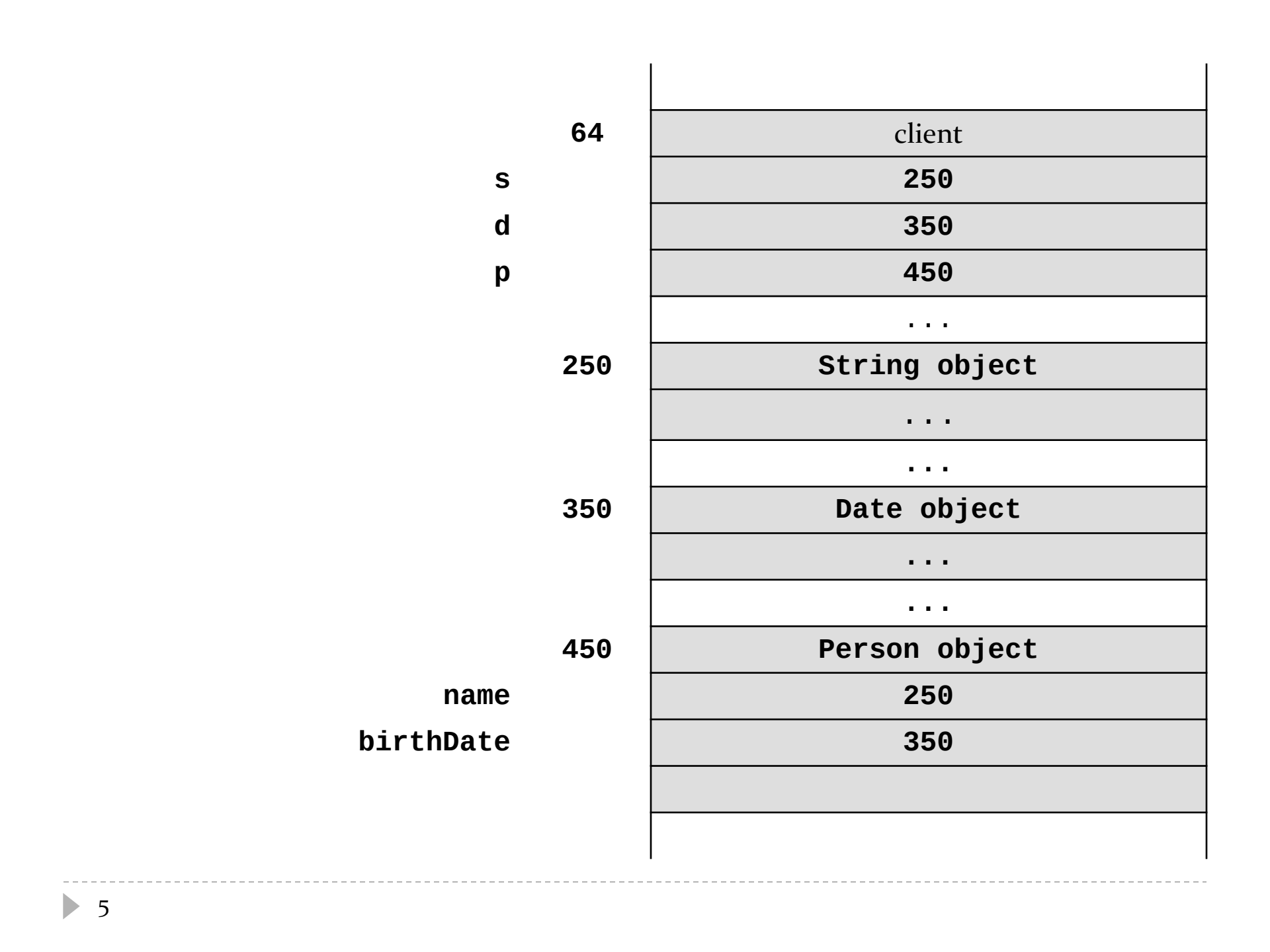
```
public class Person {
  private String name;
  private Date birthDate;

  public Person(String name, Date birthDate) {
    this.name = name;  this.birthDate = birthDate;
  }

  public Date getBirthDate() {
    return birthDate;
  }
}
```

- the **`Person`** example uses aggregation
  - notice that the constructor does not make a copy of the name and birth date objects passed to it
  - the name and birth date objects are shared with the client
  - both the client and the **`Person`** instance are holding references to the same name and birth date

```
// client code somewhere
String s = "Billy Bob";
Date d = new Date(91, 2, 26);  // March 26, 1991
Person p = new Person(s, d);
```


64
client
s
250
d
350
p
450
...
250
String object
...
...
350
Date object
...
...
450
Person object
name
250
birthDate
350

- what happens when the client modifies the **`Date`** instance?

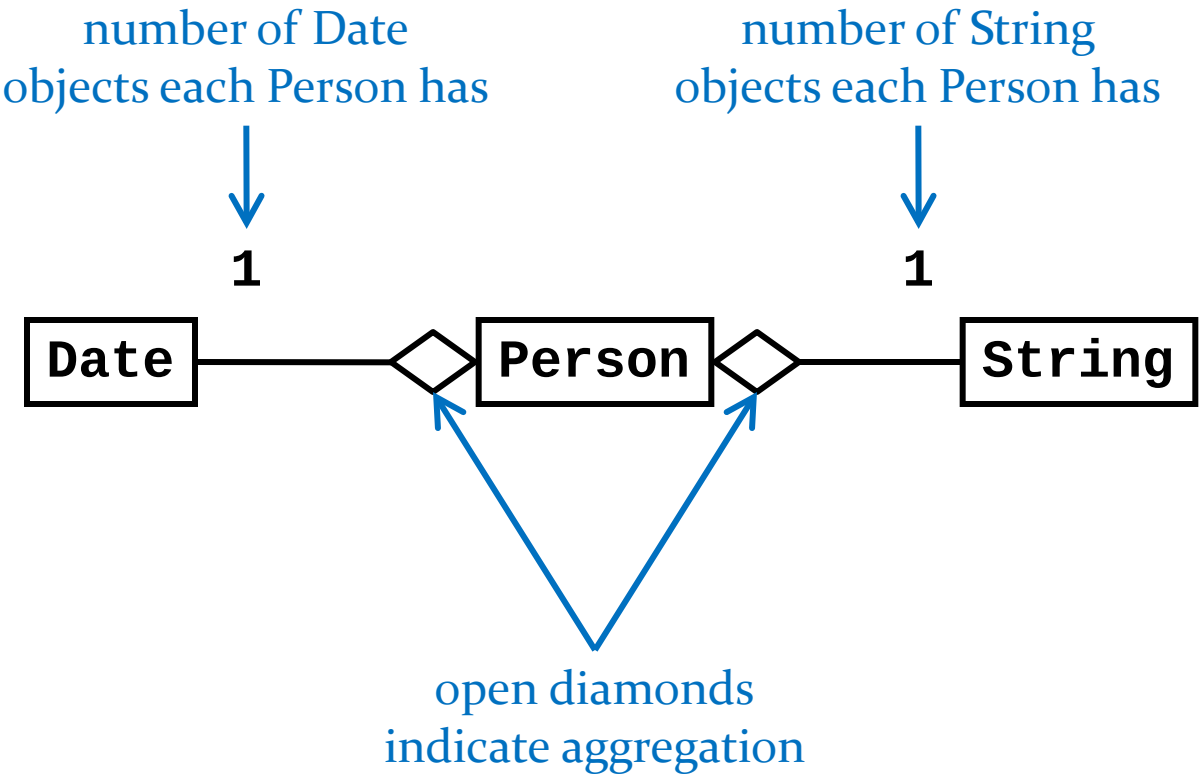
```
// client code somewhere
String s = "Billy Bob";
Date d = new Date(90, 2, 26);  // March 26, 1990
Person p = new Person(s, d);

d.setYear(95);                 // November 3, 1995
d.setMonth(10);
d.setDate(3);
System.out.println( p.getBirthDate() );
```

  - prints **`Fri Nov 03 00:00:00 EST 1995`**

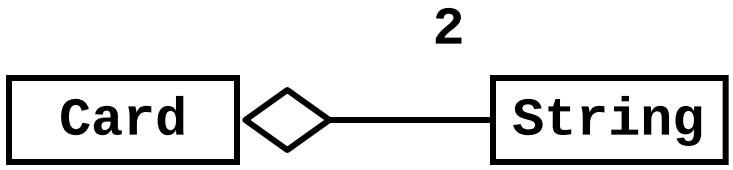
- because the **`Date`** instance is shared by the client and the **`Person`** instance:
  - the client can modify the date using **`d`** and the **`Person`** instance **`p`** sees a modified **`birthDate`**
  - the **`Person`** instance **`p`** can modify the date using **`birthDate`** and the client sees a modified date **`d`**
- note that even though the **`String`** instance is shared by the client and the **`Person`** instance **`p`**, neither the client nor **`p`** can modify the **`String`**
  - immutable objects make great building blocks for other objects
  - they can be shared freely without worrying about their state

UML Class Diagram for Aggregation



Aggregation Example

- suppose we want to implement a class to represent standard playing cards
  - a card has-a suit (club, diamond, heart, spade)
    - we will represent the suit with a String
  - a card has-a rank (2, 3, 4, …, jack, queen, king, ace)
    - we will represent the rank with a String
  - cards have a natural ordering (by rank; we will assume aces are the highest rank)
  - impose a class-invariant
    - the suit and rank are always valid for a constructed card



CardUtil

- we will find it useful to create a utility class that defines the legal ranks and suits of cards

```
package playingcard;

import java.util.HashMap;
import java.util.ArrayList;

final class CardUtil {

  static final HashMap<String, Integer> RANKS =
                                          createRanks();
  static final ArrayList<String> SUITS = createSuits();
```

no access modifier means that CardUtil is "package private"; only classes inside the package playingcard can see CardUtil

CardUtil

```
static HashMap<String, Integer> createRanks()
{
  HashMap<String, Integer> r =
                          new HashMap<String, Integer>();
  r.put("two", 2);
  r.put("three", 3);
  r.put("four", 4);
  // and so on ...
  r.put("queen", 12);
  r.put("king", 13);
  r.put("ace", 14);
  return r;
}
```

CardUtil

```
static ArrayList<String> createSuits()
{
  ArrayList<String> s = new ArrayList<String>();
  s.add("club");
  s.add("diamond");
  s.add("heart");
  s.add("spade");
  return s;
}
```

Card

```
package playingcard;

public class Card implements Comparable<Card>
{
  private final String rank;
  private final String suit;

  public Card(String rank, String suit)
  {
    checkRank(rank);
    checkSuit(suit);
    this.rank = rank;
    this.suit = suit;
  }
```

Remember the class invariant: the rank and suit must always be valid for a constructed card; we have to validate the rank and suit arguments.

Card

```
private void checkRank(String rank)
{
  if(CardUtil.RANKS.get(rank) == null)
  {
    throw new IllegalArgumentException(rank + " is not a
                                        valid Card rank.");
  }
}
```

Remember that CardUtil.RANKS is a map where the keys are Strings like "two", "ace", and "seven". CardUtil.RANKS holds all of the valid ranks as keys. If the key given by the String rank is not in the map then it must be an invalid rank.

Card

```
private void checkSuit(String suit)
{
  if(!CardUtil.SUITS.contains(suit))
  {
    throw new IllegalArgumentException(suit + " is not a
                                        valid Card suit.");
  }
}
```

Remember that CardUtil.SUITS is a list holding all of the valid suits. If the suit given by the String suit is not in CardUtil.SUITS then it must be an invalid suit.

Card

```
public String getRank()
{
  return this.rank;
}

public String getSuit()
{
  return this.suit;
}
```

Card

```
@Override public boolean equals(Object obj)
{
  boolean eq = false;
  if(this == obj)
  {
    eq = true;
  }
  else if(obj != null && this.getClass() == obj.getClass())
  {
    Card other = (Card) obj;
    eq = this.getRank().equals(other.getRank()) &&
         this.getSuit().equals(other.getSuit());
  }
  return eq;
}
```

Remember the pattern for implementing equals is always the same:

1. check if this and obj are the same object
2. check if obj is not null AND
3. this and obj are instances of the same class
4. check if the corresponding attributes are equals

Remember to watch out for the case where the attributes can be null; this complicates step 4.

Card

```
@Override public int hashCode()
{
  return this.getRank().hashCode() +
         this.getSuit().hashCode();
}

@Override public String toString()
{
  StringBuffer result = new StringBuffer();
  result.append(this.getRank());
  result.append(" of ");
  result.append(this.getSuit());
  result.append("s");
  return result.toString();
}
```

Remember that because we overrode equals we need to override hashCode.

Card

```
@Override public int compareTo(Card other)
{
  String myRank = this.getRank();
  String otherRank = other.getRank();
  Integer myValue = CardUtil.RANKS.get(myRank);
  Integer otherValue = CardUtil.RANKS.get(otherRank);

  return myValue.intValue() - otherValue.intValue();
}
```

Remember that if we choose to implement Comparable, we have to implement compareTo. compareTo must return:

1. a negative int if the value of this is less than the value of other
2. zero if the value of this is the same as the value of other
3. a positive int if the value of this is greater than the value of other