# Chapter 4

# Implementing Aggregation and Composition

## 4.1 Implementing Aggregation

### 4.1.1 What is Aggregation?

Aggregation is a relation. It is also known as the *has-a* relation. It is a relation on classes. The relation captures that one class is part of another class. For example, assume that the class `Student` represents a student. For each student we want to keep track of the student's homepage. We decide to represent the student's homepage as an instance of the class `URL`, which is part of the package `java.net`. In this case, the classes `Student` and `URL` are related by the aggregation relation. We say that `Student` has a `URL`, and `URL` is called a component of `Student`.

The API of the `Student` class can be found at this link. The header of the three-parameter constructor

```
public Student(String id, String name, URL homepage)
```

reflects that the client has to create two instances of the `String` class and one instance of the `URL` class in order to create an instance of the `Student` class. In the next code fragment, the client creates an instance of the `URL` class, creates an instance of the `Student` class from two `String` literals and the instance of the `URL` class, and finally prints the `Student` object.

```
1 URL homepage = new URL("http://www.cse.yorku.ca/~cse81234/");
2 Student student = new Student("123456789", "Jane Smith", homepage);
3 output.println(student);
```

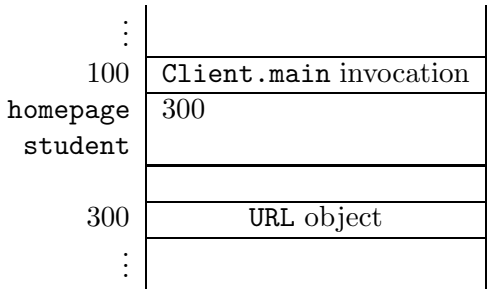
The complete client code can be found at this link.

### 4.1.2 UML and Memory Diagrams

The aggregation relation can be reflected in UML class diagrams. For example, the fact that `Student` has a `URL` can be depicted as follows.

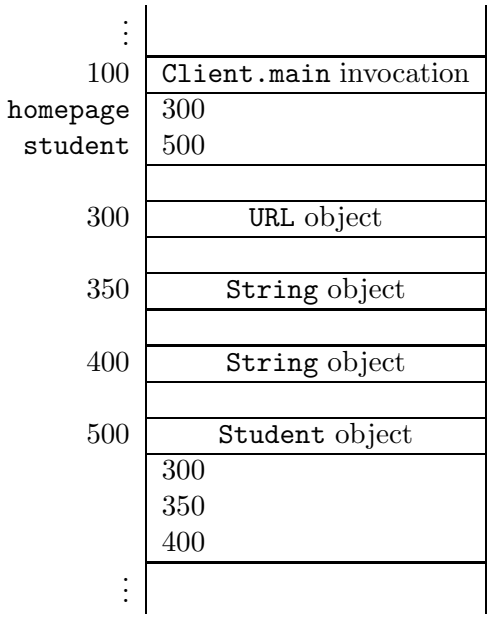

The number 1 in the above class diagram is called the multiplicity of the aggregation. It represents the number of `URL` objects that each `Student` object has: each `Student` has one `URL`.

Memory diagrams can also be utilized to reason about aggregation. For example, consider the code fragment introduced in Section 4.1.1. Once the execution reaches the end of line 1, memory can be depicted as follows.




The above diagram reflects that a `URL` object has been created at address 300 and that the local variable `homepage` refers to that object.

Once the execution reaches the end of line 2, the memory content has changed and can now be depicted as follows.




The diagram now also contains a `Student` object at address 500 and the local variable `student` refers to the `Student` object. Note also that the `Student` object refers to the two `String` objects and the `URL` object as it contains the addresses of these objects. We have not included yet the names of the attributes which refer to these objects. These attributes will be discussed in the next section.

### 4.1.3 The Attributes Section

As an implementer, we have to decide on the number of attributes and for each attribute we have to choose its type and its name. Usually, we look at the constructors and also the accessors and mutators in the API for inspiration. In the API of the `Student` class we find a three-parameter constructor with the following header.

```
public Student(String id, String name, URL homepage)
```

In the API, we also encounter the accessors `getId`, `getName` and `getHomepage` and the mutator `setHomepage`. These suggest the following three attributes.

```
1 private String id;
2 private String name;
3 private URL homepage;
```

Note, however, that choosing the attributes is in general not as simple as picking the parameters of the constructor with the most parameters. Although the above attributes are a natural choice, other choices are possible. We will come back to this in Section 4.1.6.

We will implement the constructors and methods in such a way that the following class invariant is maintained.

```
1 /* class invariant: this.id is a 9-digit string and this.name != null */
```

As we will see, maintaining this class invariant will allow us to simplify our code.

### 4.1.4 The Constructors Section

In the constructors, we initialize the attributes `id`, `name` and `homepage`. According to the API, the `Student` class has three constructors. One of them takes two arguments (`id` and `name`), another has three parameters (`id`, `name` and `homepage`), and the third one is a copy constructor (and, hence, takes a single argument of type `Student`). The two-parameter constructor can be used if the student does not have a homepage. To avoid code duplication, we will utilize constructor chaining as discussed in Section 2.3.1. In this case, the two-parameter constructor and the copy constructor both delegate to the three-parameter constructor. To reflect the fact that a student does not have a homepage, we initialize the attribute `homepage` to `null`. Hence, the body of the two-parameter constructor may be implemented as follows.

```
1 this(id, name, null);
```

The body of the copy constructor can be implemented as follows.

```
1 this(student.getId(), student.getName(), student.getHomepage());
```

To avoid code duplication, we use the accessors. These accessors will be discussed in the next section.

In the three-parameter constructor, we utilize the mutators to initialize the attributes. As we have seen in Section 2.3.1, this avoids code duplication. Hence, the body of the three-parameter constructor may be implemented as follows.

```
this.setId(id);
this.setName(name);
this.setHomepage(homepage);
```

According to the API, the three-parameter and the two-parameter constructor throw an `IllegalArgumentExcep` if the ID provided by the client is not valid. The parameter `id`, which is of type `String`, is valid if

- it is not `null`,
- it consists of nine digits.

Rather than checking it in the constructor, we have decided to delegate the validation to the mutator `setId`.

The fact that those constructors throw an `IllegalArgumentException` if the ID provided by the client is not valid is reflected in the header of the constructor (and, of course, also in the documentation comments). For example, the header of the two-parameter constructor is as follows.

```
public Student(String id, String name) throws IllegalArgumentException
```

One may wonder why the copy constructor does not throw an `IllegalArgumentException`.[1] Intuitively, since the argument, a `Student` object, was created successfully, its ID is valid. Formally, we can infer it from the class invariant.

We leave it to the reader to check that the class invariant holds at the end of each constructor invocation.

### 4.1.5 The Methods Section

Next, we discuss the (public and private) methods of the `Student` class.

#### Accessors

For each attribute, we introduce a corresponding accessor. Since all follow the pattern that we introduced in Section 2.2.4, we only present the accessor for the attribute `homepage`.

```
public URL getHomepage()
{
   return this.homepage;
}
```

Note that the return type of the above accessor is non-primitive, i.e. it returns an object of type `URL`.

[1] Since `IllegalArgumentException` is a runtime exception, we do not need to catch or specify it.

### Mutators

Since we do not want to allow clients to change the name or the ID of a student, we simply make the mutators `setName` and `setId` private. As a consequence, these mutators do not appear in the API and the attributes `name` and `id` are immutable (by the client).

The mutator of the `name` attribute follows the pattern presented in Section 2.2.4.

```
1 private void setName(String name)
2 {
3    this.name = name;
4 }
```

Also the mutator for the `homepage` attribute follows the same pattern, but this mutator is public.

```
1 public void setHomepage(URL homepage)
2 {
3    this.homepage = homepage;
4 }
```

Recall that the mutator of the attribute `id` not only sets the value of the attribute, but also validates the passed argument. If the passed ID is invalid, the mutator throws an `IllegalArgumentException`. For an ID to be valid, it has to be different from `null` and consist of nine digits. To check the latter property, we can use the regular expression $\backslash d\{9\}$, which is captured by the string literal `"\\{d9}"`.[2] A string matches this pattern if and only if it consists of nine digits. Hence, we can implement the mutator `setId` as follows.

```
1  private void setId(String id) throws IllegalArgumentException
2  {
3     final String PATTERN = "\\d{9}";
4     if (id != null && id.matches(PATTERN))
5     {
6        this.id = id;
7     }
8     else
9     {
10       throw new IllegalArgumentException("Invalid ID");
11    }
12 }
```

### The `toString` Method

As we already mentioned in Section 2.2.4, most classes override the `equals` method of the `Object` class. Also our `Student` class overrides the `equals` method. The method returns a string representa-

[2] In string literals, the backslash character ('\') serves to introduce so-called escaped constructs. For example, the expression \t represents a tab. The expression \\ represents a single backslash.

tion of the `Student` object on which it is invoked. According to the API, this string representation, when printed on the screen, will result in one or two lines. The first line contains the student's ID and name, separated by a colon. The second line is only present if the student has a homepage. In that case, the second line consists of (a string representation of) the URL of the student's homepage. For example, for the `Student` object created in the code snippet in Section 4.1.1, the invocation of the `toString` method on this object results in the following.

```
123456789:Jane Smith
www.cse.yorku.ca/~cse81234/
```

The `toString` method can be implemented as follows.

```
1  public String toString()
2  {
3     StringBuffer result = new StringBuffer();
4     result.append(this.getId());
5     result.append(":");
6     result.append(this.getName());
7     if (this.getHomepage() != null)
8     {
9        result.append("\n");
10       result.append(this.getHomepage().toString());
11    }
12    return result.toString();
13 }
```

Rather than constructing several `String` objects,[3] we decided to use a `StringBuffer` object instead. Note that we use two invocations of the `toString` method in the body of the `toString` method. In line 10, we invoke the `toString` method on the `URL` object to obtain a string representation of the student's homepage. In line 12, we invoke the `toString` method on the `StringBuffer` object. Note that we use the accessors, rather than accessing the attributes directly. This avoids code duplication and makes it easier to change the representation of the attributes if necessary.

### The `equals` Method

Most classes also override the `equals` method of the `Object` class. According to the API of the `Student` class, two `Student` objects are considered equal if their IDs are equal. The `equals` method has the same general structure as the one in Section 2.2.4. Instead of comparing the width and the height of a rectangle, both being represented by attributes the type of which is primitive, we compare the IDs of the students, represented by an attribute of the non-primitive type `String`. As a consequence, we use `equals`, rather than `==`, to compare the IDs of the students.[4]

[3] Recall that the class `String` is immutable and, hence, concatenation does not change the object but returns a new object.

[4] Although the class `String` is immutable, it does not implement the "one instance per state" pattern and, hence, we should not use `==` to compare `String` objects. For example, the snippet

```
1  public boolean equals(Object object)
2  {
3     boolean equal;
4     if (object != null && this.getClass() == object.getClass())
5     {
6        Student other = (Student) object;
7        equal = this.getId().equals(other.getId());
8     }
9     else
10    {
11       equal = false;
12    }
13    return equal;
14 }
```

From the class invariant we can conclude that the invocation `this.getId()` in line 7 does not return `null`. Hence, the invocation of `equals` in line 7 never throws a `NullPointerException`.

### The `hashCode` Method

Since we did override the `equals` method, we had better override the `hashCode` method as well to satisfy the property

- if `x.equals(y)` returns true then `x.hashCode()` and `y.hashCode()` return the same integer.

Since students are considered equal if their IDs are the same, we can let the `hashCode` method return the ID (as an integer). Note that, according to the class invariant, the ID is a 9-digit string and, therefore, can be represented as an `int`. Hence, the `hashCode` method can be implemented as follows.

```
1  public int hashCode()
2  {
3     return Integer.parseInt(this.getId());
4  }
```

### The `compareTo` method

According to the API of the `Student` class, the class implements the `Comparable<Student>` interface. As a consequence, we have to implement the single method of that interface: the `compareTo`

---

```
String first = new String("123456789");
String second = new String("123456789");
```

creates two different `String` objects (and, hence, `first == second` returns false) which represent the same ID (`first.equals(second)` return true). Note that the class `Class` does implement the "one instance per state" pattern and, hence, we can use `==` to compare `Class` objects.

method. As we have already seen in Section 2.2.4, this method provides an ordering on objects which can be used, for example, for sorting. The ordering of students is based on an ordering of their IDs. The IDs are ordered lexicographically.[5] Hence, to compare two students we can simply get their IDs and delegate the comparison to the `String` class.

```
1 public int compareTo(Student student)
2 {
3    return this.getId().compareTo(student.getId());
4 }
```

Note that also in this case we use the accessors to get hold of the attributes.

Again we can infer from the class invariant that no `NullPointerException` is thrown in the body of the method.

The code of the entire `Student` class can be found by following this link.

### 4.1.6 Beyond the Basics

#### The Attributes Revisited

One may wonder whether we need an attribute of type `URL` in the class `Student`. Although it is a natural choice to represent the student's homepage by means of the an instance of the `URL` class, this information can be represented in several other ways as well. For example, we can represent it as a `String` object. To refactor our `Student` class such that the student's homepage is represented by an instance of the `String` class, we make the following three changes. First of all, we replace the declaration of the `homepage` attribute with

```
1 private String homepage;
```

We add to the class invariant[6]

```
1 this.homepage is a well-formed URL
```

Secondly, we modify the accessor of the attribute as follows.

```
1 public URL getHomepage()
2 {
```

[5] According the API of the `String` class, if two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let $k$ be the smallest such index; then the string whose character at position $k$ has the smaller value (recall that characters can be compared using the `<` relational operator) lexicographically precedes the other string. In this case, the `compareTo` method of the `String` class returns the difference of the two character values at position $k$ in the two string, i.e.

```
this.charAt(k) - other.charAt(k)
```

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, the `compareTo` method of the `String` class returns the difference of the lengths of the strings, i.e.

```
this.length() - other.length()
```

[6] The string is well-formed if it specifies a legal protocol. Examples of legal protocols include http and ftp.

```
3     URL homepage;
4     if (this.homepage == null)
5     {
6        homepage = null;
7     }
8     else
9     {
10       try
11       {
12          homepage = new URL(this.homepage);
13       }
14       catch (MalformedURLException e)
15       {
16          homepage = null;
17       }
18    }
19    return homepage;
```

The API of the `URL` class tells us that the constructor `URL(String)` throws a `MalformedURLException` if the given string specifies an unknown protocol. Since a `MalformedURLException` is a checked exception, the exception has to be either specified or caught. Because the accessor `getHomepage` does not specify the exception in the API, we have to catch the exception. Note, however, that we can infer from the class invariant that `new URL(this.homepage)` never throws a `MalformedURLException`.[7]

Finally, we change the method `setHomepage` to the following mutator.

```
1  public void setHomepage(URL homepage)
2  {
3     if (homepage == null)
4     {
5        this.homepage = null;
6     }
7     else
8     {
9        this.homepage = homepage.toString();
10    }
11 }
```

Note that the new accessor and the mutator have to handle the case that the student has no homepage as a special case. This complicates the code. Since we generally strive to keep our code as simple as possible, we prefer to represent the homepage as a `URL` object, rather than a `String` object.

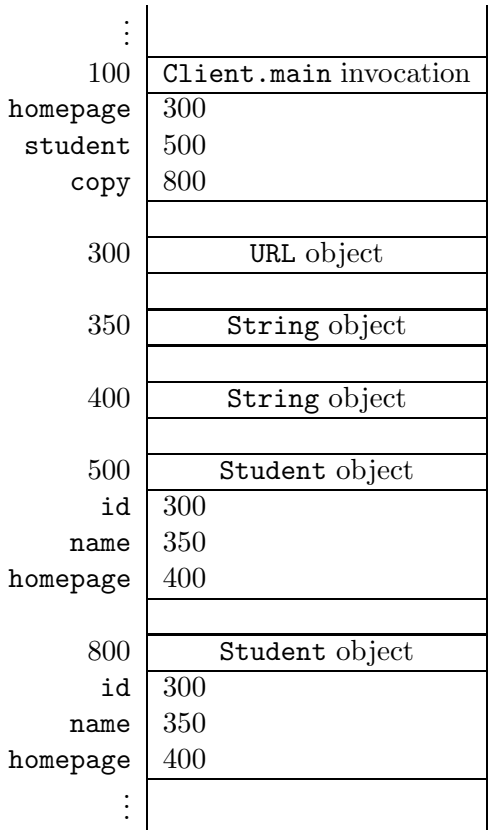
[7] The compiler, however, does not reach this conclusion since it is not aware of the class invariant.

### The Copy Constructor Revisited

Consider the following fragment of client code that uses the copy constructor.

```
1 URL homepage = new URL("http://www.cse.yorku.ca/~cse81234/");
2 Student student = new Student("123456789", "Jane Smith", url);
3 Student copy = new Student(student);
```

Once we reach the end of line 3, memory can be depicted as follows.




Note that both `Student` objects refer to the same `String` and `URL` objects. If we were to change the state of either one of the `String` objects or the `URL` object, both `Student` object would change. However, the `String` class is immutable and, hence, we cannot change the state of `String` objects. Since the API of the `URL` class contains no public mutators, we cannot change the state of `URL` objects either.

## 4.2 Implementing Composition

### 4.2.1 What is Composition?

Composition is a special type of aggregation. Hence, it is also a relation on classes. The composition relation not only captures that one class is part of another class. An instance of the latter class

can be said to "own" an instance of the former class. As an example, we will modify our `Student` class. For each student, we also keep track of the student's date of joining the university. Let us represent this date by an instance of the `Date` class, which is part of the `java.util` package.

The API of this modification of the `Student` class can be found at this link. To create an instance of the `Student` class, the client also has to provide an instance of the `Date` class. In the next code snippet, the client creates an instance of the `URL` class, an instance of the `Date` class, and an instance of the `Student` class.

```
1 URL homepage = new URL("http://www.cse.yorku.ca/~cse81234/");
2 Date now = new Date();
3 Student student = new Student("123456789", "Jane Smith", homepage, now);
```

A key difference between the class `Date` and the classes `URL` and `String` is that the class `Date` is mutable, whereas the other classes are not. For example, the client can get the student's date of joining the university and add one hour to it as follows.

```
4 Date date = student.getJoinDate();
5 final long HOUR_IN_MILLISECONDS = 60 * 60 * 1000;
6 date.setTime(date.getTime() + HOUR_IN_MILLISECONDS);
```

The method `getTime` returns the number of milliseconds from January 1, 1970, 00:00:00 GMT until the date represented by the `Date` object on which it is invoked.

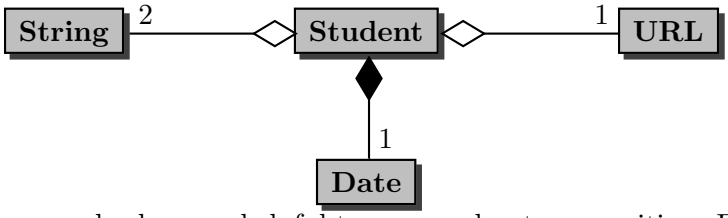
Recall that the classes `Student` and `Date` form a composition, i.e. a `Student` object "owns" its `Date` object. This means that the `Student` object has exclusive access to its `Date` object. Others, including the client, cannot modify its `Date` object.

The accessor `getJoinDate` cannot return a reference to the `Date` object of the `Student` object. Towards a contradiction, assume that the accessor `getJoinDate` does return a reference to the `Date` object of the `Student` object. In that case, the above code snippet mutates the `Date` object "owned" by the `Student`. But this contradicts that `Student` and `Date` form a composition.

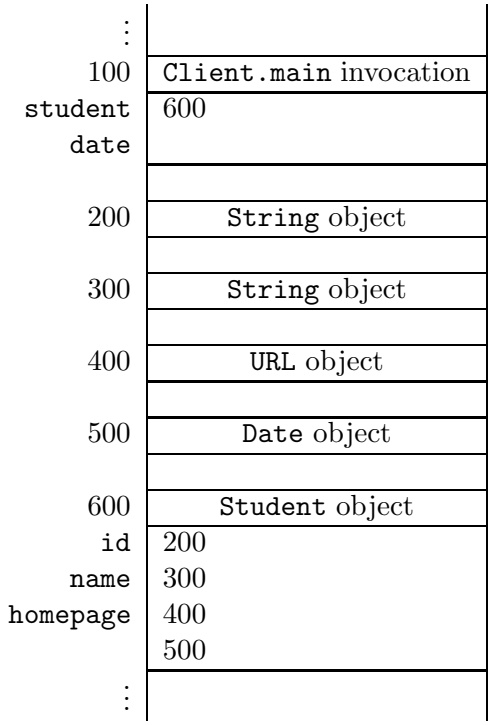
Rather than returning a reference to the `Date` object of the `Student` object, the mutator `getJoinDate` returns the reference to a copy of that `Date` object. Hence, the above code snippet mutates the copy of the `Date` object, not the `Date` object "owned" by the `Student` object.

### 4.2.2 UML and Memory Diagrams

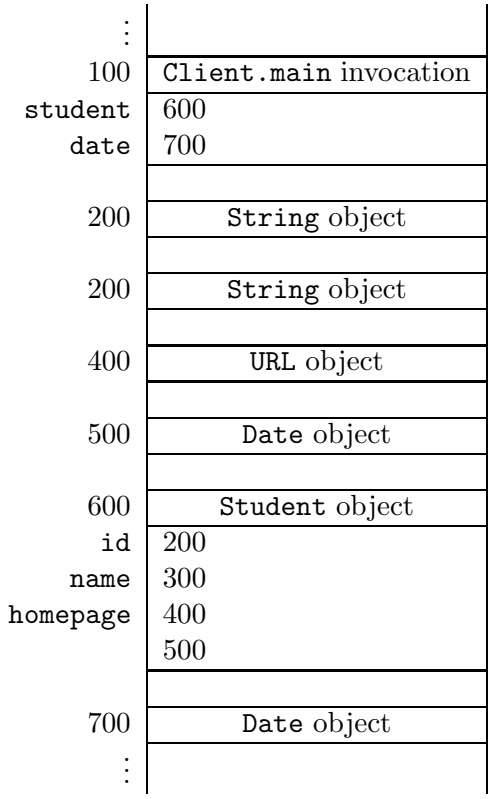
Also the composition relation can be reflected in UML class diagrams. For example, the fact that the classes `Student` and `Date` form a composition can be depicted as follows.




Memory diagrams can also be very helpful to reason about composition. For example, consider the code fragment introduced in Section 4.2.1. Once the execution reaches the end of line 3, memory can be depicted as follows.




Once we reach the end of line 4, the above memory diagram has changed into the one below.



Note that the local variable `date` refers to the new instance of the `Date` class at address 700, and *not* to the instance of the `Date` class at address 500. The latter `Date` object is "owned" by the `Student`. As a consequence, in line 6 the `Date` object "owned" by the `Student` object is *not* modified. Instead, its copy located at address 700 is changed.

### 4.2.3 The Attributes Section

A natural choice to represent the additional information, the date at which the student joined the university, is an attribute of type `Date`. Hence, we add the following declaration to our `Student` class.

```
1 private Date joinDate;
```

We strengthen the class invariant by adding `this.joinDate != null`.

### 4.2.4 The Constructors Section

According to the API, the class `Student` contains five constructors. As in Section 4.1.4, we exploit constructor chaining to avoid code duplication. Here, we focus on the constructor to which all other constructors delegate: the four-parameter constructor with the following header.

```
public Student(String id, String name, URL homepage, Date joinDate)
```

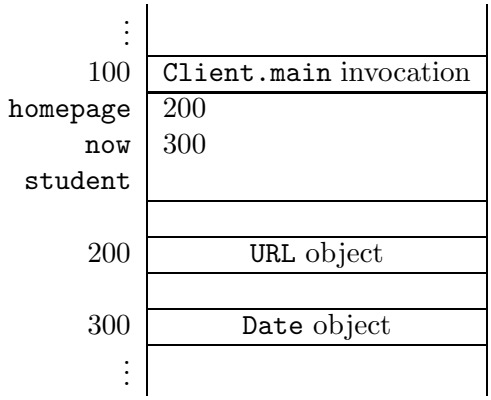
We focus on the fourth parameter. Before implementing the constructor, we first consider the following fragment of client code.

```
1 URL homepage = new URL("http://www.cse.yorku.ca/~cse81234/");
2 Date now = new Date();
3 Student student = new Student("123456789", "Jane Smith", homepage, now);
4 final long HOUR_IN_MILLISECONDS = 60 * 60 * 1000;
5 now.setTime(now.getTime() + HOUR_IN_MILLISECONDS);
```

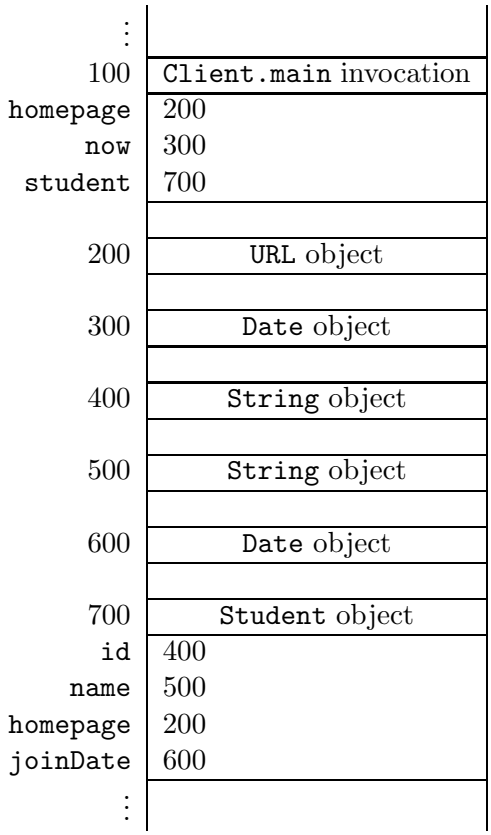
The above example shows that we cannot initialize the attribute `joinDate` to (the address of) the `Date` object (referred to by) `now`. Towards a contradiction, assume that the attribute `joinDate` *is* initialized to the `Date` object `now`. In that case, the client has access to the `Date` object "owned" by the `Student` object by means of its local variable `now`. Hence, the `Student` object does *not* have exclusive access to its `Date` object, which contradicts that the classes `Student` and `Date` form a composition.

Instead of initializing the attribute `joinDate` to the `Date` object passed as an argument, we initialize the attribute to a copy of the passed `Date` object. This copy is the `Date` object which is "owned" by the `Student` object. Note that the client does not have access to this copy.

Consider the above snippet of client code. Assume that the memory can be depicted as follows once we reach the end of line 2.




When we reach the end of line 3, memory can be depicted as follows.




Note that the `Student` object refers to a `Date` different from the one known to the client. Hence, in line 5 of the above code snippet the `Date` object at address 300 is modified, *not* the `Date` object at address 600 which is "owned" by the `Student` object.

Let us return to our attempt to implement the four-parameter constructor. From the above discussion we can conclude that we have to initialize the `joinDate` attribute to a copy of the passed `Date` object. Rather than doing the copying in the constructor, we delegate this to the mutator of the `joinDate` attribute. Hence, the four-parameter constructor can be implemented as follows.

```
1 public Student(String id, String name, URL homepage, Date joinDate) throws
      IllegalArgumentException
2 {
3    this.setId(id);
4    this.setName(name);
5    this.setHomepage(homepage);
6    this.setJoinDate(joinDate);
7 }
```

### 4.2.5 The Methods Section

Next, we only discuss the methods of the `Student` class which are either new or modified.

#### Accessor

As we already discussed in Section 4.2.1, the accessor of the `joinDate` attribute has to return a copy of the `Date` object "owned" by the `Student` object. Unfortunately, the `Date` class does not have a copy constructor. However, it has a constructor that is very similar to a copy constructor. The constructor

```
Date(long time)
```

creates a `Date` object which represents the date which is the given number of milliseconds (captured by the parameter `time`) after "the epoch", namely January 1, 1970, 00:00:00 GMT. Hence, using this constructor in combination with the method `getTime`, which we already saw in Section 4.2.1, we can implement the accessor for the `joinDate` attribute as follows.

```
1 public Date getJoinDate()
2 {
3    return new Date(this.joinDate.getTime());
4 }
```

#### Mutator

Since the API of the `Student` class does not contain a mutator for the `joinDate` attribute, we make the mutator private. Recall that the four-parameter constructor delegates the copying of the argument of type `Date` to the mutator. We use the same technique as used in the accessor to copy the `Date` object.

```
1 private void setJoinDate(Date joinDate)
2 {
3    this.joinDate = new Date(joinDate.getTime());
4 }
```

**The `toString` Method**

Since the class contains some additional information, namely the date the student joined the university, we want to reflect this additional information in the string representation returned by the `toString` method. In particular, we add one line to the string representation. This line contains the date at which the student joined the university. This date is represented as

- the month, formatted as two digits with leading zeros as necessary, and
- the year, formatted as at least four digits with leading zeros as necessary,

separated by a '/'. To format the date properly, we exploit the `format` method of the `String` class as follows. The specifiers `%1$tm` and `%1$tY` represent the month and year, respectively.

```
1  public String toString()
2  {
3     StringBuffer result = new StringBuffer();
4     result.append(this.getId());
5     result.append(":");
6     result.append(this.getName());
7     if (this.getHomepage() != null)
8     {
9        result.append("\n");
10       result.append(this.getHomepage().toString());
11    }
12    result.append("\n");
13    result.append(String.format("%1$tm/%1$tY", this.getJoinDate()));
14    return result.toString();
15 }
```

The code of the entire `Student` class can be found by following this link.

## 4.3 Implementing Aggregation Using Collections

### 4.3.1 What is a Collection?

Assume we want to modify our `Student` class and keep track of two additional pieces of information. First of all, we want to record the student's gpa for each year of study (i.e., for first year, second year, third year and fourth year). Secondly, we want to keep track of the courses the student has taken. The API of the modified `Student` class can be found at this link.[8]

If a course is represented by an instance of the `Course` class,[9] then a `Student` may have zero, one of more `Courses`. The `Student` class has a collection of `Courses`. A collection is a special type of aggregation where the number of components may vary.

[8] To simplify the `Student` class a little, we do not keep track of the name, URL and join date.

[9] The API of the `Course` class can be found at this link.

In the following snippet of client code, we create a `Student` with ID `123456789`, we add two courses, and finally print all courses.

```
Student student = new Student("123456789");
final int FIRST = 1020;
final int SECOND = 1030;
student.addCourse(new Course(FIRST));
student.addCourse(new Course(SECOND));
Iterator<Course> iterator = student.iterator();
while (iterator.hasNext())
{
   output.println(iterator.next());
}
```

The complete client code can be found at this link.

### 4.3.2 The Attributes Section

The API of the `Student` class contains a single public attribute named `NUMBER_OF_YEARS`. As we can see in the API, this attribute is static and it is a constant of type `int`. In the API, we can also find its value, which is four. Hence, we introduce the following declaration and initialization.

```
public static final int NUMBER_OF_YEARS = 4;
```

As an implementer, we have to decide how to represent the additional information: the four gpa's and the collection of courses. Since the constructors of the modified class have the same headers[10] as the those of the original class, we look at the methods for inspiration.

Let us first have a look at all those methods that manipulate the gpa's.

```
public double getGpa(int year)
public void setGpa(int year, double gpa)
```

The method `getGpa` returns the student's gpa of the given `year`. For example, `student.getGpa(2)` returns the gpa of `student` for the second year. The `setGpa` method sets the gpa of the given `year` to the given `gpa`. For example, `student.setGpa(2, 8.15)` sets the gpa of the second year of `student` to 8.15.

Given the above method headers, we may decide to represent each gpa by a `double` or an instance of its wrapper class `Double`. To represent the four gpa's we can use, for example, the following four attributes.

```
private double first;
private double second;
private double third;
private double fourth;
```

[10] The headers are not exactly the same, since we do not consider the student's name, home page and join date.

However, this choice will lead to the use of conditionals in the implementation of the methods `getGpa` and `setGpa` which renders code that is error prone and not scalable. To avoid the use of conditionals (simplifying the implementation of `getGpa` and `setGpa`), we can represent the four gpa's as a collection. In particular, we can use either a `List`, a `Set` or a `Map`. Since the collection of gpa's may contain duplicates (the student may, for example, have the exact same gpa in first and second year), a `Set` is not appropriate for representing the collection. Let us represent the collection of gpa's as a `List` of `Double`s.[11] To the attribute section of the `Student` class we add the following declaration.

```
1 private List<Double> gpas;
```

Note that `List`is an interface. As we already discussed in Section 1.6.1, using interfaces may make our code more versatile.

Let us also have a look at all those methods that manipulate the collection of courses.

```
public void addCourse(Course course)
public Iterator<Course> iterator()
```

The `addCourse` method adds the given `course` to the collection of courses taken by the student. The `iterator` method returns an `Iterator` object. The latter object can be used to enumerate the courses taken by the student using the methods `next` and `hasNext`.

Each course is represented by an instance of the `Course` class. Since the collection of courses does not contain duplicates (we do not keep track of the number of times the student takes a course), a `Set` is more appropriate than a `List`. Hence, we also add the following declaration to the attribute section of the `Student` class.

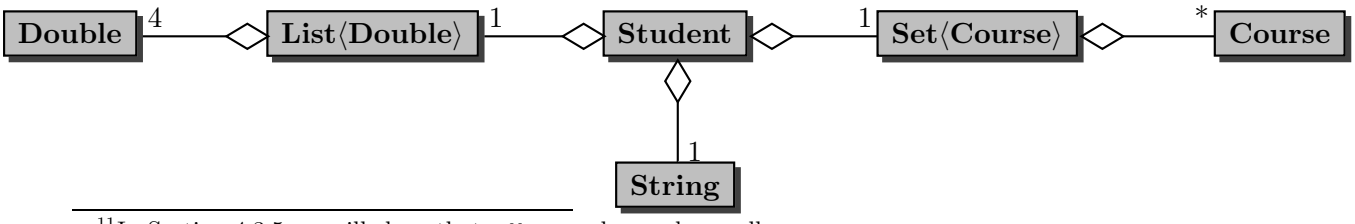
```
1 private Set<Course> courses;
```

Recall that there is a difference between `null`, which represents no object at all, and an object that represents an empty list or set. Since we want the attributes `gpas` and `courses` to always refer to a `List` object and a `Set` object, respectively, we strengthen the class invariant with the following.

```
1 this.gpas != null && this.courses != null
```

We leave it to the reader to verify that all public constructors establish this class invariant and that all public methods maintain this class invariant.

The classes discussed above are related as follows.




[11] In Section 4.3.5 we will show that a `Map` can be used as well.

The $*$ in the above class diagram indicates the multiplicity of the aggregation. It represents the number of `Course` objects that each `Student` object has: each `Student` has a variable number of `Courses`.

### 4.3.3 The Constructors Section

In the API of the `Student` class we find the following two constructor headers.

```
public Student(String id)
public Student(Student student)
```

The first constructor creates a `Student` with the given `id`, an empty collection of courses, and initializes all four gpa's to zero. The second one is a copy constructor. Neither the first one can delegate to the second one, nor the second one can delegate to the first one.

The one-parameter constructor has to initialize the attribute `id` to the argument of the constructor, the attribute `gpas` to a list of four zeroes, and the attribute `courses` to an empty set. Since `List` and `Set` are interfaces, they cannot be instantiated. Hence, we have to create instances of classes that implement these interfaces. The `List` interface is implemented by classes such as `ArrayList` and `LinkedList`, and the `Set` interface is implemented by classes such as `HashSet` and `TreeSet`. Let us pick `ArrayList` and `HashSet`. Then the one-parameter constructor can be implemented as follows.

```
1  public Student(String id)
2  {
3     this.setId(id);
4     List<Double> gpas = new ArrayList<Double>(Student.NUMBER_OF_YEARS);
5     for (int i = 0; i < Student.NUMBER_OF_YEARS; i++)
6     {
7        gpas.add(0.0);
8     }
9     this.setGpas(gpas);
10    this.setCourses(new HashSet<Course>());
11 }
```

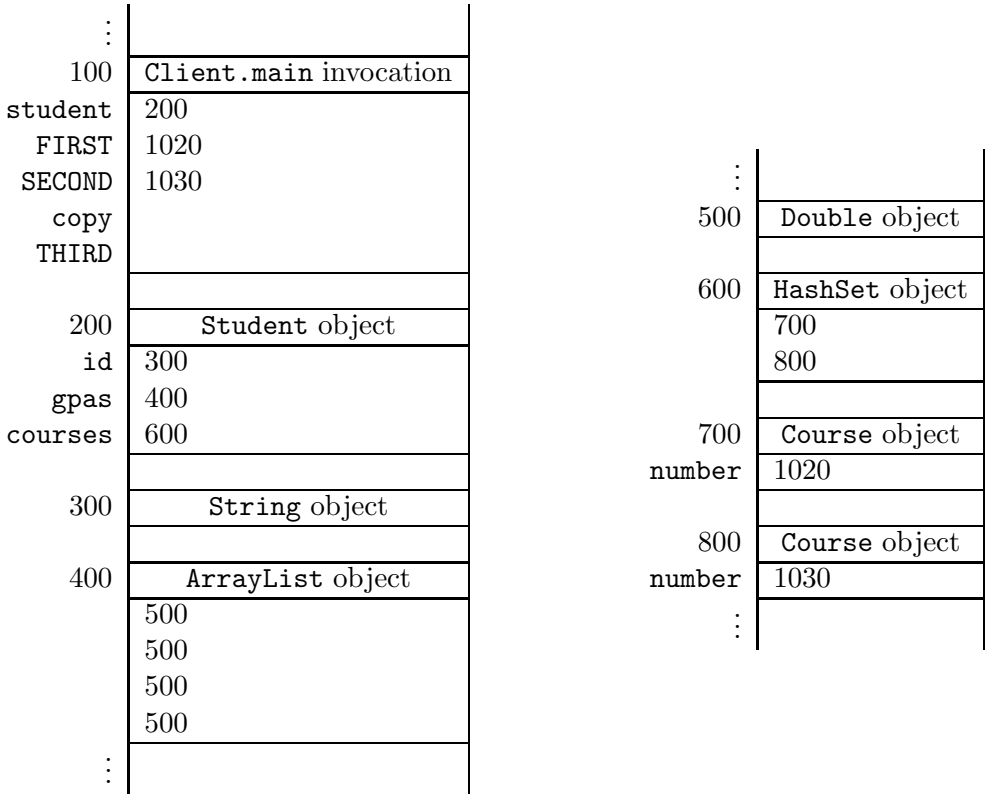
We create an empty list in line 4 and add four zeroes to that list in line 5–8. In line 10 we create an empty set.

The copy constructor can be implemented in several different ways. To compare the different implementations, we will consider the following snippet of client code.

```
1  Student student = new Student("123456789");
2  final int FIRST = 1020;
3  final int SECOND = 1030;
4  student.addCourse(new Course(FIRST));
5  student.addCourse(new Course(SECOND));
6  Student copy = new Student(student);
7  final int THIRD = 2011;
```

```
8 student.addCourse(new Course(THIRD));
```

After the execution has reached the end of line 5, memory can be depicted as follows.



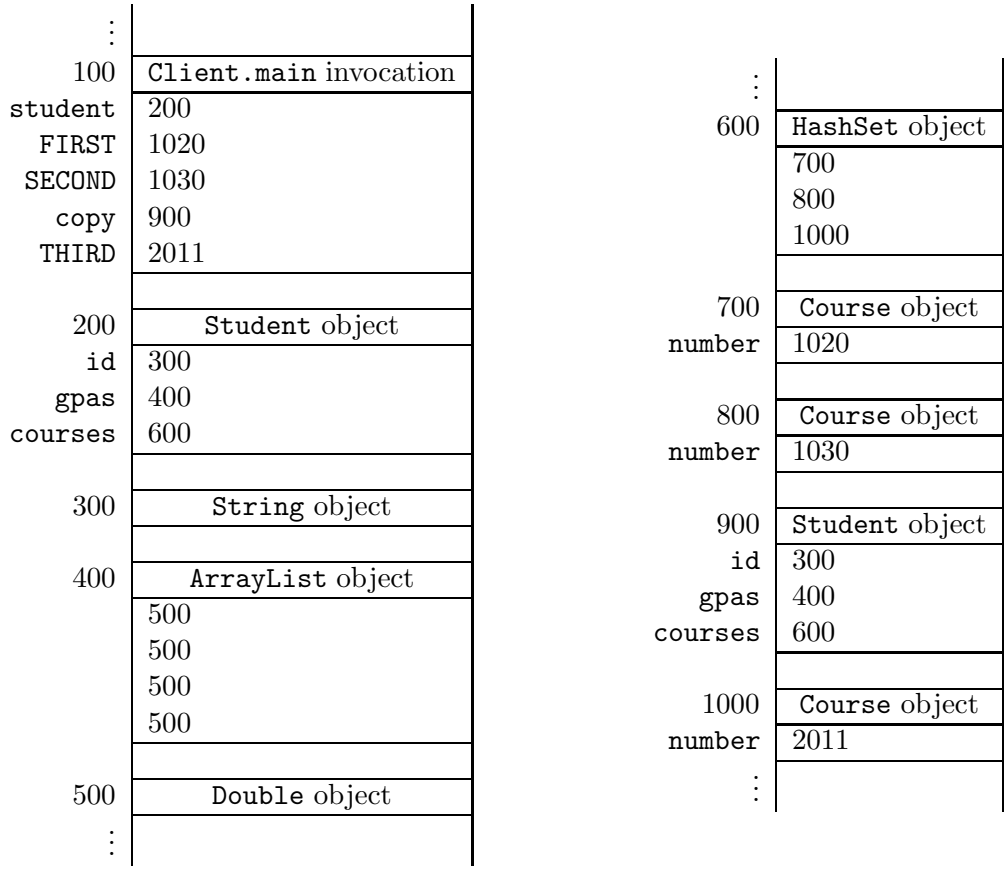

Since the class `Double` implements the pattern "one instance per state", the above memory diagram contains only a single `Double` object.

### Alias

We can implement the copy constructor in such a way that the attribute `gpas` of `this` object and the attribute `gpas` of the `student` object are merely aliases, that is, they both refer to the same object. The attribute `courses` can be initialized similarly. This leads to the following implementation of the copy constructor.

```
1 public Student(Student student)
2 {
3    this.setId(student.getId());
4    this.setGpas(student.getGpas());
5    this.setCourses(student.getCourses());
6 }
```

Once the execution of the above snippet of client code reaches the end of line 8, memory can be depicted as follows.




Note that the values of the attributes `gpas` and `courses` of both `Student` objects are the same. As a consequence, both `student` and `copy` have three courses.

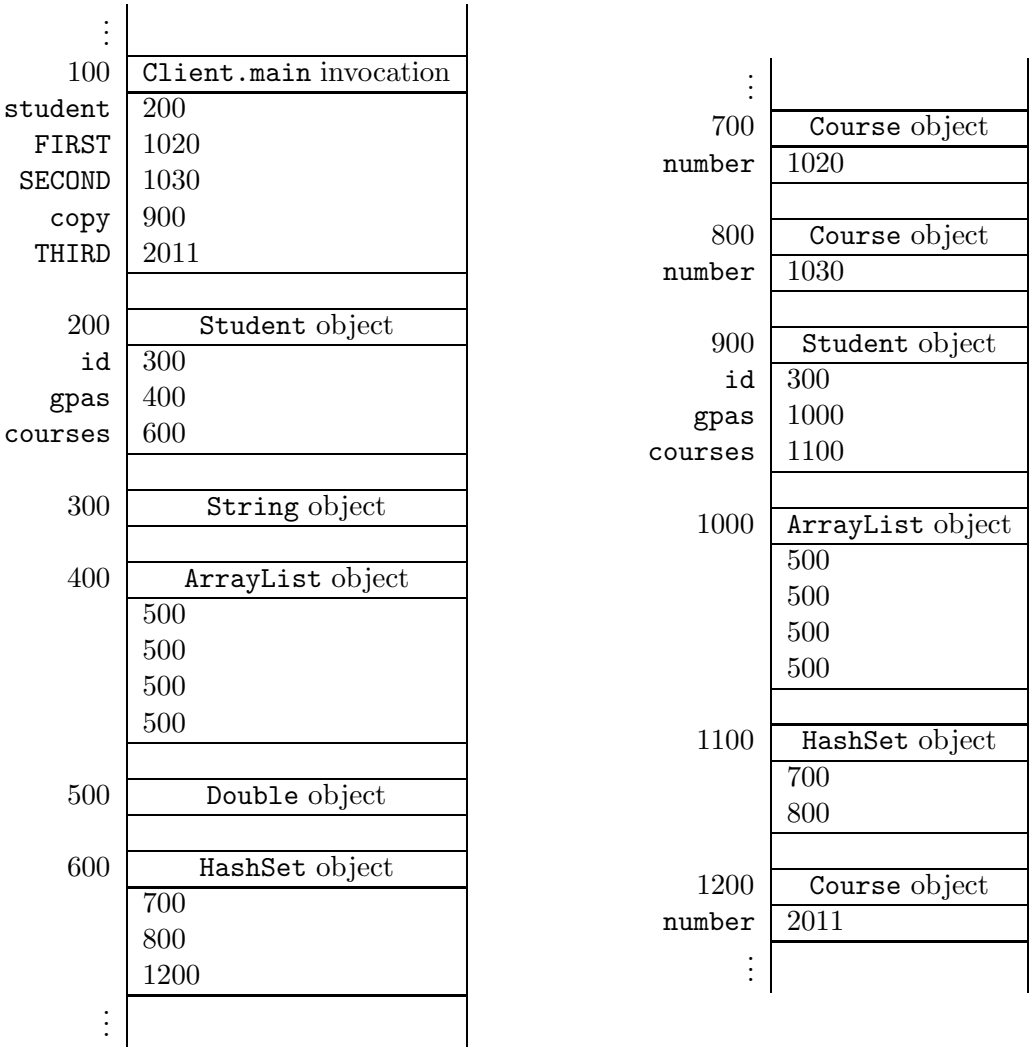
### Shallow Copy

As an alternative, we can implement the copy constructor in such a way that the attribute `gpas` of the `student` object is a shallow copy of the attribute `gpas` of `this` object. The attribute `courses` can be initialized similarly. This leads to the following implementation of the copy constructor.

```
1 public Student(Student student)
2 {
3    this.setId(student.getId());
4    this.setGpas(new ArrayList(student.getGpas()));
5    this.setCourses(new HashSet(student.getCourses()));
6 }
```

We use the copy constructors of the classes `ArrayList` and `HashSet`. From the class invariant we can deduce that neither `student.getGpas()` nor `student.getCourses()` returns null. As a consequence, neither copy constructor throws a `NullPointerException`.

Once the execution of the above snippet of client code reaches the end of line 8, memory can be depicted as follows.




This time the values of the attributes `gpas` and `courses` of both the `Student` objects are different. Observe also that `student` has three courses, but `copy` has only two. Finally, note that both `HashSet` objects contain the same `Course` objects at the addresses 700 and 800.

### Deep Copy

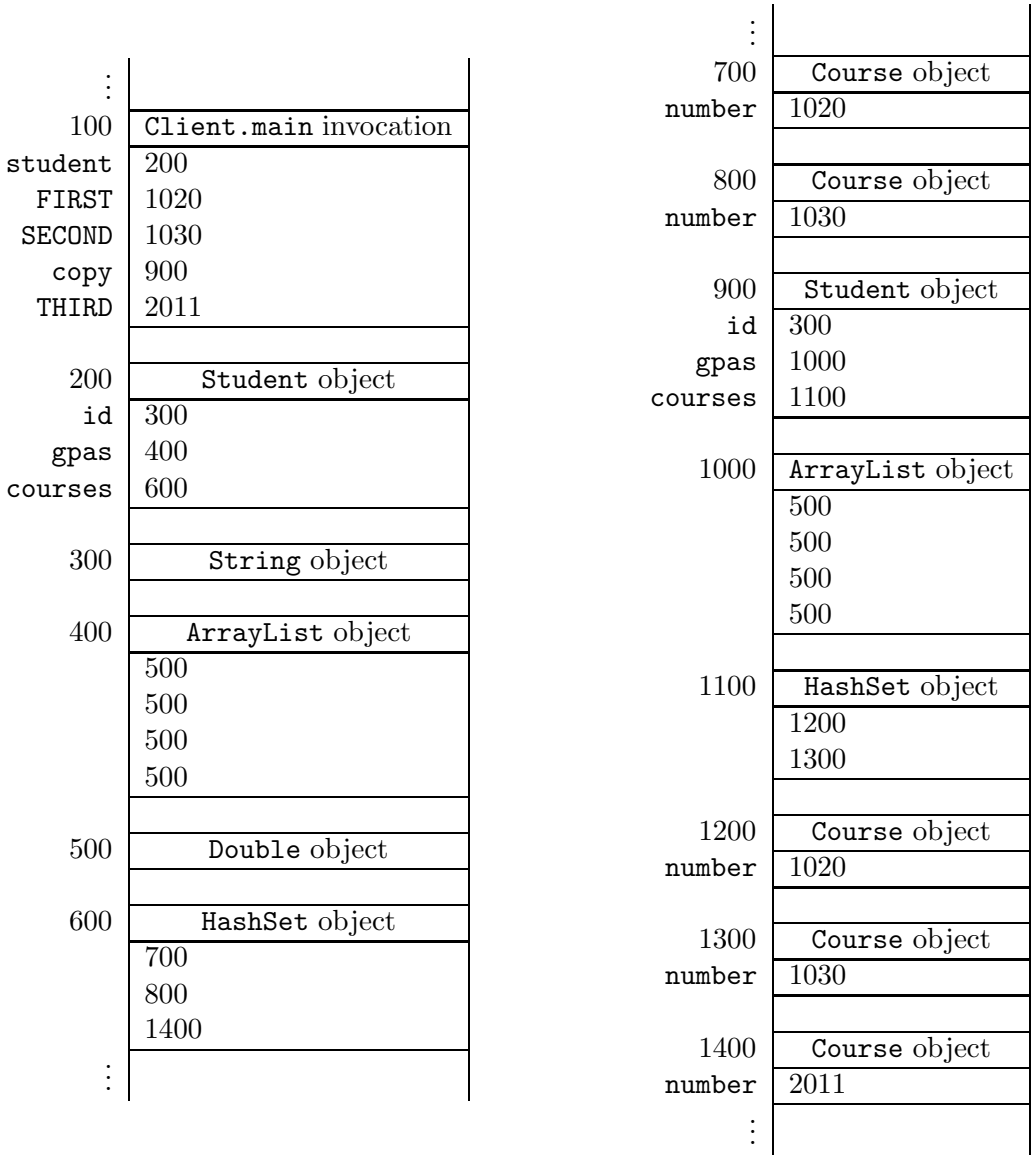
In our final implementation of the copy constructor, we ensure that the `courses` attribute of the `student` object is a deep copy of the `courses` attribute of `this` object. Since the class `Double` is immutable, it makes no sense to create a deep copy of the `gpas` attribute. Hence, we use a shallow copy instead.

```
1 public Student(Student student)
2 {
3    this.setId(student.getId());
4    this.setGpas(new ArrayList(student.getGpas()));
```

```
5     Set<Course> copy = new HashSet<Course>();
6     Iterator<Course> iterator = student.iterator();
7     while (iterator.hasNext())
8     {
9        copy.add(new Course(iterator.next()));
10    }
11    this.setCourses(copy);
12 }
```

In line 5, we create an empty collection. In line 6–10, we add a copy of each course, using the copy constructor of the `Course` class, to this collection.

Once the execution of the above snippet of client code reaches the end of line 8, memory can be depicted as follows.




This time, the `HashSet` objects contain different `Course` objects. If we were to execute also the following snippet

```
9  iterator = student.iterator();
10 Course course = iterator.next();
11 final int NEW = 1021;
12 course.setNumber(NEW);
```

only the `Course` object at address 700 would change to the following.

| 700 | Course object |
|---|---|
| number | 1021 |

### 4.3.4 The Methods Section

Here, we only discuss the methods which are new. That is, we consider the implementation of the following four methods.

```
public double getGpa(int year)
public void setGpa(int year, double gpa)
public void addCourse(Course course)
public Iterator<Course> iterator()
```

Recall that the gpa's are represented by a `List` of `Doubles`. Hence, returning the gpa of the second year amounts to returning the second element of the list. To retrieve this element from the list, we can simply delegate to the `get` method of the `List` interface. Since the index of the first element of a list is zero, the index of the gpa of the given `year` is `year` $-$ 1. Hence, the `getGpa` method can be implemented as follows.

```
1 public double getGpa(int year)
2 {
3    return this.getGpas().get(year - 1);
4 }
```

To implement the `setGpa` method, we delegate to the `set` method of the `List` interface as follows.

```
1 public void setGpa(int year, double gpa)
2 {
3    this.getGpas().set(year - 1, gpa);
4 }
```

The collection of courses is represented by a `Set` of `Courses`. Adding a given `course` to this collection can be implemented to simply delegating to the `add` method of the `Set` interface.

```
1 public void addCourse(Course course)
2 {
3    this.getCourses().add(course);
4 }
```

The `add` method of the `Set` interface returns a boolean. Note that we simply discard that returned boolean in the above method.

Finally, we implement the `iterator` method. This method returns an `Iterator` object that allows us to enumerate the courses. Again, we can simply delegate to the `Set` interface.

```
1 public Iterator<Course> iterator()
2 {
3    return this.getCourses().iterator();
4 }
```

Note that, when implementing the bodies of above methods, the implementer takes on the role of a client of the `List` and `Set` interfaces.

The interface `Iterable` contains the single method `iterator`. Since we have implemented the `iterator` method, our `Student` class implements `Iterable<Course>`. This is reflected in the header of the class `Student` as follows.

```
1 public class Student implements Comparable<Student>, Iterable<Course>
```

Implementing this interface allows a `Student` object to be the target of the "foreach" statement.[12] For example, line 6–10 of the client snippet presented in Section 4.3.1 can be replaced with

```
6 for (Course course : student)
7 {
8    output.println(course);
9 }
```

### 4.3.5 Beyond the Basics

#### Using a Map

Instead of a `List`, we can also use a `Map` to represent the four gpa's. The attribute maps years, represented by `Integer`s, to gpa's, represented by `Double`s. In our implementation we replace the declaration of a `List` with the following declaration of a `Map`.

```
1 private Map<Integer, Double> gpas;
```

To initialize this attribute, we replace line 4–9 of the one parameter constructor with

```
4 Map<Integer, Double> gpas = new HashMap<Integer, Double>();
5 for (int year = 1; year <= Student.NUMBER_OF_YEARS; year++)
6 {
7    gpas.put(year, 0.0);
8 }
9 this.setGpas(gpas);
```

Here, we only consider the copy constructor that makes a shallow copy. In that case, we replace line 4 with

```
4 this.setGpas(new HashMap<Integer, Double>(student.getGpas()));
```

In this setting the methods `getGpa` and `setGpa` can be implemented as

```
1 public double getGpa(int year)
2 {
3    return this.getGpas().get(year);
4 }
```

[12] This is something specific to Java.

and

```
1 public void setGpas(int year, double gpa)
2 {
3    this.getGpas().put(year, gpa);
4 }
```