13 ASSOCIATIONS

Terry Marris 5 May 2001

13.1 OBJECTIVES

By the end of this lesson the student should be able to

- implement associations between objects
- understand how objects communicate with each other

13.2 PRE-REQUISITES

The student should be comfortable with class and object diagrams, associations and attributes, constructors, setting, getting, toString() and main methods, parameters and arguments, the *this* operator and class (static) variables.

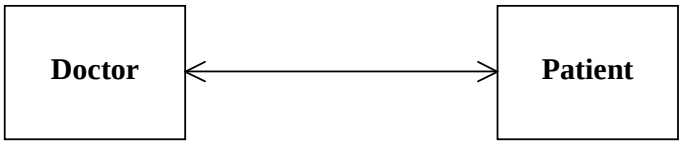
13.3 PREVIEW

We see that associations can be either uni-directional or bi-directional. We see that attributes implement associations.

13.4 NAVIGATION

We have already seen how to represent multiplicity, the numbers of objects involved in an association. We now see that associations have a direction.

An example of a bi-directional association is the association between a doctor (general practitioner) and his or her patients. A doctor can tell you who his or her patients are. A patient can tell you who his or her doctor is. We say that a doctor is *responsible* for knowing who his or her patients are. And that a patient is responsible for knowing who his or her doctor is. We show a bi-directional association with a double-headed arrow.



In a Java implementation, *Doctor* would have a field containing *Patient* objects and *Patient* would have a field referring to a *Doctor* object.

An example of a uni-directional association is the association between a patient and an observation (e.g. temperature) taken at a particular time. If we expect a patient to be responsible for knowing his or her observations, and if we do not expect an observation to be responsible for knowing which patient it belongs to, we have a uni-directional association. We show a uni-directional association with an arrow.

In a Java implementation, *Patient* would have a field containing *Observation* objects but *Observation* would not have a field referring to a *Patient*.

The direction in which an association takes place is known as navigability. If navigability arrows are not shown on a class diagram we assume that navigability has not been decided.

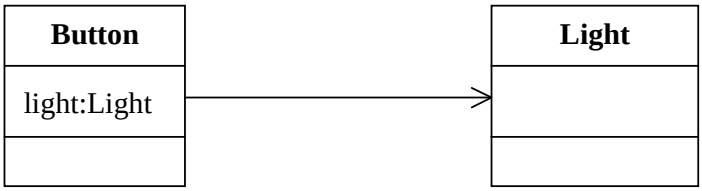
13.5 UNI-DIRECTIONAL ONE-TO-ONE

Perhaps the simplest example involves a button and a light object. The button acts as a toggle: you push the button and the light comes on; you push the button again and the light goes off.

A button controls one light. A light is controlled by one button.

We can imagine that a button sends messages to its light: if you are on, switch yourself off; if you are off, switch yourself on. We cannot imagine a situation where the light sends a message to its button. So we decide that the association is in one direction, from button to light.

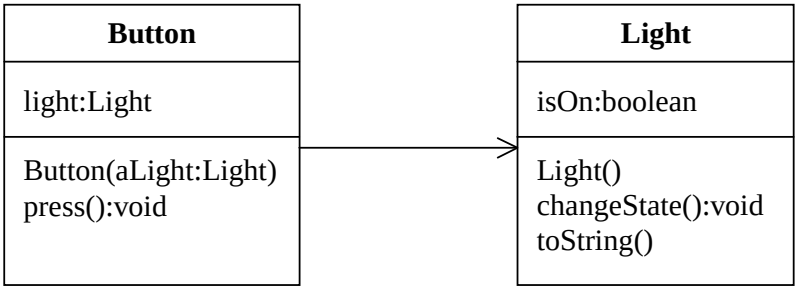
We use attributes to implement associations.



A button has a light.

A button's constructor is responsible for initialising its light.

A light's state, whether lit or not, is represented by the boolean attribute, *isOn*. A light can change its own state.



Let's look at some coding.

```
/* Button.java
   Terry Marris  5 May 2001
*/


public class Button {
  private Light light;              a button has a light

  public Button(Light aLight)
  {
    light = aLight;             its light is initialised with the given light, aLight
  }


  public void press()
  {
    light.changeState();      message is sent to light: go and change your state
  }
}
```

```
/* Light.java
   Terry Marris  5 May 2001
*/

public class Light {
  private boolean isOn;             represents this light's state

  public Light()
  {
    isOn = false;                   initially, this light is off
  }

  public void changeState()
  {
    if (isOn)                       if you are on
      isOn = false;                 switch yourself off
    else                            if you are off
      isOn = true;                  switch yourself on
  }

  public String toString()
  {
    if (isOn)
      return "on";
    else
      return "off";
  }

  public static void main(String[] s)
  {
    Light aLight = new Light();               create a new light
    Button aButton = new Button(aLight);      create a new button with
                                              the light
    System.out.println("Before pressing button, light is " +
                       aLight);
    aButton.press();                        button, go and press yourself
    System.out.println("After pressing button, light is " +
                       aLight);
    aButton.press();
    System.out.println("After pressing button again, " +
                      "light is " + aLight);
  }
}
```

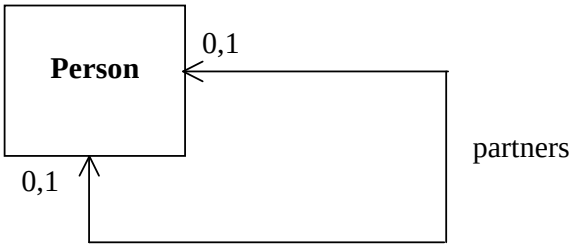
Program run:

```
Before pressing button, light is off
After pressing button, light is on
After pressing button again, light is off
```

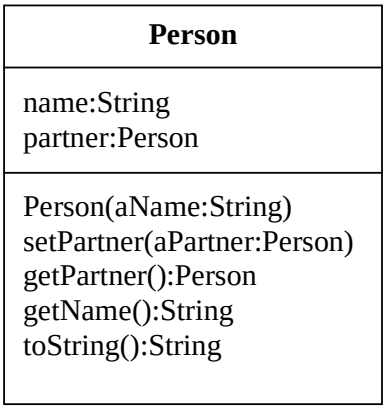
13.6 BI-DIRECTIONAL ONE TO ONE

Perhaps the most obvious one-to-one association is where two people are partners to each other. In the object diagram shown below, we see that fred's partner is wilma and that wilma's partner is fred.

We have just one class and a one-to-one association between two objects of the same class. We expect a person object to send messages to its partner and so the association is bi-directional.



We use an attribute *partner:Person* to implement the association.



As usual, we provide setting and getting methods for each attribute, including the attribute *partner:Person.*

Let's look at the coding.

The instance variables are straightforward.

```
public class Person {
  private String name;
  private Person partner;
```

The constructor initialises this name with the given name, and this partner to *null*. There is no partner for this person ... yet.

```
  public Person(String aName)
  {
    name = aName;
    partner = null;
  }
```

The *setPartner(Person)* method makes the given person this person's partner, and makes this person the given person's partner.

```
public void setPartner(Person aPerson)
{
  if (partner == null) {
    partner = aPerson;          make the given person this person's partner
    aPerson.setPartner(this);   make this person the given person's partner
  }
}
```

Two person objects are partners to each other at the same time; you cannot have it any other way.

[Technical note: *aPerson* is a reference to a *Person* object. So we can change the state of *aPerson* by sending it messages even though arguments are passed by value.]

The *getPartner()* and *getName()* methods are straightforward.

```
public Person getPartner()
{
  return partner;
}


public String getName()
{
  return name;
}
```

But the *toString()* method requires care in its implementation.

```
public String toString()
{
  return "Name: " + name + ", Partner: " +
          partner.getName();
}
```

If you wrote *return "Name: " + name + "Partner: " + partner;* you will have a run-time problem because *partner* contains a reference to this person that has a *partner* that contains a reference to this person that has a *partner* ... You have a set of circular references. Nasty.

Here is the coding complete with program run.

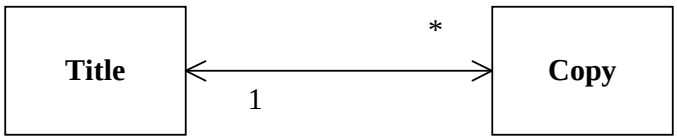
```
/* Person.java  Terry Marris  5 May 2001 */
public class Person {
  private String name;
  private Person partner;

  public Person(String aName)
  {
    name = aName;
    partner = null;
  }

  public void setPartner(Person aPerson)
  {
    if (partner == null) {
      partner = aPerson;
      aPerson.setPartner(this);
    }
  }

  public Person getPartner()
  {
    return partner;
  }

  public String getName()
  {
    return name;
  }

  public String toString()
  {
    return "Name: " + name + ", Partner: " +
            partner.getName();
  }

  public static void main(String[] s)
  {
    Person fred = new Person("Fred");
    Person wilma = new Person("Wilma");

    fred.setPartner(wilma);    fred, set your partner to wilma.  fred is also
                               responsible for setting wilma's partner to fred
    System.out.println(fred);
    System.out.println(wilma);
  }
}
```

Program run:

```
Name: Fred, Partner: Wilma
Name: Wilma, Partner: Fred
```

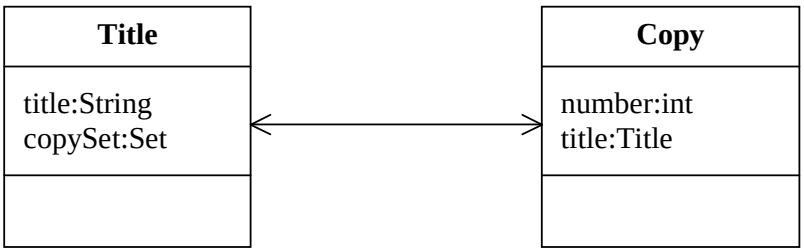
13.7 ONE-TO-MANY

Librarians refer to books as titles. There may be several copies of the title Core Java. A copy has just one title.

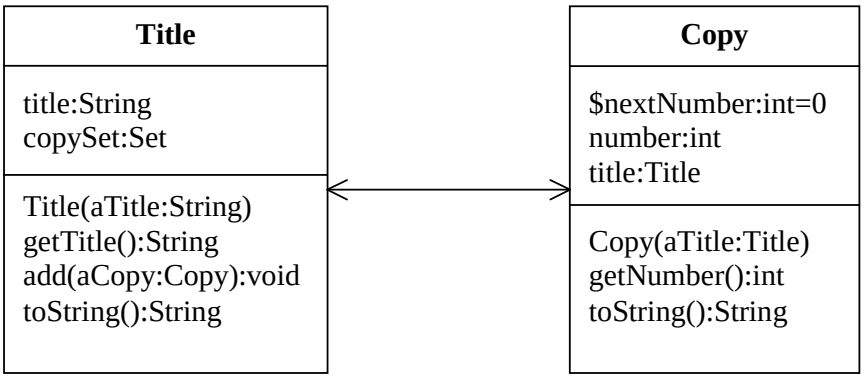


We expect a title to know what copies it has and a copy to know what its title is. So we make the association bi-directional.

We implement the association with a pair of attributes. A title has a set of copy objects. A copy has a title.



When we create a new copy, we want it to add itself to its title's set of copies. We shall make the *Copy(Title)* constructor responsible for making the link in both directions.



Here is the coding.

```
/* Title.java
   Terry Marris  5 May 2001
*/

import java.util.*;


public class Title {
  private String title;
  private Set copySet;              copySet holds this title's copies

  public Title(String aTitle)
  {
    title = aTitle;
    copySet = new HashSet();        initially, there are no copies for this title
  }


  public String getTitle()
  {
    return title;
  }


  public void add(Copy aCopy)       adds the given copy to the set of copies
  {
    copySet.add(aCopy);
  }


  public String toString()
  {
    String s = title + " has " + copySet.size() + " copies\n";
    Iterator it = copySet.iterator();
    while (it.hasNext()) {
      Copy aCopy = (Copy)it.next();
      s += "Number: " + aCopy.getNumber() + "\n";
    }
    return s;
  }
}
```

toString() returns a string something like

```
Core Java has 3 copies
Number: 1
Number: 2
Number: 3
```

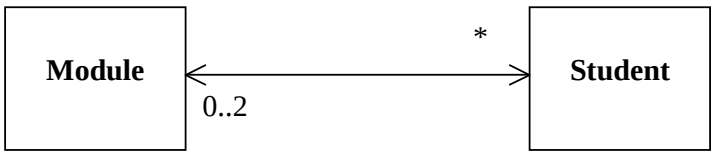
```
/* Copy.java
   Terry Marris  5 May 2001
*/

public class Copy {
  private static int nextNumber = 0;
  private int number;
  private Title title;

  public Copy(Title aTitle)
  {
    nextNumber++;
    number = nextNumber;            each copy has a unique #
    title = aTitle;
    title.add(this);                adds this copy to its title
  }


  public int getNumber()
  {
    return number;
  }


  public String toString()
  {
    return "Copy number " + getNumber() +
           " has title " title.getTitle();
  }


  public static void main(String[] s)
  {
    Title coreJava = new Title("Core Java");
    Copy aCopy = new Copy(coreJava);
    aCopy = new Copy(coreJava);
    aCopy = new Copy(coreJava);
    System.out.println(aCopy);
    System.out.println();
    System.out.println(coreJava);
  }
}
```

Program run:

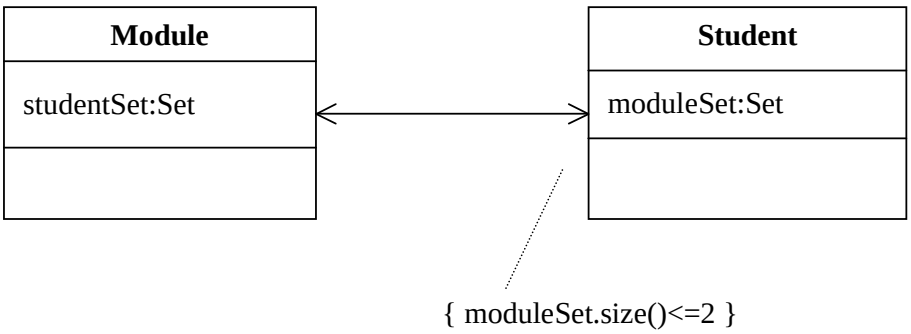
```
Copy number 3 has title Core Java

Core Java has 3 copies
Number: 3
Number: 1                  sets do not store their elements in any particular order
Number: 2
```

13.8 MANY-TO-MANY

A student may register for up to two modules. A module may have any number of students. A student knows the modules it has registered for. A module knows who its students are.



We implement the multiplicity with sets.



moduleSet contains the modules a student has registered for. *studentSet* contains the students registered for a module.

The restriction { moduleSet.size()<=2 } specifies that the number of modules stored in *moduleSet* cannot exceed two.

We choose to make the Student class responsible for maintaining the link in both directions. When a student registers for a module, not only does the student add the module to its set of modules, it sends a message to the module: add me to your set of students.

Here is the coding.

```
/* Module.java
   Terry Marris  5 May 2001
*/

import java.util.*;

public class Module {
  private String title;
  private Set studentSet;           contains this modules students

  public Module(String aTitle)
  {
    title = aTitle;
    studentSet = new HashSet();     initially, there are no students on this
  }                                 module


  public void addStudent(Student aStudent)
  {
    studentSet.add(aStudent);       adds the given student to the set of
  }                                 students


  public String getTitle()
  {
    return title;
  }


  public HashSet getStudentSet()
  {
    return studentSet;
  }
}
```

getStudentSet() returns the entire set of students for this module. The set is returned with an iterator and so the client can deal with the set as they wish.

```
/* Student.java
   Terry Marris  5 may 2001
*/

import java.util.*;


public class Student {
  private static int maxModule = 2;
  private String name;
  private Set moduleSet;            contains this students modules


  public Student(String aName)
  {
    name = aName;
    moduleSet = new HashSet();      initially, there are no modules
  }


  public void registerForModule(Module aModule)
  {
    if (moduleSet.size() < maxModule) {
      moduleSet.add(aModule);   adds the given module to this students modules
      aModule.addStudent(this);adds this student to the given module's students
    }
  }


  public String getName()
  {
    return name;
  }


  public HashSet getModuleSet()
  {
    return moduleSet;
  }
```

```
  public static void main(String[] s)
  {
    Module ot = new Module("Object Technology with Java");
    Module zed = new Module("Maths For Programmers");
    Module ct = new Module("Catastrophe Theory");

    Student tom = new Student("tom");
    tom.registerForModule(ct);
    tom.registerForModule(zed);
    tom.registerForModule(ot);

    Student terri = new Student("terri");
    terri.registerForModule(zed);

    Student zak = new Student("zak");
    zak.registerForModule(zed);

    HashSet tomsModuleSet = tom.getModuleSet();
    System.out.print(tom.getName() + " has registered for " +
                     tomsModuleSet.size() + " modules: ");
    Iterator it = tomsModuleSet.iterator();
    while (it.hasNext()) {
      Module aModule = (Module)it.next();
      System.out.print(aModule.getTitle());
      if (it.hasNext())
        System.out.print(", ");
    }
    System.out.println();

    HashSet zedStudentSet = zed.getStudentSet();
    System.out.print(zed.getTitle() + " has " +
                     zedStudentSet.size() + " students: ");
    it = zedStudentSet.iterator();
    while (it.hasNext()) {
      Student aStudent = (Student)it.next();
      System.out.print(aStudent.getName());
      if (it.hasNext())
        System.out.print(", ");
    }
    System.out.println();
  }
}
```

Program run

```
tom has registered for 2 modules: Maths For Programmers,
Catastrophe Theory
Maths For Programmers has 3 students: tom, zak, terri
```

13.9 FURTHER READING

FOWLER & SCOTT *UML Distilled* pp 56, 155
PARSONS *Object Oriented Programming with C++* pp 133
ERIKSON & PENKER *UML Toolkit* pp77
HORSTMANN & CORNELL *Core Java 2 Volume 1* pp 128

13.10 REVIEW

We use attributes to implement associations.

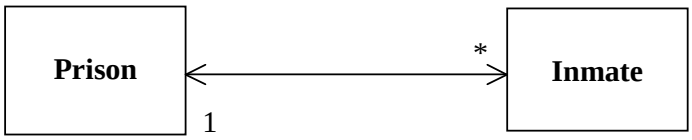
We begin thinking about navigability when we progress from class diagrams to Java implementations.

We decide whether to allow navigability in both directions. If so, we decide which class is to be responsible for keeping the two-way links up to date.

13.11 EXERCISES

1 In the *Person* class described in §15.6 implement and test the method *separate()* that sets both partners at the same time to *null*.

2 A prison knows who its inmates are. An inmate knows which prison he or she is in. An inmate has a number and an offence. A prison has a name. Implement the association between the *Prison* and *Inmate* classes shown below.



3 In the Module-Student scenario described in §15.8 provide a *withdrawFromModule(Module)* method in the *Student* class.

4 A ship may, or may not, have a captain. A captain may or may not have a ship. A ship has a name and a tonnage. A captain has a name. Draw a class diagram to model the scenario. Implement Ship and Captain classes and the association between them.

5 A person may or may not occupy a residence. A residence may be empty or occupied by any number of residents. An occupant knows their residence. A residence knows who its occupants are. A resident has a name. A residence has an address. Draw a class diagram to model the scenario. Implement Resident and Residence classes and the association between them.

6 A secret service knows who its agents are and an agent knows the secret services he or she works for. An agent has a name, a number and may, or may not, be licensed to score. A secret service has a name. Draw a class diagram to model the scenario. Implement SecretService and Agent classes and the association between them.

7 Buttons on a panel control a gearbox. The allowable states of the gearbox are reverse, neutral, first, second and third. The control panel has four buttons. Reverse selects reverse gear. Neutral selects neutral. The change up button selects the next higher gear. So, starting at neutral, pressing the change up button three times leaves the gearbox in third gear. If the gearbox is already in third gear, the change up button has no effect. The change down button selects the next lower gear. If the gearbox has first gear selected, the change down button has no affect. Draw a class diagram to represent the association between gearbox and its control panel. Implement the control panel and gearbox classes and the association between them.

8 A mobile phone may or may not have a number assigned to it. A telephone number may or may not have a telephone assigned to it. Draw a class diagram to represent the association between a mobile phone and a telephone number. Implement the association.