UCL

COMP1008 Associations , Static, and Overloading

UCL

Outline

- Quick review of relationship between class associations and references.
- The dreaded static.
- Overloading.

UCL

Classes and Program Structure

- A program consists of a collection of classes.
- Those classes define the abstract structure of the program in terms of the relationships or *associations* between the classes.
- When the program is run, the associations are realised by object references.

Representing Associations

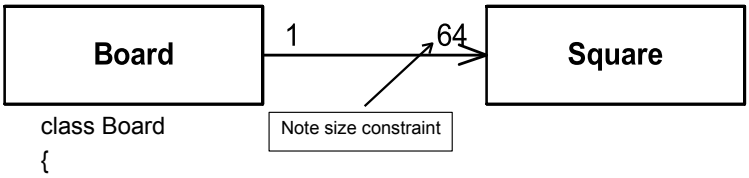
- An association between two classes means that an object of one class has a reference to an object of another class.


```
class Country
{
        private City capital;
        ...
}
```

In UML an association is bi-directional by default.
In the implementation, we usually want a uni-directional reference only. This can be made explicit in UML by adding an arrow head to one end of the association.

Representing Associations (2)

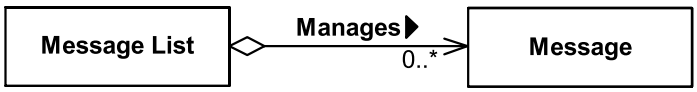
- The type used to represent the association needs to be determined correctly.




```
class Board
{
  private Square[] squares = new Square[64];
  // or
  // ArrayList<Square> squares = new ArrayList<Square>();
  ...
}
```

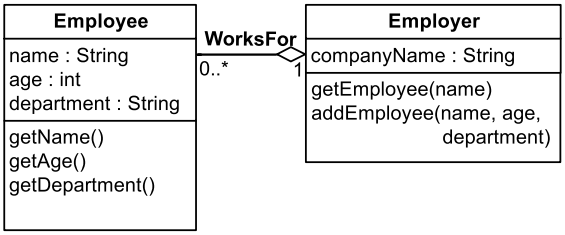

Representing Associations (3)

- Aggregation/Compositions are treated in the same way but we may need to be careful about sharing objects.




```
class MessageList
{
  private List< messages = new ArrayList();
 ...
}
```

Representing Associations (4)



- Need to decide if Employee has reference to Employer.
- If both have references to each other, then they mutually reference each other.
- This creates compilation and initialisation issues.
  - Which is compiled first, how is an Employee object initialised?
- Mutual references often indicate a design problem. Avoid if possible.

Static

- Why are some methods and variables declared as static?
- It depends on whether variables or methods “belong” to the class or to instance objects of the class.

Static (2)

- Non-static variables are instance variables.
  - Each object gets its own independent copy of each variable.
- Static variables are class variables.
  - A *single* copy of each variable exists and can be accessed by any other method in the class.

```
class Test
{
    private int instanceVar;
    private static int classVar;
}
```

Example

- Count number of times a method is called for all instance objects of a class.

```
private static int count = 0;
public void f()
{
  count++;
  // Rest of method…
}
```


final

- Public static variables are often used to create symbolic constants.
  - E.g., Math.PI (static variable PI in class Math)
- Such variables are additionally declared final:
  - public *static* final double PI = 3.141;
- The value of a final variable cannot be changed by assignment.

Static (3)

- Non-static methods are instance methods.
  - An instance method *must* be called for an object of the class.
  - x.method(args);
  - or method(args) if called on the same object.
- Static methods are class methods.
  - Not called on an object.
    - Can still write obj.staticmethod().
  - Can be called by any method declared by the class, or any method at all if public.
  - Cannot access instance variables (no object).

Example – Singleton

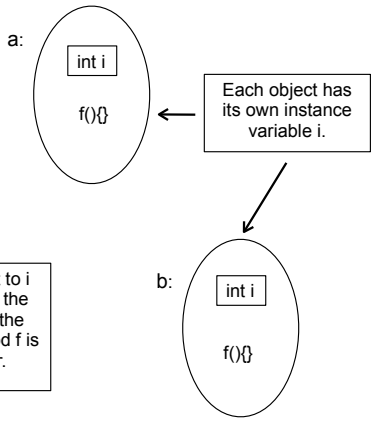
```
class MyClass {
  private static MyClass instance;
  private MyClass() { … }
  public static MyClass getInstance() {
    if (instance == null) { instance = new MyClass(); }
    return instance;
  }
  // Rest of class
}
```

Allow a single instance object only to be created.

Static (4)

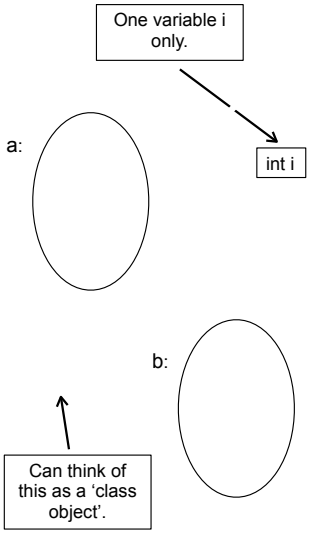
```
public class X
{
    private int i;
    public void f(){ i = 10; }
}

X a = new X();
X b = new X();
a.f();
b.f();
```



Static (5)

```
public class X
{
    private static int i;
    public static void f(){}
}
X a = new X();
X b = new X();
a.f();
b.f();
X.f();
```



Questions

Initialisation

- We have seen that constructors can be used to initialise instance variables.
- Both class and instance variables can also be directly initialised by initialisation expressions.
- private *int x = 2*;

Initialisation (2)

- And also by an initialiser block
  - Declared in a class outside of any methods.

```
private Stack x;
{ x = new Stack(); x.push(1); x.push(2);}
```

- A static initialiser block can be used for static variables.

```
private static Stack x;
static { x = new Stack(); x.push(1); x.push(2);}
```

Choosing

- 3 ways to initialise - how do you choose?
- No single answer but:
  - aim to initialise a variable as close to the point of declaration as possible.
  - or group all initialisation into the constructor, so it is all in the same place.

More than one constructor

- A class can have more than one constructor.
- Each can be used to initialise objects in a specific way.
- But won't all the constructors have the same name?
- Yes.

Overloading

- Two or more methods or constructors can have the same name.
- But must have different arguments.
  - String()
  - String(byte[])
  - String(char[])
  - String(String)
  - String(byte[], int)

Overloading (2)

- Return types are not considered:
  - int f(int)
  - float f(int) // Error
  - int f(int,int) // OK
  - float f(int, float) // OK
  - int f() // OK
- The compiler determines which method to call by matching the argument types.

this

- this is special variable that is automatically declared in an instance method.
- It is a reference to the object the method was called for.
- Allows you to refer directly to the current object.

this (2)

```
class T
{
    private Thing t;
    public int f(int x)
    {
        t.doSomething(this);
    }
}
```

Pass a reference to current object to another method called on a different object.

this (3)

```
class T
{
   private int x;
   public int f(int x)
   {
     this.x = x;
   }
}
```

Don't forget this idiom.

this (4)

- Can also be used to call a different overloaded constructor:

```
// This constructor does the real work
T(int x, int y, String z) { ... // Do the initialisation}

T() // Supply default values
{
  this(0,0,"Hello");
}
```

Avoids duplicating initialisation code in another constructor.

Only allowed in another constructor of the same class.

Summary

- Looked at various details of the construction and use of classes.
- Overloading is a new variety of abstraction.
- Lots of details for the programmer to know about and use carefully.