UCL

COMP1008
Classes and References

UCL

Agenda

- Defining simple classes
- Instance variables and methods
- Objects
- Basic UML notation for classes and objects
- Object references

UCL

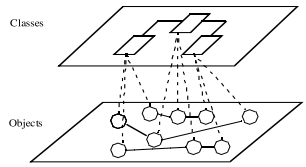
Reading

- You should be reading:
  - Part I chapters 6,9,10
- And browsing:
  - Part IV chapter 30

UCL

Classes & Objects

- We want to design programs in terms of classes and objects.
- Write the classes using the programming language.
  - Define structure and behaviour.
- Use the objects when the program runs.



UCL

Class

- Defines the structure and behaviour of an *instance object*.
  - A collection of *instance variables* to represent the state of the object.
  - A collection of *instance methods*, which can be called on the object.
- Acts as a template or blueprint.
- An object is an *instance* of one (and only one) class.
- A class may have many *instance objects*.

UCL

Abstraction

- A representation or model that includes the important, essential or distinguishing aspects of something while suppressing or ignoring less important, immaterial or diversionary details.
- Removing distinctions to emphasise commonality.
- Leaving out of consideration one or more properties of a complex object so as to attend to others.
- The process of formulating general concepts by abstracting common properties of instances.

Abstraction and Classes

- Classes represent abstractions.
- Abstraction is used to separate the essential from the full detail.
- They provide a selective and simplified view of the concepts being represented as objects.
- Good OO programming is all about identifying and using the right abstractions.

Class Role

- To Represent
  - Entities and things (Person, Account, Date, String, etc.)
  - Strategies and behaviours
  - Data structures
  - Relationships
- To Structure
  - Divide code into manageable chunks
  - Enforce encapsulation and information hiding

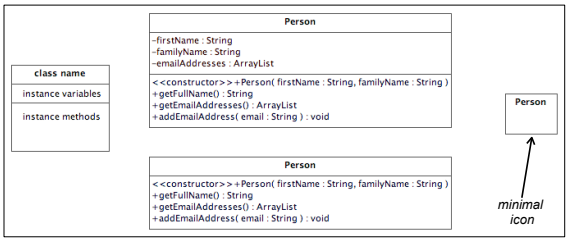
Notations

- Classes can be represented by:
  - Binary code
    - produced by the compiler (.class files)
    - not human readable
  - Source code
    - the programming language
  - Modelling language
    - formal: UML
    - informal: notes, diagrams, doodles
- Different levels of abstraction
  - Source code embodies all the details
  - Modelling language gives a more abstract view
    - Allows design to be represented without having all the detail of code

UML - The Unified Modeling Language

- De-facto standard notation
  - A small subset introduced in COMP1008, much more in COMP2010 next year.
  - Maintained by the OMG (Object Management Group), see www.uml.org
- UML provides a complete language for describing object-oriented models (like a programming language).
- Also provides a *visual notation* for displaying models.
  - This is what we are interested in here.

UML Class Icons

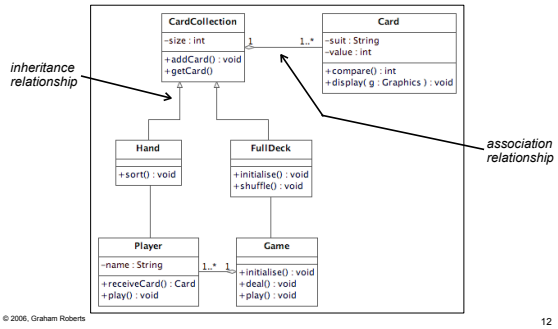


Instance variables are often referred to as attributes.

Amount of detail in icon can be selected depending on context.

UML Class Diagrams

- Show classes and their relationships.



Example Java Class

```
import java.util.ArrayList;
class Person {
  private String firstName;
  private String familyName;
  private ArrayList<String> emailAddresses;
  public Person(String firstName, String familyName) {
    this.firstName = firstName;
    this.familyName = familyName;
    this.emailAddresses = new ArrayList<String>();
  }
  public String getFullName() {
    return firstName + " " + familyName;
  }
  public void addEmailAddress(String address) {
    emailAddresses.add(address);
  }
    public ArrayList<String> getEmailAddresses() {
    return new ArrayList<String>(emailAddresses);
  }
}
```

Instance Variables

Constructor
(to initialise a new instance object)

Instance Methods

Class Person

- By convention a class name always starts with a capital letter.
- Instance objects can be created using:
  - Person person = new Person("Arthur","Dent") ;

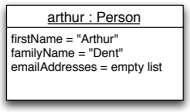
Create new Person object initialised to given name.

- As many objects as you need can be created.
  - Each object represents a distinct person.

A Person Object

- Has three private instance variables.
- And three public instance methods that can be called.
- And a constructor used to initialise the object.

UML Object Icon

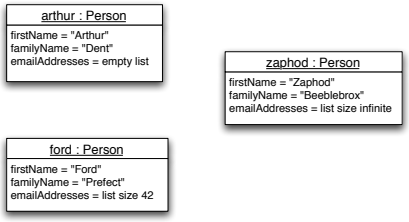


id : Class
used to label icon.
id can be omitted.

Class name underlined.
Can show instance variable values.

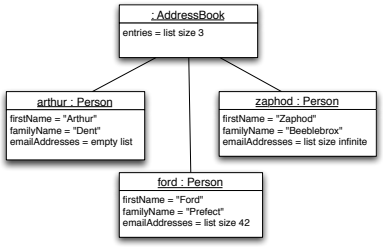
Objects

- Each object has its own *separate* set of variables.
- Each object has its own identity.
- Each object is independent.



UML Object Diagram

- Shows a snapshot of objects and links during execution of program



Private

- A class defines a scope.
- Declaring a method or variable private means that it can be accessed *only* within the scope of the class.
  - This means within a method body or an instance variable initialisation expression.
- At runtime objects implement the scope rules.
  - Class + compiler + type checking ensures behaviour must conform.
- The internal *state* of an object should be *private* and changed only by the object's methods.
  - The state is represented by the values of the instance variables.

Public

- Methods and variables declared public can be accessed by anything that has a reference to an object of the class.
  - Variables belong to an object.
  - Methods are called on an object.
- They form the public interface of objects of the class.
  - The services the object can perform.

Encapsulation

- Public and private are how encapsulation is enforced.
- Good practice states:
  - Instance variables should always be private.
  - Only a minimal number of methods should be made public.
  - Limit a name to the minimum scope.
- Why?
  - To enforce design decisions.
  - To protect against mistakes.
  - Information hiding.
  - To avoid misuse of classes and objects.
  - Experience demonstrates encapsulation is a very important design strategy.

"Encapsulation is a technique for minimising interdependencies among separately written modules by defining strict external interfaces. The external interface acts as a contract between a module and its clients. If clients only depend on the interface, modules can be re-implemented without affecting the client. Thus the effects of changes can be confined." Synder 1986

Questions?

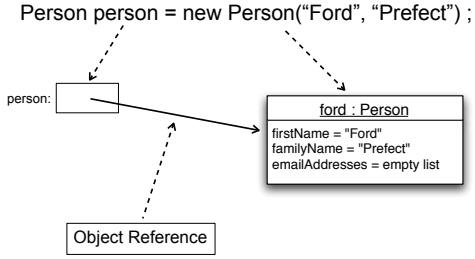
Type Person

- Person is a actually a *new type*.
- This allows Person to be used for declarations such as:

  Person person = new Person("Ford", "Prefect");
- In fact, Person is a *User Defined Type*.

Object References

Person person = new Person("Ford", "Prefect") ;

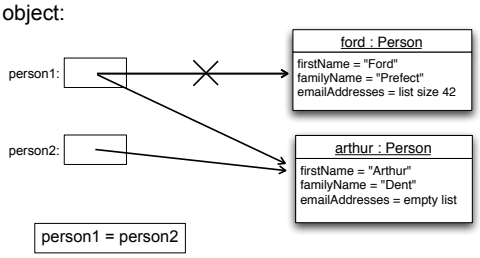


References

- A variable of a class type holds a *reference* to an object.
  - A reference is a pointer (form of memory address).
- Variable doesn't hold the object itself.
- The variable can go out of scope but the object can *still* exist (providing it is referenced by some other variable).
- One object can be referenced by several references and, hence, variables.

Class Type Assignment

- Assignment means storing a reference to a different object:



person1 = person2

So,

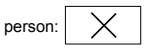
- When you declare a variable of class type, you get a container that holds an object reference.
- Assigning an "object to a variable" means storing a reference to the object in the container.

Null Reference

- null keyword.
- No object is referenced, so no methods can be called.

```
Person person = null;
```

- Default value if variable not initialised.



NullPointerException

- Caused by calling method on a null reference.
- No object, so no method can be called.
- If the error occurs find out why the variable is not referencing an object.

Questions?

Calling methods

- Given an object reference, a method can be called:
  Person person = new Person("Arthur", "Dent");
  String name = person.*getName()*;
- Only methods provided by class Person can be called.
  - Those methods declared by Person,
  - and inherited methods (as we will see later).

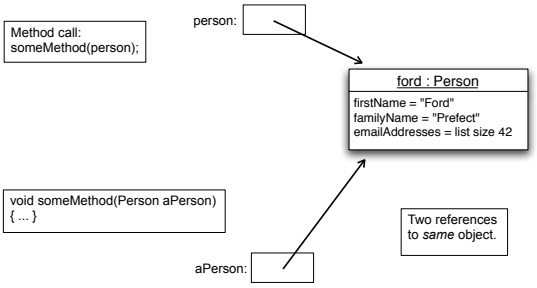
Object Reference Parameters

- You can pass an object reference as a parameter to a method:

```
void someMethod(Person aPerson)
{
   … // Use aPerson in method body
}
```

- A parameter variable is declared as normal.

Parameters & References



Object Parameters

- The parameter value is an *object reference*, not an object.
- The parameter variable is initialised to hold a copy of the reference.
- The object is *not copied*.
  - The reference is copied *not* the object.

Consequences

- If an object reference is passed as a parameter then:
  - Changing the object inside the method changes the object outside the method.
  - They are the same object!
- Don't forget that arrays are objects.
  - Changing elements in an array parameter changes the array outside the method.

Remember Primitive Types?

- Values of primitive types (int, char, long, boolean, etc.) are stored directly in variables using a binary representation.
- They are not objects.
- You can't have references to values of primitive types.
  - But there are classes to represent values of primitive types.
  - Class Integer, Double, Character, etc.

Primitive type parameters

- When a value of a primitive type is passed as a parameter, it is always *copied*.
- The parameter variable is initialised to a *copy* of the argument value.
- The argument value is computed in the method call:
  - obj.f(*a* + *b*);
- The value is used to initialise the parameter variable of the called method:

```
public void f(int n) { … }
```


Call-by-value

- The parameter passing mechanism used by Java is called “Call-by-value”.
- This means that the value of a parameter is always copied and a parameter variable initialised with the copy.
- Objects are not passed as parameters, only references to objects.
  - The reference is copied.

Don’t confuse references with a mechanism called pass-by-reference. Java does not support pass-by-reference.

“Variable is passed…”

- Beware, this means the *value* held in the variable is passed as a parameter:

```
int x = 10;
obj.f(x);
```

- The variable itself is not passed.
- The value of the variable is not changed by the method called.
  - But an object referenced by a variable of class type can change.

Return-by-value

- Returning a value from a method works in the same way as parameter passing.

```
public Person findPerson(String name)
{
    // find
    return aPerson;
}
```

- The value returned is a *copy* of the value computed in the return statement.

Object lifetime

- An object exists as long as it is accessible.
  - A reference to the object is available.

```
public Person makePerson()
{
    ... // Get person's name
    Person newPerson = new Person(firstName,familyName);
    ... // Add some email addresses
    return newPerson;
}
```

newPerson goes out of scope but object reference returned and object remains in existence.

- Use the method:

```
Person myPerson = makePerson();
```

- Lifetime of reference variable different from lifetime of object.

Object exists and is referenced by myPerson

Summary

- Seen how to construct a simple class.
- Methods and instance variables.
- Object references.
- References and parameter passing.
- Call-by-value, return-by-value.
- Object lifetime.