12 Data abstraction

- Packages and encapsulation
- Classes, subclasses, and inheritance

 © 2012 David A Watt, University of Glasgow

Packages

- A **package** (or **module**) is a named group of components declared for a common purpose.
- These components may be types, constants, variables, procedures, inner packages, etc. (depending on the PL).
- The meaning of a package is the set of bindings exported by the package
  - often called the package's **application program interface** (**API**).

University of Glasgow

Example: simple Python module

- Outline of a module (dictionary.py)

```
words = […]

def contains (word):
    global words
    return (word in words)

def add (word):
    global words
    if word not in words:
        words += [word]
```

- This module's API:

{ `words` → a list of words,
`contains` → a function that tests whether a word is in the list,
`add` → a procedure that adds a word to the list }

University of Glasgow

Encapsulation *(1)*

- Some of the components of a program-unit (package/class) may be private. This is called **encapsulation**.
- Levels of privacy:
  - A component is **private** if it is visible only inside the program-unit.
  - A component is **protected** if it is visible only inside the program-unit and certain “friendly” program-units.
  - A component is **public** if it is visible to application code outside the program-unit.
- A program-unit’s API consists of its public bindings only.

University of Glasgow

Encapsulation *(2)*

- Most PLs (such as Ada, Java, Haskell) allow individual components of a program-unit to be specified as private/protected/public.
- Python has a *convention* that components whose names start with "_" are private, whilst those whose names start with letters are public.
  - This convention is not enforced by the Python compiler.

University of Glasgow

Example: Python module with encapsulation

- Outline of a module (dictionary.py)

```
_words = […]

def contains (word):
    global _words
    return (word in _words)

def add (word):
    global _words
    if word not in _words:
        _words += [word]
```

- This module's API:

{ `contains` → a function that tests whether a word is in the list,
`add` → a procedure that adds a word to the list }

University of Glasgow

Example: Java packages

- In Java, the components of a package are classes and inner packages.
- Package components are added incrementally.
- Outline of a class declaration within a package:

```
package sprockets;

import widgets.*;

public class C {
    …
}
```

declares that class `C` is a component of package `sprockets`

declares that class `C` uses public components of package `widgets`

University of Glasgow

Classes *(1)*

- An **object** is a tagged tuple of variable components (**instance variables**), equipped with operations that access these instance variables.
- A **constructor** is an operation that initializes a newly created object.
- An **instance method** is an operation that inspects and/or updates an existing object of class *C*. That object (known as the **receiver object**) is determined by the method call.
- A **class** is a set of similar objects. All objects of a given class *C* have similar instance variables, and are equipped with the same operations.

University of Glasgow

Classes *(2)*

- A Java class declaration:
  - declares its instance variables
  - defines its constructors and instance methods
  - specifies whether each of these is private, protected, or public.
- A Java instance method call has the form "*O*`.`*M*`(...)`":
  - The expression *O* yields the receiver object.
  - *M* is the name of the instance method to be called.
  - The call executes the method body, with **`this`** bound to the receiver object.

University of Glasgow

Example: Java class *(1)*

- Class declaration:

```
class Dict {

    private int size;
    private String[] words;

    public Dict (int capacity)
    { … }

    public void add (String w)
    {   if (! this.contains(w))
            this.words[this.size++] = w; }

    public boolean contains (String w)
    { … }

}
```

Example: Java class *(2)*

- Possible application code:

```
Dict mainDict = new Dict (10000);
Dict userDict = new Dict (1000);
…
if (! mainDict.contains (currentWord)
  && ! userDict.contains (currentWord))
    userDict.add (currentWord);
```

- Illegal application code:

```
userDict.size = 0;
out.print (userDict.words[0]);
```

University of Glasgow

Subclasses

- If C' is a **subclass** of C (or C is a **superclass** of C'), then C' is a set of objects that are similar to one another but richer than the objects of class C:
  - An object of class C' has all the instance variables of an object of class C, but may have extra instance variables.
  - An object of class C' is equipped with all the instance methods of class C, but may override some of them, and may be equipped with extra instance methods.

University of Glasgow

Inheritance

- By default, a subclass **inherits** (shares) its superclass’s instance methods.
- Alternatively, a subclass may **override** some of its superclass’s instance methods, by providing more specialized versions of these methods.

Example: Java class and subclasses *(1)*

- Class declaration:

```
class Shape {
    protected float x, y;

    public Shape ()
    { x = 0.0; y = 0.0; }

    public final void move (
           float dx, float dy)
    { x += dx;  y += dy; }

    public void draw ()
    { … }   // draws a point at (x, y)
}
```

abbreviations for **this.**x and **this.**y

University of Glasgow

Example: Java class and subclasses *(2)*

- Subclass declaration:

```
class Circle extends Shape {
    private float r;
    public Circle (float radius)
    { x = 0.0;  y = 0.0;  r = radius; }
    public void draw ()
    { … }  // draws a circle centred at (x, y)
    public float diameter ()
    { return 2.0*r; }
}
```

Example: Java class and subclasses *(3)*

- Subclass declaration:

```
class Box extends Shape {

    private float w, h;

    public Box (…)
    { … }

    public void draw ()
    { … }   // draws a box centred at (x, y)

    public float width ()
    { return w; }

    public float height ()
    { return h; }

}
```

Example: Java class and subclasses *(4)*

- Possible application code:

```
Shape s = new Shape();
Circle c = new Circle(10.0);
s.move(12.0, 5.0);
c.move(3.0, 4.0);
… c.diameter() …

s.draw();       ---------- draws a point at (12, 5)
c.draw();       ---------- draws a circle centred at (3, 4)

s = c;
s.draw();       ---------- ditto! (dynamic dispatch)
```

University of Glasgow

Overriding

- Each instance method of a class *C* is inherited by the subclass *C′*, unless it is overridden by *C′*.
- The overriding method in class *C′* has the same name and type as the original method in class *C*.
- Most OO PLs allow the programmer to specify whether an instance method is **virtual** (may be overridden) or not:
  - In C++, an instance method specified as `virtual` may be overridden.
  - In Java, an instance method specified as `final` may *not* be overridden.

University of Glasgow

Dynamic dispatch

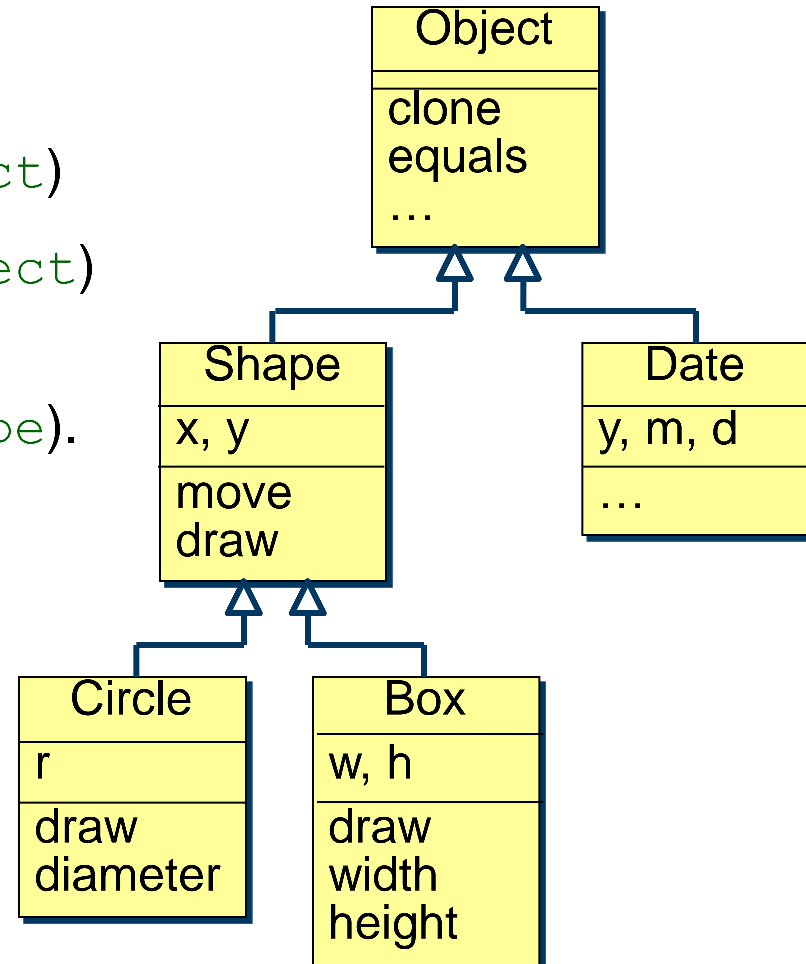
- In every OO PL, a variable of type *C* may refer to an object of any subclass of *C*.
- If method *M* is virtual, then the method call "`O.M(…)`" entails **dynamic dispatch**:
  - The *compiler* infers the type of *O*, say class *C*. It then checks that class *C* is equipped with an instance method named *M*, of the appropriate type.
  - At *run-time*, however, it might turn out that the receiver object is of class *C'*, a subclass of *C*. The receiver object's tag is used to determine its actual class, and hence determine which of the methods named *M* is to be called.

Single inheritance

- An OO PL supports **single inheritance** if each class has at most one superclass.
- Single inheritance gives rise to a hierarchy of classes.
- Single inheritance is supported by most OO PLs, including Java.

Example: Java single inheritance

- Declared classes:
  - `Date` (subclass of `Object`)
  - `Shape` (subclass of `Object`)
  - `Circle`, `Box` (both subclasses of `Shape`).
- Hierarchy of classes:

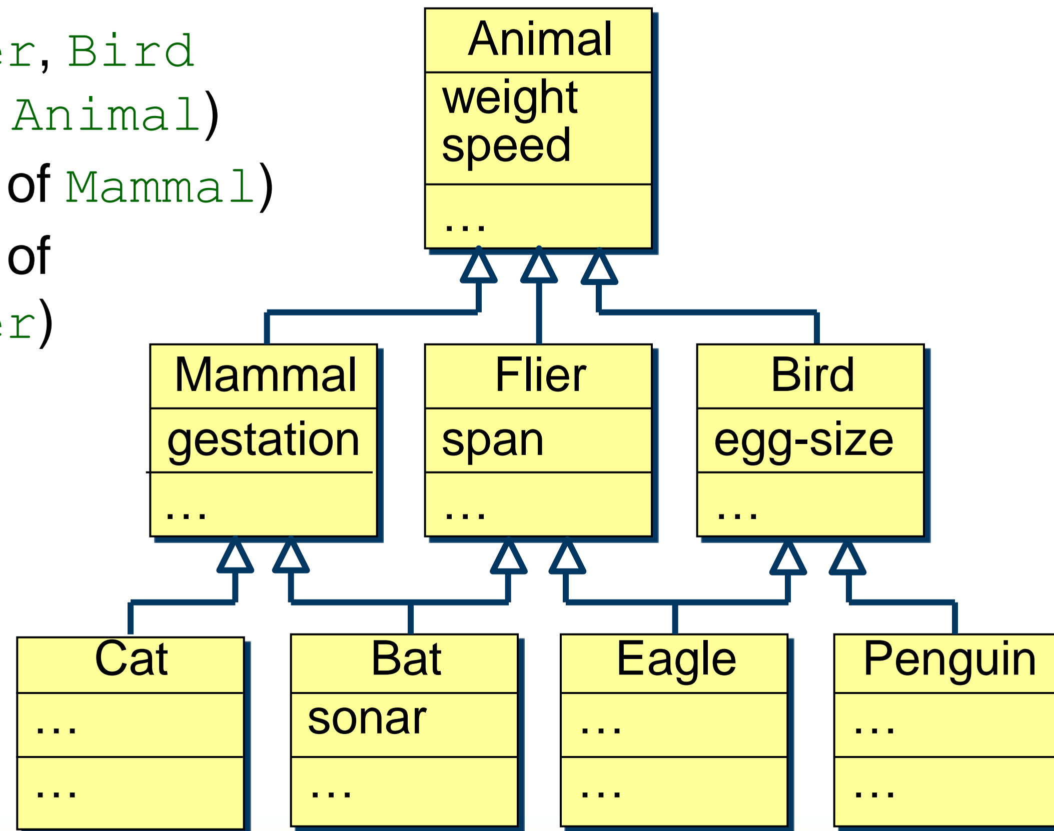


Multiple inheritance

- **Multiple inheritance** allows each class to have any number of superclasses.
- Multiple inheritance is supported by C++.
- Nevertheless, multiple inheritance gives rise to both conceptual and implementation problems.

Example: C++ multiple inheritance *(1)*

- Declared classes:
  - `Animal`
  - `Mammal`, `Flier`, `Bird` (subclasses of `Animal`)
  - `Cat` (subclass of `Mammal`)
  - `Bat` (subclass of `Mammal`, `Flier`)
  - etc.
- Class relationships:



University of Glasgow

Example: C++ multiple inheritance *(2)*

- Suppose:
  - the `Animal` class defines a method named `move`
  - the `Mammal` and `Flier` classes both override that method.
- Which method does the `Bat` class inherit?

```
Bat b = …; b.move(…);
```

- Possible answers:
  - Make it call the `Mammal` method.
  - Force the programmer to choose.
  - Make this method call illegal.