# Object Orientation

Based on the slides of Maurizio Gabbrielli

# 1 - Linguistic Concepts: Encapsulation

1. Encapsulation
2. Subtypes
3. Inheritance
4. Dynamic Lookup

- The designer of a concept has a "detailed" view of it
- The customer of the concept has an "abstract" view
- **Encapsulation** is the mechanism for separating these two views

- It is implemented through the rules of visibility

# Comparison

- In traditional languages: encapsulation through ADT or modules
- Advantages:
  - interface separate from implementation
- Disadvantages:
  - is not extensible (at least not as much as O-O)

Let's see an example with abstract types ...

# Abstract data types

**A queue:**

```
abstype q
    with
      mk_Queue : unit -> q
      is_empty : q -> bool
       insert   : q * elem -> q
       remove   : q -> elem
    is  * code *
    in
        * program *
end
```

**A priority queue:**

```
abstype pq
    with
      mk_Queue : unit -> pq
      is_empty : pq -> bool
       insert   : pq * elem -> pq
       remove   : pq -> elem
    is  * code *
    in
        * program *
end
```

Same signature (only type name differs)

But we can't "mix" queues and priority queues, although every code that uses a queue will use in a reasonable way even a priority queue!

# Abstract data types

- Ensure data structure invariants
  - Only the functions of the given type have access to the internal representation of the data
- But reuse is limited
  - You cannot apply code for code (type `Q`) to priority queues (type `PQ`), unless you use an explicit parameterization (e.g., template), even if the signatures (i.e. the interfaces) are identical!
  - We cannot create mixed data structures, e.g. colored dots and dots

- Moral: Encapsulation and abstraction over data is an important part of O-O, but the novelty is that it appears to you in an extensible way

# Classes

- A class is an abstraction representing a portion of the model

```
public class Circle {
    public double x, y;                 // Coordinates of Center
    public double r;                    // Radius

    private static final double PI = 3.14159265;        // (Local) constant

    public double circumference () {return 2 * PI * r;}          // Method
    public double area () {return PI * r * r;}                   // Method
}
```

- It defines the contents and capabilities of some kind of objects
- Objects are dynamically created with **`new`** and are allocated on the heap

# Objects

- Objects are values of type classes

```
Circle C;
c = new Circle ();
```

  - Creates an Instance of the Circle class, a single Circle object
- Data fields can be accessed

```
Circle c = new Circle();
c.x = 2.0;            // Our circle has center (2,2)
  // and radius 1
c.y = 2.0;
c.r = 1.0;
Circle d = new Circle();  // Another circle, center
     // (1,1)and radius 1
d.x = d.y = d.y = 1.0;
```

- Methods can be accessed too (with same syntax)

```
double a1,a2,len;
a1 = c.area();  // WE DO NOT WRITE: a1 = area(c);
len = c.circumference;
a2 = d.area();
```

# Information hiding: visibility modifiers

- To hide variables and methods we use modifiers:
  - public, private, protected

  - A `public` class or class member is visible everywhere

  - A `private` member of a class is visible only in methods defined within the class. Private members are not visible within subclasses, and are not inherited by subclasses as other methods are

  - A `protected` member of a class is visible in methods defined within the class and within all subclasses, and also within all classes that are in the same package as that class

  - Use `protected` visibility to hide class members from code that uses your class, but you want to give access to code extending your class

# 2 & 3 - Linguistic Concepts: Subtypes and Inheritance

1. Encapsulation
2. Subtypes
3. Inheritance
4. Dynamic Lookup

Attention:

- Concepts "close" and often confused
- often *do not* correspond to distinct mechanisms in the specific languages O-O

# What is an object's interface?

- Interface
  - Messages from an Object

    Including public variables

```
public class Circle {
   public double x,y;
   public double r;
   private static final double PI=3.14159265;
   public double circumference() {return 2*PI*r;}
   public double area() {return PI*r*r;}
}
```

Example: **`Circle`**

**`circumference`** : returns the circumference of a circle

**`area`**: returns the area of the circle

**`x,y,r`** : center and radius of the circle

therefore: an object's interface is its *type* (Java: Class)

# Subtype

```
Circle:
  double x,y;
  double r;
  double circumference();
  double area();
```

```
GraphicCircle:
  double x,y;
  double r;
  double circumference();
  double area();
  Color outline, fill;
  draw(DrawWindow dw) ;
```

- The interface for a `GraphicCircle` contains that of a `Circle`
  - `GraphicCircle` is a subtype of `Circle`

# Inheritance

- Is an implementation mechanism
- Allows the definition of new objects by reusing implementations of other objects
- The new objects *Inherit* part of their implementation

# Example

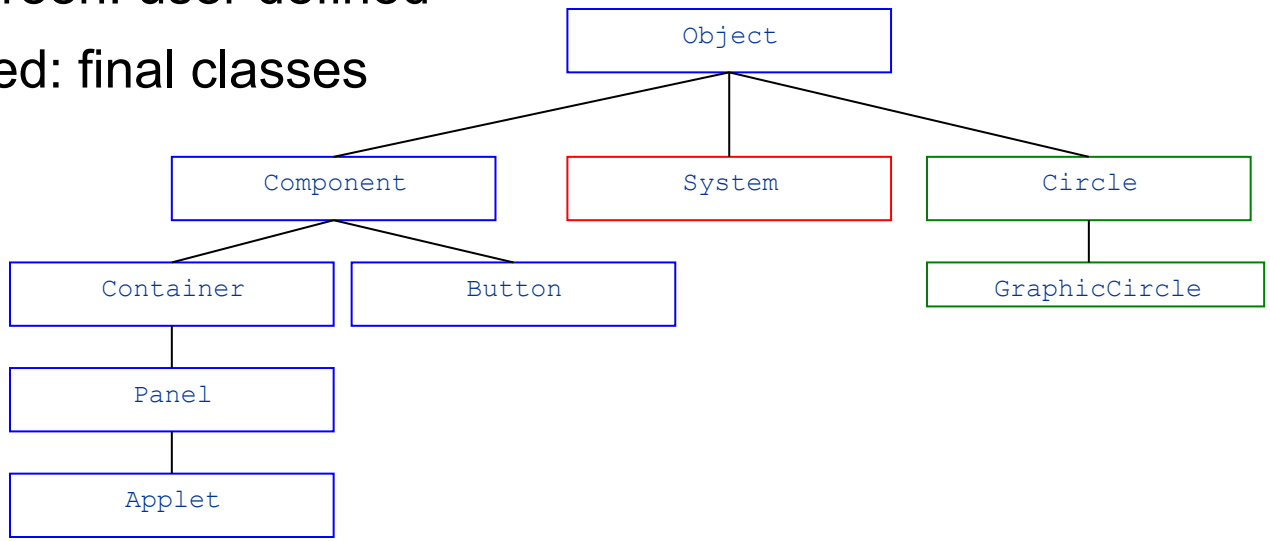
```
public class Point {
     private double x, y;
     public void move (double dx,
                       double dy) {…};
     }

public class Colored_point extends Point {
     private color c;
     public void change_color (color newc)
          {…};
     }
```

- Subtype
  - A colored point can be used in place of a point
  - Property used by the customer program
- Inheritance
  - A colored point can be implemented by re-using the implementation of `Point`
  - Property used by the class implementer

In Java the notion of subclass encompass two concepts: it provides a subtype and allows (to the abstract machine abstract) the inheritance

# The class Hierarchy

- Every defined class has a (unique) superclass
- If none is specified, the superclass is the class `Object`
- The class hierarchy is a tree
  - blue: defined in Java Application Programming Interface (API)
  - green: user defined
  - red: final classes



# Constructor Chaining

- Java guarantees that the class's constructor method is called whenever an instance is created and when an instance of any subclass is created
- Therefore, Java ensures that every constructor method calls its superclass constructor method
- If the first statement of a constructor is not an explicit call via `super`, then Java implicitly inserts the call `super()`

# Shadowed variables

- Variables in a subclass may shadow variables with the same name in a superclass

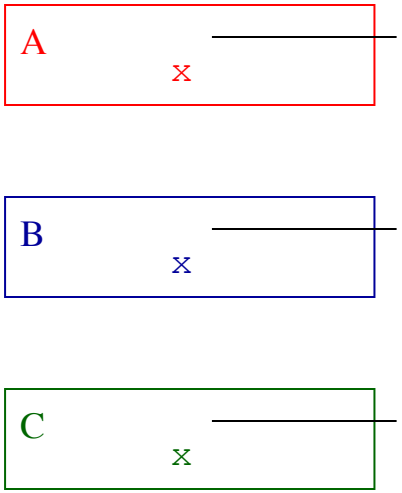
```
public class Circle{
    public double x,y,r;
    . . . }

public class NewCircle extends Circle{
    public int r;
    . . .
    public void P(){ . . . r = r++; . . . }
    . . . }
```

  - The declaration of `r` in `NewCircle` shadows the declaration of `r` in `Circle`
- The `r` variable of `Circle` may be used via `super` or via a casting:

```
public class NewCircle extends Circle{
    public int r;
    . . .
    super.r++; . . .   ((Circle)this).r++; . . . }
```

# Casting variables

- public class A {int x; . . . }
- public class B extends A {int x; . . . }
- public class C extends B {int x; . . . }




– The following code in class C:

```
x                   // var x in class C
this.x                  // var x in class C
super.x                 // var x in class B
((B)this).x         // var x in class B
((A)this).x         // var x in class A
super.super.x   // ILLEGAL: does not refer to x in class A
```

# Overriding methods

- “Shadowing” a method is called overriding
- Important and useful techniques

```
public class Time1{
    private int hour,min;
    . . .                                          //Constructor and access
    public void print() {
        System.out.println(hour,min);}
}
public class TimeSec1 extends Time1{
    private sec;
    . . .                                          //Constructor and access
    public void print() {
         System.out.println(getHour()+":"+getMin() +":"+sec);}
}
Time1 d = new Time1();
TimeSec1 s = new TimeSec1();
d.print; s.print;
```

# Overriding is not Shadowing

```
class A{
    int i = 1;
    int f(){return i;}
}
class B extends A{
    int i = 2;                          // Shadows var i in class A
    int f(){return -i;}            // Overrides method f in class A
}
public class OverrIsNotShad{
    public static void main(){
        B b = new B();
        System.out.println(b.i);     ???
        System.out.println(b.f());  ???

        A a = (A) b;       // Casts b to an instance of class A
        System.out.println(a.i); ???
        System.out.println(a.f()); ???
    }
}
```

# Overriding is not Shadowing

```
class A{
    int i = 1;
    int f(){return i;}
}
class B extends A{
    int i = 2;                          // Shadows var i in class A
    int f(){return -i;}           // Overrides method f in class A
}
public class OverrIsNotShad{
    public static void main(){
        B b = new B();
        System.out.println(b.i);     // Refers to B.i; prints 2
        System.out.println(b.f());  // Refers to B.f(); prints -2

        A a = (A) b;       // Casts b to an instance of class A
        System.out.println(a.i);// Now refers to A.i; prints 1
        System.out.println(a.f()); // Still refers to B.f(); prints -2
    }
}
```

# Virtual methods

- virtual function or virtual method:
  - function or method whose behaviour can be overridden within an inheriting class by a function with the same signature to provide the polymorphic behavior
- every non-static method in JAVA is by default virtual method except final and private methods
- in C++ virtual methods needs to be specified

# Virtual methods in C++

```
class Base
{
  public:
            void Method1 ()  {  std::cout << "Base::Method1" << std::endl;  }
    virtual void Method2 ()  {  std::cout << "Base::Method2" << std::endl;  }
};

class Derived : public Base
{
  public:
    void Method1 ()  {  std::cout << "Derived::Method1" << std::endl;  }
    void Method2 ()  {  std::cout << "Derived::Method2" << std::endl;  }
};

Base* obj = new Derived ();
  //  Note - constructed as Derived, but pointer stored as Base*

obj->Method1 ();  //  Prints "Base::Method1"
obj->Method2 ();  //  Prints "Derived::Method2"
```

# Abstract classes

- A class is an “interface” to its methods
- What about multiple implementations of the same class?
- Use `abstract` methods

```
public class Point{
  public abstract void move_right(double length);
  public abstract void print();
}
```

  - An abstract method has no body; only a signature definition
  - Any class containing an abstract method is abstract itself
  - Abstract classes cannot be instantiated, only subclassed
  - A subclass of an abstract class can be instantiated if overrides all of the abstract methods and provides an implementation for all of them

# Interfaces

- Sometimes we need implementations to match several interfaces
  But classes in Java can have only one superclass!
- Use interfaces, instead:

```
interface DrawableShapes{
  public void draw();
  }
interface Point{
    public abstract void move_right(double length);
    public abstract void print();
  }

class C_point implements DrawableShapes,Point {
  private double x,y;
  public C_point(double a, double b) {x=a; y=b;};
  public void move_right(double length) {x=x+length;}
  public void move_left(double length) {x=x-length;}
  public void print(){System.out.println("C_Punto
  ("+x+","+y+")");}
  public void draw(){. . .}
  }
```

# Interfaces, II

- Interfaces cannot be instantiated
- We can only produce classes implementing an interface
- Interfaces can be extended, like classes can have subclasses

```
interface DrawableShape{
   public void draw();
   }
interface DrawableAndResizableShape extends DrawableShape {
   public abstract void resize(double ratio);
}
class C_point implements DrawableAndResizableShapes,Point {
   . . .
   public void print(){. . .}                                  // For Point
   public void draw(){. . .}          // For DrawableShape
   public void resize(double ratio){…}        // For
      // DrawableAndResizableShape
   }
```

# Multiple Inheritance

- There are languages that support multiple inheritance (C++, Eiffel)
- Presents problems → name clashes

```
class A{
    int x;
    int f(){
        return x;
    }
}
class B{
    int y;
    int f(){
        return y;
    }
}
class C extending A,B{
    int h(){
        return f();
    }
}
```

# Name clashes, possible approaches

- Forbid clash syntactically
- Required that conflicts should be resolved by programmer
    - e.g. write B::f() or A::f() when there is a clash
- Decide based on convention (e.g. order in the extending clause)

- No solution is universally accepted
- Flexible tool but no simple, unequivocal and elegant solution

# Diamond Problem

- One class inherits from two superclasses, each inheriting from single superclass
- Which f?

**We will see later**

```
class Top{
    int w;
    int f(){
        return w;
    }
}
class A extending Top{
    int x;
    int g(){
        return w+x;
    }
}
class B extending Top{
    int y;
    int f(){
        return w+y;
    }
    int k(){
        return y;
    }
}
class Bottom extending A,B{
    int z;
    int h(){
        return z;
    }
}
```

# In C++

- In C++ inheritance and subtyping mechanism are independent
- Use public to introduce a subtype relation
    - B and C inherits from A, only B subtype of A

```
class A{
public:
   void f(){...}
   ...
}
class B : public A{
public:
   void g(){...}
   ...
}
class C : A{
public:
   void h(){...}
   ...
}
```

# Linguistic Concepts: Dynamic Lookup

1. Encapsulation
2. Subtypes
3. Inheritance
4. Dynamic Lookup

- In O-O programming,

  The code you run depends on the *method, arguments* and *object*

Warning: Dynamic lookup is not overloading!

Overloading is static: always resolved at compile time

Dynamic lookup always at run time.

# Dynamic selection and overriding

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
class NewCounter extending Counter{
    private int num_reset = 0;
    public void reset(){
        x = 0;
        num_reset = num_reset + 1;
    }
    public int quanti_reset(){
        return num_reset;
    }
}
```

```
NewCounter n = new NewCounter()
Counter c;
c = n;
c.reset() // which reset ?
```

# Dynamic selection
# Uniform treatment of objects of different types

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
class NewCounter extending Counter{
    private int num_reset = 0;
    public void reset(){
        x = 0;
        num_reset = num_reset + 1;
    }
    public int quanti_reset(){
        return num_reset;
    }
}
```

```
Counter V[100];

// fill V

for (int i = 0; i<
  100; i=i+1)
  V[i].reset();
```

# Late binding

```
class A{
    int a = 0;
    void f(){g();}
    void g(){a=1;}
}
class B extending A{
    int b = 0;
    void g(){b=2;}
}
```

```
B b = new B();
b.f(); // ???
```