Abstraction: Polymorphism

Abstracting Objects
Polymorphism

- Polymorphism is one of the central ideas of object-orientation (OO) – that a single object may take on multiple different roles within a program.
Polymorphism

- The word originates from Greek, meaning "having multiple forms."
  - “poly”: many
  - “morph”: forms
Polymorphism

• Some roles treat the object as it relates to its information content and true conceptual purpose within the program.

• Other roles may exist as an extreme abstraction of its true purpose, extracting a single aspect of the “true” object to allow abstracted methods to utilize it.
Polymorphism

• There are times when we do not need all the specific details of object, but merely a few pieces.
  – For sorting, we merely need a way to determine the ordering of two same-type objects – we could care less if they are ints or strings, for example.
  – A... “least common denominator”, if you will, among many types.
Polymorphism

• In order to facilitate this, a programmer may create custom types for the sole purpose of representing each such role.
  – In Java, these are called *interfaces*.
  – Each such custom type *declares* a set of methods necessary to fulfill the functionalities of that role.
  – For the last slide’s example, we would need a comparison method.
Polymorphism

• The idea is that each such custom type provides the minimum specification and blueprint necessary for performing that role.
  – This custom type is then implemented by classes in order to perform the represented role.
Polymorphism

- Since the specification and method names are declared in the custom type, those methods can be accessed from through the custom type, without needing more specific type information.

- The actual implementation is left to each implementing (specific) class.
Polymorphism in C++

• For a starter example, let’s suppose we want to use polymorphism to calculate geometrical properties of shapes.
  – The user first specifies a shape, with its relevant parameters.
  – Afterward, the user may ask for its perimeter length, area, or (ideally) for it to be drawn.
Polymorphism in C++

• We note that perimeter length and area are common properties of any shape.

• Shapes also commonly have visual forms.

• Thus, these are all reasonable properties for a common “Shape” role to have within our program.
Polymorphism in C++

class Shape
{
    public:
    virtual double area() = 0;
    virtual double perimeter() = 0;
}

- The "= 0" on each method indicates that our class Shape does not define the method – it is the responsibility of any class fulfilling this role to implement them instead.
Polymorphism in C++

class Shape
{
    public:
    virtual double area() = 0;
    virtual double perimeter() = 0;
}

• The keyword “virtual” on the methods indicates that Shape expects implementing classes to provide their own definition, and will allow those to be accessed from the Shape perspective.
Polymorphism in C++

• Note: because the area() and perimeter() methods have no implementation within Shape, it is not possible to create an instance of Shape directly.

• Instead, the point is to have other classes implement the Shape role and to be able to use them from that perspective.
class Circle: public Shape
{
    public:
    Circle(double r);
    double area();
    double perimeter();
    void draw();

    private:
    double radius;
};
double Circle::area()
{
    // Assuming a predefined PI constant.
    return PI * radius * radius;
}

Circle::Circle(double r)
{
    radius = r;
}

/* Implementation of the other methods left to the imagination. */
class Circle: \textbf{public} Shape
{
    double radius;
};

Note the phrasing here – this indicates that Circle is inheriting the specifications and preexisting blueprint of the type Shape.

\textbf{public} indicates that the original access modifiers of Shape should remain unchanged.
Polymorphism

• All of the following are legal code lines, assuming good class definitions.

Shape* s1 = new Circle(4);
Shape* s2 = new Square(4);
Shape* s3 = new Pentagon(4);
• Suppose, then, that we have a vector of Shapes:

```cpp
vector<Shape*> shapeList, and want to sum up the area of all the stored, referenced Shapes.
```

```cpp
double areaSum = 0;

for(int i=0; i < shapeList.size(); i++)
    areaSum += shapeList[i]->area();
```