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.
• 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, \textit{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.
Polymorphism in C++

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. */
Polymorphism in C++

class Circle: public Shape
{
 double radius;
}

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

public indicates that the original access modifiers of Shape should remain unchanged.
Polymorphism in C++

• If a class does not provide an implementation of a “pure” virtual method, it is impossible to directly instantiate that class.
  – This *includes* cases where an object does not define a virtual method declared in its parent class, as would happen if we left area() undefined within our Circle class.
Polymorphism

Polymorphism allows the programmer to provide alternate, abstracted views of an object.

- These “views” will limit access to class fields and methods, exposing only those defined as part of that “view.”

- In a manner of speaking, the type Shape exists to provide a restricted “view” of Circle.
Polymorphism

- Polymorphism allows the programmer to provide alternate, abstracted views of an object.
  - While this isn’t useful if all we have is Circle, the reasoning becomes more important with Square, Pentagon, and Hexagon (for example).
  - Squares aren’t Circles, but they are both Shapes, with Shape properties.
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);
Polymorphism

• Suppose, then, that we have `vector<Shape*> shapeList`, and want to sum up the area of all the stored, referenced Shapes.

double areaSum = 0;

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

Note how we’re storing each Shape via reference.

– This is because in creating a by-value variable of Shape, it is attached directly to an inherent Shape instance.

– For one, this is impossible to instantiate, as Shape has virtual methods.

– Secondly, that instance would be of exactly type Shape – attempting assignment would be on an implied Shape::operator=().
C++ Practicalities

• Note how we’re storing each Shape via pointer.
  – In short, use of polymorphism in C++ requires handling objects via their pointers.
  – A pointer of a subclass can always be stored as a variable of its superclass.
  – Technically, through polymorphism, a pointer of a subclass *is a* pointer to an instance of the superclass – just with further specification.
At this point in the class, we will not be examining full “inheritance.”

- Understanding and being able to use polymorphism is a large-enough issue at this time.

- The main difference: full inheritance allows for pre-defined methods and fields; actual functionality is, well, inherited.