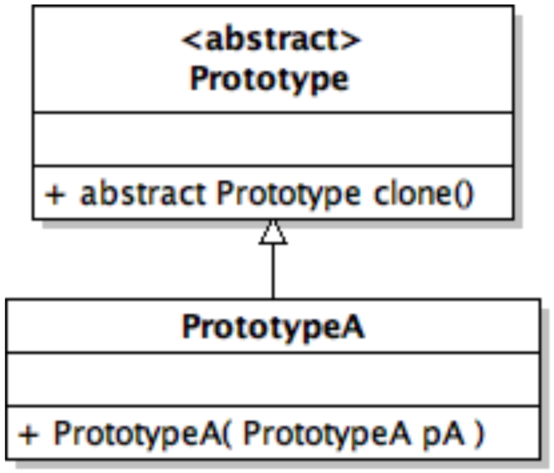
CECS 277 – Lecture 21 – Design Patterns

Prototype

The Prototype pattern is a Creational design pattern that allows you to create copies of existing objects by delegating the process of making the copy to the object itself. It is commonly used either when not all of the fields are publicly accessible (ie. private members and no get methods), and/or when the data type of the object to be cloned is not known (ie. the object is polymorphic). Since the object knows its own type and it has access to its own data members, it makes sense that the object itself should make the copy. A handy option in Java is to just implement Clonable and override Object.clone().

Prototype UML – The Prototype pattern has an abstract class or interface that has an abstract clone method, that way, all subclasses are forced to override the clone method. Each subclass requires an overloaded constructor that passes in an instance of itself. The clone method calls the overloaded constructor with 'this' as a parameter.



Prototype Template Code:

```
public interface Prototype {
   public Prototype clone( );
}

public class PrototypeA implements Prototype {
   private int value;
   public PrototypeA( int v ) {
      value = v;
   }
   public PrototypeA( PrototypeA p ) {
      if( p != null ) {
         value = p.value;
      }
   }
   @Override
   public Prototype clone( ) {
      return new PrototypeA( this );
   }
```

Prototype Example Code:

```
public abstract class Shape {
   private int x;
   private int y;
   public Shape( int xVal, int yVal ) {
      x = xVal;
      y = yVal;
   }
   public Shape( Shape s ) {
      if( s != null ){
         this.x = s.x;
         this.y = s.y;
      }
   }
   public abstract Shape clone( );
   @Override public boolean equals( Object o ) {
      if( o instanceof Shape ){
         Shape s = (Shape) o;
         return x == s.x && y == s.y;
      }
      return false;
   }
}
public class Circle extends Shape {
   private int radius;
   public Circle( int xVal, int yVal, int r ) {
      super( xVal, yVal );
      radius = r;
   }
   public Circle( Circle c ) {
      super( c );
      if( c != null ) {
         radius = c.radius;
      }
   }
   @Override public Shape clone( ) {
      return new Circle( this );
   }
   @Override public boolean equals( Object o ) {
      if( o instanceof Circle ) {
         Circle c = (Circle) o;
         return super.equals( c ) && radius == c.radius;
      }
      return false;
   }
}
```

```
public class Main {
   public static void main( String [] args ) {
      Circle c1 = new Circle( 3, 7, 2 );
      Circle c2 = new Circle( 1, 3, 2 );

      //Can’t copy over values, no get methods!
      //Circle c1Clone = new Circle(c1.getX(),
                              c1.getY(), c1.getRadius());

      //Can create shallow copy from same memory address
      //but if the clone is modified, the original is too
      Circle c1Clone = c1;

      //Use clone() to create new instance with same values
      Shape c2Clone = c2.clone( );

      //comparing memory addresses
      if( c1 == c1Clone ) {
         System.out.println( “C1 - Same object” );
      } else {
         System.out.println( “C1 - Different objects” );
      }
      if( c2 == c2Clone ) {
         System.out.println( “C2 - Same object” );
      } else {
         System.out.println( “C2 - Different objects” );
      }

      //comparing values
      if( c1.equals( c1Clone ) ) {
         System.out.println( “C1 – Same values” );
      } else {
         System.out.println( “C1 – Different values” );
      }
      if( c2.equals( c2Clone ) ) {
         System.out.println( “C2 – Same values” );
      } else {
         System.out.println( “C2 – Different values” );
      }
   }
}
/* Output:
C1 – Same object
C2 – Different objects
C1 – Same values
C2 – Same values
*/
```