AP Computer Science
Curriculum Module: An Introduction to Polymorphism in Java

Dan Umbarger
AP Computer Science Teacher
Dallas, Texas

An Introduction to Polymorphism in Java

The term **homonym** means “a word the same as another in sound and spelling but with different meaning.” The term **bear** could be a verb (to carry a burden) or it could be a noun (a large, hairy mammal). One can distinguish between the two usages through the use of context clues. In computer science the term **polymorphism** means “a method the same as another in spelling but with different behavior.” The computer differentiates between (or among) methods depending on either the method signature (after compile) or the object reference (at run time).

In the example below polymorphism is demonstrated by the use of multiple `add` methods. The computer differentiates among them by the method signatures (the list of parameters: their number, their types, and the order of the types.)

```
// A Java program written to demonstrate compile-time
//   polymorphism using overloaded methods

public class OverLoaded
{
  public static void main(String [] args)
  {
     DemoClass obj = new DemoClass();
     System.out.println(obj.add(2,5));        // int, int
     System.out.println(obj.add(2, 5, 9));    // int, int, int
     System.out.println(obj.add(3.14159, 10));  // double, int
  }  // end main
}// end OverLoaded
```

```
public class DemoClass
{
   public int add(int x, int y)
   {
      return x + y;
   }// end add(int, int)

   public int add(int x, int y, int z)
   {
      return x + y + z;
   }// end add(int, int, int)

   public int add(double pi, int x)
   {
      return (int)pi + x;
   }// end add(double, int)
}// end DemoClass
```

This form of polymorphism is called **early-binding** (or **compile-time**) **polymorphism** because the computer knows after the compile to the byte code which of the `add` methods it will execute. That is, after the compile process when the code is now in byte-code form, the computer will “know” which of the `add` methods it will execute. If there are two actual `int` parameters the computer will know to execute the `add` method with two formal `int` parameters, and so on. Methods whose headings differ in the number and type of formal parameters are said to be **overloaded methods**. The parameter list that differentiates one method from another is said to be the **method signature list**.

There is another form of polymorphism called **late-binding** (or **run-time**) **polymorphism** because the computer does not know at compile time which of the methods are to be executed. It will not know that until “run time.” Run-time polymorphism is achieved through what are called **overridden methods** (while compile-time polymorphism is achieved with overloaded methods). Run-time polymorphism comes in two different forms: run-time polymorphism with abstract base classes and run-time polymorphism with interfaces. Sometimes run-time polymorphism is referred to as **dynamic binding**.

Types of Run-Time Polymorphism

There are five categories or types of run-time polymorphism:

1. Polymorphic assignment statements
2. Polymorphic Parameter Passing
3. Polymorphic return types
4. Polymorphic (Generic) Array Types
5. Polymorphic exception handling *(not in AP subset*)

1. Polymorphic Assignment Statements

When learning a new concept, it is often helpful to review other concepts that are similar and to use the earlier, similar skill as a bridge or link to the new one. Look at the following declaration:

```
int x = 5;
double y = x;   // results in y being assigned 5.0
```

This is an example of "type broadening." The `int x` value of `5`, being an `int` which is a subset of the set of `doubles`, can be assigned as the value of the `double y` variable.

On the other hand,

```
double x = 3.14;
   int y = x;
```

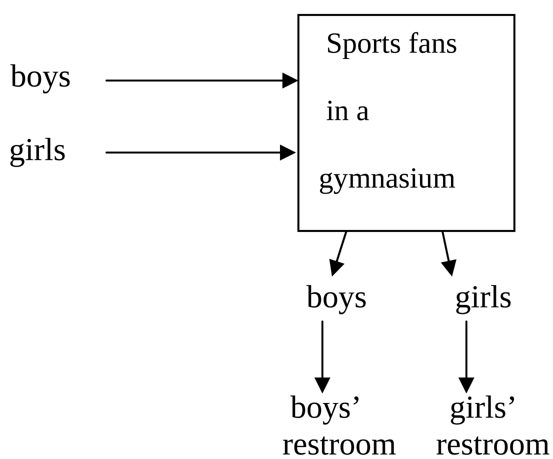
results in the compile error message "`Possible loss of precision`." The JVM knows that it will have to truncate the decimal part of `3.14` to do the assignment and is fearful to do so, thinking that you have made a mistake. You can assure the JVM that you really do know what you are doing and really do wish to effect that truncation by coding a "type cast."

```
double x = 3.14;
    int y = (int) x;
```

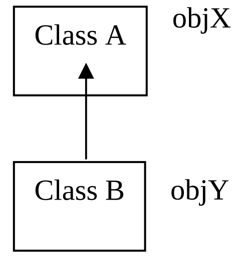
At right is some curious code to analyze. The variable value `y` received from `x` was originally an `int` value (`5`), but we are not allowed to assign that value (`5`) to the `int` variable `z` without a type cast on `y`. It seems as though the "type broadening " from `5` to `5.0` has somehow changed the nature of the value. This situation will be helpful to remember in another few pages when we discuss a concept called "down-casting."

```
public class DownCast
{
        public static void main(String [] args)
        {
                int x = 5;
                double y = x;
                //int z = y;    y = x = 5 right???
                int z = (int)y;  // now it's O.K.
        }// end main
}// end class

Possible loss of precision (compile error)
```

Consider the following example. In the figures shown here boys and girls enter a gymnasium where they become generic sports fans, but are not allowed to enter gender-specific restrooms without first being converted back (type cast) to their specific gender types.



We now move from discussing primitive variables to object reference variables. The figure at the right pictorially represents an “is-a” relation between two classes. `ClassB` is an extension of `ClassA`. `ObjY` is a type of `ClassA`, but `objX` is not a type of `ClassB`. This relation is not symmetrical.




```
public class PolymorphicAssignment
{
        public static void main(String [] args)
        {
            ClassA obj1 = new ClassA();
            ClassA obj2 = new ClassA();
            ClassB obj3 = new ClassB();

1)          obj1 = obj2;   // no problem here...same data types
2)          obj1 = obj3;   // obj3 is a type of ClassA…ok
3)          //obj3 = obj2;   // "incompatible types" compile message
4)          //obj3 = obj1;     // still incompatible as the obj3 value
                               // stored in obj1 (see line 2 above)
                               // has lost its ClassB identity
5)          obj3 = (ClassB)obj1; // the ClassB identity of the object
                             // referenced by obj1 has been retrieved!
                             // This is called "downcasting"
6)          obj3 = (ClassB)obj2; // This compiles but will not run.
                             // ClassCastException run time error
                             // Unlike obj1 the obj2 object ref. variable
                             // never was a ClassB object to begin with

        }  // end main
}// end class
```

```
public class ClassA
{

}// end ClassA
```

```
public class ClassB extends ClassA
{

}// end ClassB
```

In the code above, **line 1** is a snap. Both object reference variables `obj1` and `obj2` are of `ClassA()` type. Life is good. **Line 2** works because `obj3` is an object reference variable of type `ClassB`, and `ClassB` type variables are a type of `ClassA`. `Obj3` is a type of `ClassA`. Life is still good. **Line 3** will not compile, as the code is attempting to assign a `ClassA` variable value to a variable of `ClassB` type. That is analogous to trying to assign a `double` value to an `int` variable. **Line 4** is more complicated. We know from **line 2** that `obj1` actually does reference a `ClassB` value. However, that `ClassB` information is now no longer accessible as it is stored in a `ClassA` object reference variable. **Line 5** restores the `ClassB` class identity before the assignment to `ClassB` object reference variable `obj3` with a type cast. Life is good again. **Line 6** is syntactically equivalent to **line 5** and will actually compile because of it, but will result in a “`ClassCastException`” at run time because `obj2` never was `ClassB` data to begin with.

Exercise 1:

1. How is **line 2** above conceptually similar to the discussion of “type broadening?” Use the term “is-a” in your response.
2. What is wrong with the code in **lines 3 and 4** above? Relate to the discussion on the previous page.
3. Why isn’t **line 4** okay? From **line 2** aren’t you assigning a `ClassB` value to a `ClassB` variable?
4. **Lines 5 and 6** are syntactically equivalent. They both compile but **line 6** will not execute. Why? Explain the difference between those two lines.

Code to Demonstrate Polymorphic Assignment

```
import java.util.Random;

public class PolyAssign
{
    public static void main(String [] args)
    {
        Shape shp = null;

        Random r = new Random();
        int flip = r.nextInt(2);

        if (flip == 0)
            shp = new Triangle();
        else
            shp = new Rectangle();

        System.out.println("Area = " + shp.area(5,10));
    }  // end main
}// end class
```

```
abstract class Shape
{
  public abstract double area(int,int);

} // end Shape

public class Triangle extends Shape
{
    public double area(int b, int h)
    {
      return 0.5 * b * h;
    }

}// end Triangle.

public class Rectangle extends Shape
{
    public double area(int b, int h)
    {
      return  b * h;
    }
}// end Rectangle
```

Output is chosen at random: either
Area = 25 (area triangle) or Area = 50 (area rect)

Here we see run-time polymorphism at work! The JVM will not know the value of variable "**`shp`**" until run time, at which time it selects the `area()` method associated with the current object assigned to "**`shp`**." The "`abstract`" specification on class `Shape` means that that class cannot be instantiated and only exists to indicate common functionality for all extending subclasses. The "`abstract`" specification on the `area` method means that all extending subclasses will have to provide implementation code (**unless they also are `abstract`**).

2. Polymorphic Parameter Passing

Early in the Java curriculum we encounter the syntax requirement that actual (sending) and formal (receiving) parameters must match in number, type, and sequence of type. With polymorphism we can write code that appears to (but does not actually) break this rule. As before, we use the idea of "type broadening" of primitives as a bridge to understanding how polymorphism works with parameter passing of objects. If we write the code:

```
int actual = 5;
method(actual);
```

and the method

```
public void method(double formal)
{      implementation code     }
```

the `int actual` parameter is "broadened" to a `5.0` and received by parameter `formal`.

Note that

```
double actual = 5.0;
method(actual);
```

and the method

```
public void method(int formal)
{   implementation code}
```

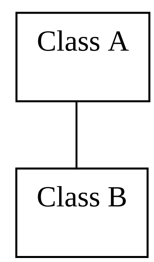
would result in a "`type incompatibility`" compile error message unless a type cast were made on the actual (sending) parameter first: `method( (int) actual);`

We now move from the discussion of type broadening of primitives to the idea of polymorphic parameter passing. We create objects for each of the two classes shown at right here and proceed to show their objects being passed as parameters to a method.

```
public class PolymorphicParameterPassing
{
        public static void main(String [] args)
        {
            ClassA obj1 = new ClassA();
            ClassA obj2 = new ClassA();
            ClassB obj3 = new ClassB();

        1)    method1(obj1);
        2)    method1(obj3);
        3)    //method2(obj1);
        4)    obj1 = obj3;
        5)   //method2(obj1);
        6)    method2((ClassB) obj1);
        7)   // method2((ClassB) obj2);
        }  // end main

public static void method1(ClassA formal)  {}

public static void method2(ClassB formal)  {}

}// end class
```

```
public class ClassA
{

}// end ClassA
```

```
public class ClassB extends ClassA
{

}// end ClassB
```



In **line 1,** at left, an object reference variable of `ClassA` type is passed to `method1` and received as a `ClassA` object reference variable. Actual and formal parameter types are the same. Life is good! **Line 2** shows a `ClassB` object reference variable passed to and received as a `ClassA` type variable. This is okay, as a `ClassB` type variable "is-a" type of `ClassA` variable. **Line 3** fails, as you are passing a superclass type variable to be received as a subclass type. It seems as though **line 5** should work, as `obj1` received the value of a `ClassB` variable, but it doesn't work unless the `ClassB` identity is restored through a type cast as shown in **line 6**. **Line 7** will compile, as it is syntactically the same as **line 6,** but **line 7** will result in a "`type cast exception`" upon program execution.

Code to Demonstrate Polymorphic Parameter Passing

```
import java.util.Random;
public class PolyParam
{
        public static void main(String [] args)
        {
            Shape shp;
            Shape tri  = new Triangle();
            Shape rect = new Rectangle();
            Random r = new Random();
            int flip = r.nextInt(2);
            if (flip == 0)
               shp = tri;
            else
               shp = rect;

            printArea(shp);   // output will vary
        }  // end main

        public static void printArea(Shape s)
        {
                System.out.println("area = " + s.area(5, 10));
        }// end printArea()
}  //    end class
```

```
abstract class Shape
 {
   abstract double area(int a, int b);
 }// end Shape

public class Triangle extends Shape
{
        public double area(int x, int y)
        {
           return 0.5 * x * y;
        }

}// end Triangle

public class Rectangle extends Shape
{
        public double area(int x, int y)
        {
           return x * y;
        }
}// end Rectangle
```

Exercise 2: What is the output of this program? Why?

3. Polymorphic Return Types

When returning values from return methods we know that there must be a type compatibility between the type of the variable receiving the value and the value being returned. For example:

`int x = retMethod();` and `public int retMethod(){ return 5;}`

We can code: `double x = retMethod();` and `public int retMethod(){ return 5;}`

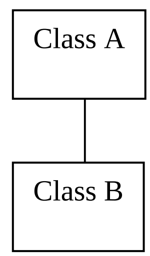
because the value being returned (`5`) is a type of `double` and, after the `int 5` is "broadened" to a `5.0`, that value can be assigned to the `double` variable `x`.

The code `int x = retMeth2();` with `public double retMeth2() { return 5;}`

results in a "`type compatibility`" error message (even though `5.0` was originally an integer value) unless we type cast the `double` return value as:

```
int x = (int) retMeth2();
```

Again we move from discussion of type broadening of primitives to the idea of polymorphic return types. We create objects for each of the two classes shown at right here and proceed to show their objects being used in return methods.




```
public class PolymorphicReturnTypes
{
   public static void main(String [] args)
    {
        ClassA obj1 = new ClassA();
        ClassA obj2 = new ClassA();
        ClassB obj3 = new ClassB();

        1.)  obj1 = method1();
        2.)  obj1 = method2();
        3.)  //obj3 = method1();  // incompatible types
        4.)  //obj3 = method3();  // incompatible...why?
        5.)  obj3 = (ClassB) method3();
        6.)  //obj3 = (ClassB) method1();

     }  // end main

     public static ClassA method1()  {  return new ClassA(); }

     public static ClassB method2()  {  return new ClassB(); }

     public static ClassA method3()  {  return new ClassB(); }

}// end class
```

```
public class ClassA
{

}// end ClassA
```

```
public class ClassB extends ClassA
{

}// end ClassB
```

Exercise 3:

1. Why does **line 1** compile and execute?
2. Why does **line 2** compile and execute?
3. Why does **line 3** fail to compile and execute?
4. Why does **line 4** fail to compile and execute? How is **line 4** different from **line 3**?
5. **Line 5** is similar to **line 4**. How does it succeed when **line 4** fails to compile?
6. **Line 6** is similar to **line 5**. **Line 6** will compile but not execute. Why?

Code to Demonstrate Polymorphic Return Types

```
import java.util.Random;

public class PolyReturn
{
        public static void main(String [] args)
        {
            Shape shp = retMethod();
            System.out.println(shp.area(5, 10));
        }  // end main

        public static Shape retMethod()
        {
                Random r = new Random();
                int flip = r.nextInt(2);
                if (flip == 0)
                    return new Triangle();
                else
                    return new Rectangle();
        }// end retMethod()
}// end class
```

```
abstract class Shape
{
   public abstract double area(int a, int b);
}// end Shape

class Triangle extends Shape
{
        public double area(int x, int y)
        {
          return 0.5 * x * y;
        }
}// end Triangle

class Rectangle extends Shape
{
        public double area(int x, int y)
        {
            return x * y;
        }
}// end Rectangle
```

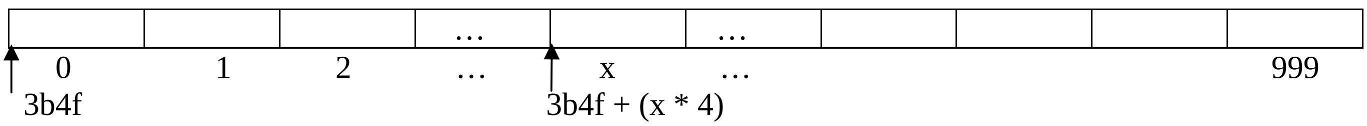
Exercise 4: What is the output of the preceding code? Why?

4. Polymorphic (Generic) Array Types

Arrays in Java, as with other languages, are homogeneous data types. This means that every array position must have the same type, hence the same length.

```
int arr = new int[1000];   // every array position reserves space for 4 bytes (in Java)
```

The declaration above reserves 4000 (1000 * 4) bytes of contiguous memory—each 4 bytes of memory storing an `int` value. Combining homogeneous position length together with contiguous memory is what gives arrays their O(1) access. In the figure below, storage or retrieval of data in position x can be determined by the formula "address of arr[x] =   base + (x * 4)." Hence if the first item in the array is located in memory position 3b4f then the *x*th position in the array would be 3b4f + (x * 4).



With "one-based counting" the same addressing formula would be "address of data in array = base + **(x – 1)** * 4." This is why computer scientists like "zero-based counting." Thus contiguous memory arrays allow us to use a single identifier for many, many different items of data while still having O(1) access. We need only to observe the restriction that every position in the array reserves the same amount of memory. If, instead of working with an array of primitives, we declare an array of objects, we can still achieve O(1) access but also achieve the effect of a "virtual" heterogeneous array. This effect will require some background to understand but is important as it is the "trick" behind polymorphic arrays.

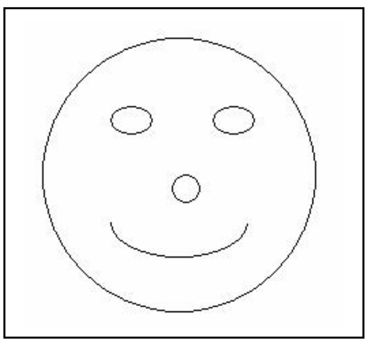
The code example below is used to prepare the reader for the idea of polymorphic arrays.

<APPLET CODE="Poly1.class" WIDTH=900 HEIGHT = 600> </APPLET>

```
public class Poly1 extends java.applet.Applet
{
    public void paint(Graphics g)
   {
          Head objRef0 = new Head();
          objRef0.draw(g);
          RightEye objRef1 = new RightEye();
          objRef1.draw(g);
          LeftEye objRef2 = new LeftEye();
          objRef2.draw(g);
          Nose objRef3 = new Nose();
          objRef3.draw(g);
          Mouth objRef4 = new Mouth();
          objRef4.draw(g);
    }// end paint
}// end class poly1
```

```
public class Head
{
          public void draw(Graphics g)

          {   g.drawOval(75, 75, 400, 400);  }

}// end class Head

public class RightEye
{
          public void draw(Graphics g)

          {   g.drawOval(150, 200, 60, 40);   }

}// end class RightEye


public class LeftEye
{
          public void draw(Graphics g)

          {   g.drawOval(300, 200, 60, 40);   }

}// end class LeftEye

public class Nose
{
          public void draw(Graphics g)

          {   g.drawOval(250, 300, 30, 30);   }

}// end class Nose

public class Mouth
{
          public void draw(Graphics g)

          {   g.drawArc(175, 300, 200, 100, 180, 180);   }

}// end class Mouth
```

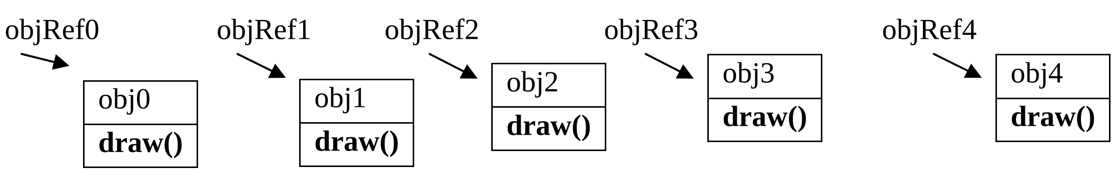
The code in method `paint` above is implemented by the JVM by creating five object reference variables, each referencing an object as shown in the figures below.



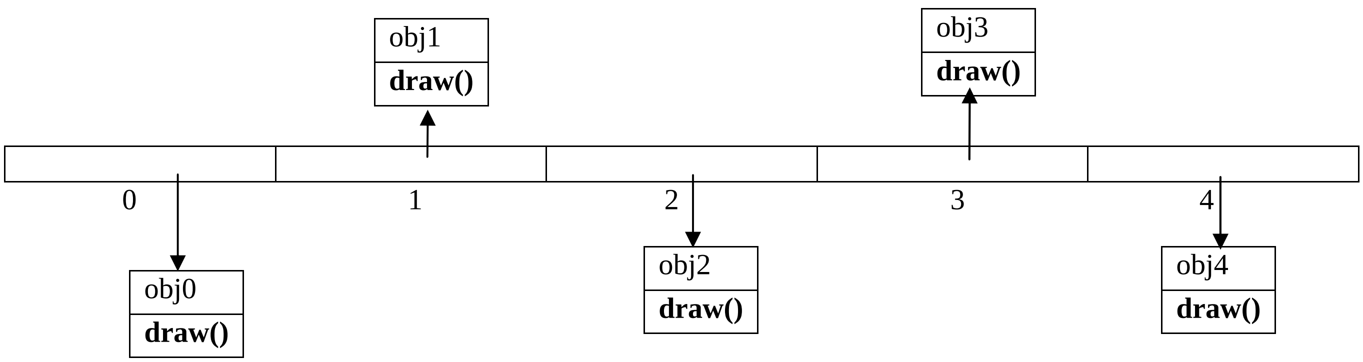
objRef0.**draw**(g);
objRef1.**draw**(g);
objRef2.**draw**(g);
objRef3.**draw**(g);
objRef4.**draw**(g);

compile time polymorphism

Each respective `draw()` method here has a different implementation which draws a different part of the face.



What we wish to do is to rewrite the code above so that the effect will be as follows:



If we could place all the object references inside an array, then the code

```
objRef0.draw(g);
objRef1.draw(g);
objRef2.draw(g);
objRef3.draw(g);
objRef4.draw(g);
```

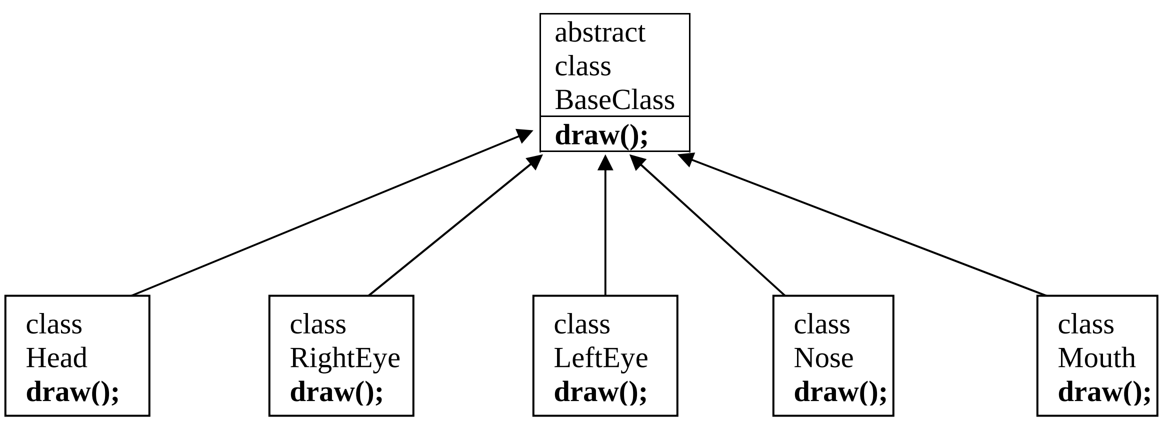
could be replaced by

```
for (int k = 0; k < 5; k++)
        refArray(k].draw(g);
```

Run-Time Polymorphism

Compile-Time Polymorphism

The benefit of this new code would increase as the number of desired `draw` methods increased. The problem of achieving such code is the requirement that all contiguous memory arrays be typed—the same type—so that each array position will be the same length. What type could we choose? We might try `Object refArray = new Object[5].` But then there would be the problem of “type casting” each object back to its reference variable subclass type. Java 1.5 automatically unboxes for the `ArrayList` but only if a common type can be specified when the `ArrayList` is declared: `ArrayList <type> refArray = new ArrayList<type>();` What could we use for `<type>` ? What we need is an abstract “place-keeper” type—sort of like a variable in Algebra I, but representing an unknown type as opposed to an unknown value.



Classes `Head`, `RightEye`, `LeftEye`, `Nose`, and `Mouth` all extend from abstract class `BaseClass`. Therefore each of those classes “is-a” type of `BaseClass`. Hence the code we are looking for to instantiate an array of generic type would be:

```
BaseClass [] refArray = new BaseClass[5];
```

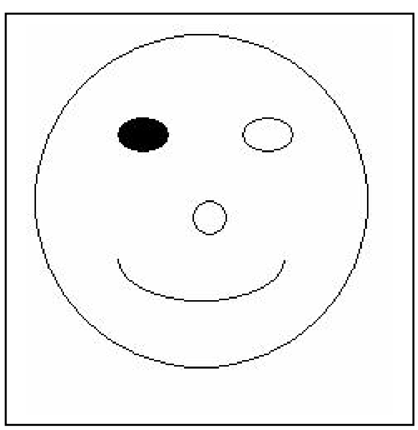
Below is the code to draw the happy-face figure using a “polymorphic array” with an abstract base class type.

```
import java.awt.*;

public class PolyArray extends java.applet.Applet
{
        public void paint(Graphics g)
        {
            BaseClass [] refArray = new BaseClass[5];

            refArray[0] = new Head();
            refArray[1] = new LeftEye();
            refArray[2] = new RightEye();
            refArray[3] = new Nose();
            refArray[4] = new Mouth();

            for (BaseClass e: refArray)
                e.draw(g);

        }// end paint
}// end class PolyArray

abstract class BaseClass
{
        abstract void draw(Graphics g);
}
```

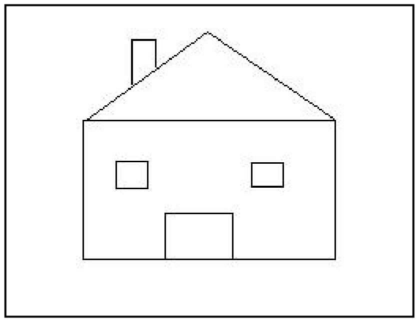
```
class Head extends BaseClass
{
        public void draw(Graphics g)

        {   g.drawOval(75, 75, 400, 400);  }

}// end class Head

class RightEye extends BaseClass
{
        public void draw(Graphics g)

        {   g.fillOval(150, 200, 60, 40);   }

}// end class RightEye


class LeftEye extends BaseClass
{
        public void draw(Graphics g)

        {   g.drawOval(300, 200, 60, 40);   }

}// end class LeftEye
```

```
class Nose extends BaseClass
{
        public void draw(Graphics g)

        {   g.drawOval(250, 300, 30, 30);   }

}// end class Nose
```

```
class Mouth extends BaseClass
    {
        public void draw(Graphics g)

        {  g.drawArc(175, 300, 200, 100, 180, 180);  }

    }  // end class Mouth
```



Exercise 5: Using a polymorphic array and abstract base class, draw the house below.



There is another way to effect a polymorphic array, through a technique known as using a generic interface (base class). An interface (base class) is similar to an abstract base class but with differences. The table below compares an abstract base class with an interface base class. They both can serve as "generic" types.

Abstract Base Class	Interface (abstract by default)
1) Must be specified as abstract. Classes are concrete by default. 2) Can have both variables and constants 3) Usually has a mix of concrete and abstract methods. 4) Subclasses **extend** abstract base classes. 5) Subclasses can only extend 1 abstract base class. 6) Methods can be private, public, or protected.	1) Abstract may be specified but is default abstract regardless. Does not implement code! 2) Can only have constants, no variables. 3) Can only have abstract methods. Interface methods may be specified as abstract but are default abstract regardless. 4) Subclasses **implement** an interface (base class). 5) Subclasses can implement more than 1 interface. 6) All methods (actually method headings) are public.

Below is the code to draw the happy-face figure using a "polymorphic array" with an abstract interface class type.

```
import java.awt.*;

public class PolyArray extends java.applet.Applet
{
    public void paint(Graphics g)
    {
            BaseInterface [] refArray = new BaseInterface[5];

            refArray[0] = new DrawHead();
            refArray[1] = new DrawLeftEye();
            refArray[2] = new DrawRightEye();
            refArray[3] = new DrawNose();
            refArray[4] = new DrawMouth();

            for (BaseInterface e: refArray)
                  e.draw(g);

    }// end paint
}// end class poly1

interface BaseInterface
{
            /* abstract */ void draw(Graphics g);
}
```

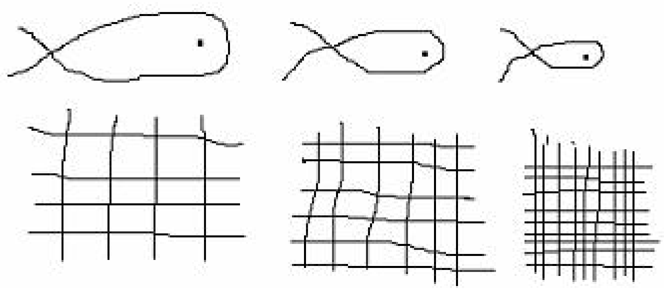
```
public class Head implements BaseInterface
{
            public void draw(Graphics g)

            {   g.drawOval(75, 75, 400, 400);  }

}// end class DrawHead

public class RightEye implements BaseInterface
{
            public void draw(Graphics g)

            {   g.fillOval(150, 200, 60, 40);   }

}// end class RightEye

public class LeftEye implements BaseInterface
{
            public void draw(Graphics g)

            {   g.drawOval(300, 200, 60, 40);   }

}// end class LeftEye
```

```
public class Nose implements BaseInterface
{
            public void draw(Graphics g)

            {   g.drawOval(250, 300, 30, 30);   }

}// end class Nose
```

```
public class Mouth implements BaseInterface
{
      public void draw(Graphics g)

            {    g.drawArc(175, 300, 200, 100, 180, 180); }

 }  // end class Mouth
```

5. Polymorphic Exception Handling (*Optional, not in AP subset*.)

To catch only the large fish in the figure below we would use the net with the large mesh, which would allow the two other smaller fish to swim right through the net. Then, to catch the middle-sized fish, we would use the net with middle-sized mesh, allowing the smaller fish to swim through the net. On the other hand, if we used the net with the smallest mesh on our first cast we would catch all three fish at once. Catching exceptions with try-catch blocks is similar to catching fish of different sizes, due to the polymorphic relation of the exception classes.



In the API segment below, you can see that the bottom class in the extended inheritance relation indicated is the "`ArrayIndexOutOfBoundsException`." Catching an exception with a "`ArrayIndexOutOfBoundsException`" object is analogous to catching the fish with the large mesh. It can only catch the largest fish. Due to polymorphism, the higher up one goes in the chain of inheritance the more errors can be caught (analogous to using a finer-mesh fishnet). The ability to catch errors selectively allows for different sections of recovery code targeted at specific error situations.

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ java.lang.RuntimeException
              └ java.lang.IndexOutOfBoundsException
                  └ java.lang.ArrayIndexOutOfBoundsException
```

```
import java.util.ArrayList;
import static java.lang.System.*;

public class PolyException
{
 public static void main(String [] args)
 {
  int [] x = new int[10];
  String str = "Dallas";
  try{   x[-1] = 3;                    //  error #1
         int y = str.charAt(-1); //  error #2
         int z = 5/0;                  //  error #3
   }
   // catches all three errors shown in try block
   catch(RuntimeException e) {
         out.println("RuntimeException");
   } // end catch RuntimeException
```

```
// catches "big fish", then "middle fish", then "little fish"
/*
catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("ArrayIndex OutOfBoundsException");
 }// end catch ArrayIndexOutOfBoundsException

catch(IndexOutOfBoundsException e){
      System.out.println("IndexOutOfBoundsException");
 } // end IndexOutOfBoundsException

  catch(RuntimeException e) {
      System.out.println("RuntimeException");
 }// end RuntimeException
*/
```

```
  }// end main
}// end class
```

Exercise 6:

1. Type in and execute the code above.
2. One at a time, comment out errors 1 and 2, recompile, and rerun. Explain the resulting outputs from the three executions.
3. Reactivate all three errors, comment out the single catch block shown in this code example, then activate the commented three catch blocks shown to the right. Recompile and rerun. Again, one at a time, comment out error 1 and 2 as before, and rerun. Explain the resulting outputs.

Answers

Exercise 1:

1. Line 2 of PolymorphicAssignment.java assigns a Class B type object reference variable as the value of a Class A object reference variable. This is okay as `ClassB` extends `ClassA` and therefore `obj3`'s type "is a" type of Class A.
2. Line 3 is attempting to assign a superclass type object variable as the value of a subclass object reference variable. This is like trying to assign a `double` value to an `int`. This is also true for line 4, but lines 3 and 4 differ from each other as the value being assigned in line 3 (`obj2`) never was a `ClassB` object while the value being assigned in line 4 (`obj1`) was originally a `ClassB` object prior to the conversion to superclass type (like type broadening from `int` to `double`). This distinction between lines 3 and 4 is crucial to understanding the difference between lines 5 and 6.
3. Line 4 is not allowed because even though `obj1` was, in line 2, assigned the value of a `ClassB` variable, `obj1`'s internal format was somehow changed after line 2 to the superclass type in a manner analogous to the type broadening from `int` to `double`.
4. Line 5 works because `obj1`'s value was originally a `ClassB` type (see line 2) but line 6 does not work as `obj2` never was a `ClassB` type to begin with.

Exercise 2:

The output of PolyParam.java is chosen at random to be either "area = 50" or "area = 25." Assignment of the variable **`shp`** is made at random to be either **`tri`**, a variable of class type `Triangle`, or **`rect`**, a variable of class type `Rectangle`. Both those respective classes have overridden the method area in the abstract superclass to reflect the different areas for the triangle and the rectangle. Both the assignment statements `shp = tri;` and `shp = rect;` are syntactically possible due to polymorphism—**`tri`** and **`rect`** are both subclasses (hence types of) abstract class `Shape`.

Exercise 3:

1. In line 1 `method1()` returns a `ClassA` object reference variable and assigns it to `obj1`, which is also a Class A object reference variable.
2. In line 2 `method2()` returns a `ClassB` object reference variable (which is a subclass type of Class A) and assigns it to a `ClassA` object reference variable. The type assignment is still compatible.
3. In line 3 `method1()` returns a `ClassA` object reference and attempts to assign it to a `ClassB` type variable. This is analogous to assigning a `double` value to an `int` variable.
4. In line 4 `method3()` returns a `ClassA` object reference value and attempts to assign it to a `ClassB` object reference variable. Although lines 3 and 4 look very much alike, the "incompatible types" mismatches are actually different because the value returned by `method3()` was originally a `ClassB` object type that was "broadened" to `ClassA` format. This latter point is important to understanding the difference between lines 5 and 6.
5. Line 5 converts the underlying object reference type (a `ClassB` object value that was broadened to a `ClassA` value) back to its original type though the use of a type cast.
6. Line 6 cannot type cast the `ClassA` object reference value returned by `method1()` as, unlike the object returned in line 5, the object returned in line 6 never was a `ClassB` object reference to begin with.

Exercise 4:

The answer to Exercise 4 is the same as the answer for Exercise 2 except that this time the result is obtained by polymorphic return types instead of polymorphic assignment statements.

Exercise 5

<APPLET CODE="HouseBuild.class" WIDTH=800 HEIGHT = 600> </APPLET>

```
import java.awt.*;

public class HouseBuild extends java.applet.Applet
{
   public void paint(Graphics g)
  {
          BuilderClass [] refArray = new BuilderClass[5];

          refArray[0] = new Framer();
          refArray[1] = new Windows();
          refArray[2] = new Door();
          refArray[3] = new Roof();
          refArray[4] = new Chimney();

          for (BuilderClass e: refArray)
              e.build(g);

    }// end paint
}// end class

abstract class BuilderClass
{
          abstract void build(Graphics g);
}// end BuilderClass
```

```
public class Framer extends BuilderClass
{
          public void build(Graphics g)
          {
             g.drawRect(250, 300, 200, 100)          ;
          }// end build
}// end class Framer
```

```
public class Windows extends BuilderClass
{
     public void build(Graphics g)
    {
          g.drawRect(280,340, 20, 20);
          g.drawRect(400,340, 20, 20);
     }// end build
}// end class Windows
```

```
public class Roof extends BuilderClass
{
     public void build(Graphics g)
    {
          g.drawLine(250, 300, 350, 200);
          g.drawLine(350, 200, 450, 300);
     }// end build
}// end class Roof
```

```
public class Door extends BuilderClass
{
          public void build(Graphics g)
          {
             g.drawRect(330, 360, 40, 40) ;
          }// end build
}// end class Door
```

```
public class Chimney extends BuilderClass
{
          public void build(Graphics g)
          {
             g.drawLine(300, 250, 300, 200);
             g.drawLine(300,200, 330, 200);
             g.drawLine(330, 200, 330, 218);
          }// end build
}// end class Framer
```

Exercise 6

1. *Type in and execute given code.*
2. Errors 1 (`ArrayIndexOutOfBoundsException`), 2 (`IndexOutOfBoundsException`) and 3 (`RuntimeException`) will all be caught by the `RuntimeException` object. This is analogous to catching all three fish at once with the fine-mesh net. This catch is nonselective and does not allow for recovery code specified to correct specific error situation.
3. Here the errors are caught one at a time by the polymorphic exception catches. This allows for recovery code written to correct specific error situations as they occur.