**Polymorphism**

Normally, in Java, when declaring and initializing variables that hold objects, we use the same data type on both sides of the equals sign:

**SimpleCanvas** canvas = new **SimpleCanvas**();

**ArrayList<String>** list = new **ArrayList<String>**();

**Scanner** scanner = new **Scanner**(System.in);

However, polymorphism allows us to replace a base class object with a derived class object:

public class Base { }  
public class Derived extends Base { }  
  
**Base** baseObject = new **Derived**();

When this occurs, the variable baseObject has two different types: we will talk about the *declared type*, which is the data type that the variable is declared as in the code, and the *actual type*, which is the type of the object that is actually stored in the computer’s memory. In the line of code above, the declared type of baseObject is Base, whereas the actual type is Derived.

The declared type of an object controls what Java believes the object is capable of doing, whereas the actual type determines the behavior of the object.

Example:

public class Base {  
 public void f() { System.out.println(“Base f”); }  
 public void g() { System.out.println(“Base g”); }  
}  
  
public class Derived extends Base {   
 public void f() { System.out.println(“Derived f”); }  
 public void h() { System.out.println(“Derived h”); }  
}  
  
Base baseObject = new Base();

Derived derivedObject = new Derived();

Base polymorphObject = new Derived();

baseObject.f(); Prints base f  
baseObject.g(); Prints base g

derivedObject.f(); Prints derived f

derivedObject.g(); Prints base g

derivedObject.h(); Prints derived h

polymorphObject.f(); Prints derived f

polymorphObject.g(); Prints base g

// polymorphObject.h(); Illegal! Cannot call a derived-only method on an object declared as a base type.

But we can cast base-class-declared objects to a derived-class object (provided the actual type is Derived)

((Derived)polymorphObject).h(); Prints derived h

The substitution that polymorphism lets us do – substitute a derived object where a base object is needed – typically is seen in four places in code:

* Variable declarations (as seen above)
* Passing an object to a function/method
* Returning an object from a function/method
* Storing an object in another object or data structure (like in an ArrayList)

**Passing an object to a function/method:**

void function(Base baseObject) {

baseObject.f(); // may print Base f or Derived f

baseObject.g(); // prints Base g  
 // baseObject.h(): // always illegal (unless you use a cast)

}

Base baseObject = new Base();

Derived derivedObject = new Derived();

function(baseObject); // fine

function(derivedObject); // fine

**Returning an object from a function/method:**

Base function() {

Base b = new Base();

Derived d = new Derived();  
 // return b or d, Java will let either one be returned

}

Base baseObject = function(); // legal, we just don’t know if baseObject’s  
 // actual type is Base or Derived.  
baseObject.f(); // may print Base f or Derived f

baseObject.g(); // prints Base g

// baseObject.h(): // always illegal (unless you use a cast)

**Storing an object in another object or data structure (like in an ArrayList):**

ArrayList<Base> list = new ArrayList<Base>();  
  
Base baseObject = new Base();

Derived derivedObject = new Derived();

list.add(baseObject);

list.add(derivedObject);

for (int i = 0; i < list.size(); i++) { // or use enhanced for loop syntax  
 Base b = list.get(i);  
 baseObject.f(); // may print Base f or Derived f

baseObject.g(); // prints Base g

// baseObject.h(): // always illegal (unless you use a cast)  
}