2. Polymorphism :: Dynamic Method Lookup

Furthermore, consider calls to *aMethod*() which is declared in *Super* and overridden in *Sub*:

```
public void someMethod(Super pObj) {
  pObj.aMethod();
}

public void someOtherMethod(Sub pObj) {
  pObj.aMethod();
}
```

Which *aMethod*() gets called?

1. In *someMethod*() when *pObj* is a *Super*, will call *Super.aMethod*().
2. In *someMethod*() when *pObj* is a *Sub*, will call *Sub.aMethod*().
3. In *someOtherMethod*() when *pObj* is a *Sub*, will call *Sub.aMethod*().
4. It is illegal to pass a *Super* object to *someOtherMethod*().

The rule in Java is:

> **The overridden method that gets called is the one associated with the class of the actual object that the object variable refers to, *not* the class of the object variable.**

Therefore, in Item 2 when the object variable *pObj* contains a reference to an object which is actually a *Sub*, the *aMethod*() that gets called will be the one associated with *Sub*. In Java this rule is known as **dynamic method lookup** (in programming languages, **dynamic** refers to something that happens happens at runtime).

2. Polymorphism :: What is Polymorphism?

Dynamic method lookup implements an object oriented programming feature known as **polymorphism** which occurs when:

1. A superclass *Super* declares a public or protected method *M*(). It does not matter if *Super* or *M*() are abstract or not:

```
public class Super {
  public void M() {
    doSomething();
  }
}
```

OR

```
public abstract Super {  // Super is abstract
  public void M();       // M() is abstract
}
```

2. Direct and nondirect subclasses of *Super* override *M*(), i.e., each subclass provides its own specific implementation of *M*() which does something that is specific to each subclass.

```
public class DirectSub extends Super {
  @Override public void M() {
    doSomethingDifferentThanSuper();
  }
}
```

```
public class NondirectSub extends DirectSub {
  @Override public void M() {
    doSomethingDifferentThanSuperAndDirectSub();
  }
}
```

2. Polymorphism :: What is Polymorphism (continued)?

3. A subclass object *obj* is passed as an argument to a method *N()* in which the method parameter *pObj* is declared as a *Super* object.

```
public void N(Super pObj) {  // pObj can refer to a Super object or to an object
  ...                        // of any subclass of Super, i.e., DirectSub or
}                            // NondirectSub

public void someMethod() {
  NondirectSub sub = new NondirectSub(); // Since a NondirectSub is a Super it is
  N(sub);                                // legal to substitute a NondirectSub
}                                        // object for a Super object
```

4. Within *N*() a call to *pObj.M*() is made.

```
public void N(Super pObj) {  // The class of the object variable pObj is Super
  pObj.M();                  // but the class of the object to which pObj refers
}                            // is NondirectSub
```

5. Since the *pObj* object variable is declared as an object of *Super* but the class of the object to which *pObj* actually refers is *NondirectSub*, then *NondirectSub.M*() is **polymorphically** called.

```
public void N(Super pObj) {
  pObj.M();  // This is a polymorphic method call to NondirectSub.M()
}
```

Polymorphism literally means "many forms" or "many shapes" and is used to refer to this behavior since the object variable parameter *pObj* declared in *N*() appears to take on many forms or behaviors.