

Method Overloading

For the same class we can define two methods with the same name. However the parameter types and/or the number of parameters must be different for those two methods. In the java terminology, this is called **method overloading**. It is useful to use method overloading when we need to do something different based on a parameter type. For example we may have the operation : runAroundThe. We can define two methods with the same name, but different input parameter type:

```
public void runAroundThe( Building block )
{
    ...
}
public void runAroundThe( Park park )
{
    ...
}
```

Related terminology is the **method signature**. In java the method signature contains method name and the input parameter types. The java compiler takes the signature for each method and makes sure that each method signature is unique for a class. For example the following two method definitions are valid:

```
public void logIt( String param, Error err )
{
    ...
}
public void logIt( Error err, String param )
{
    ...
}
```

Because the type order is different. If both input parameters were type String, that would be a problem again since the compiler would not be able to distinguish between the two:

```
public void logIt( String param, String err )
{
    ...
}
public void logIt( String err, String param )
{
    ...
}
```

The compiler would give an error for the following method definitions as well:

```
public void logIt( String param )
{
    ...
}
public String logIt( String param )
{
    String retType;
```

```
...
    return retValue;
}
```

Note, the return type is not part of the unique signature. Why not? The reason is that a method can be called without assigning its return value to a variable. This feature came from C and C++. So for the call:

```
{
    logIt ( msg );
}
```

the compiler would not know which method to call.

Method Overriding

Obviously a method signature has to be unique inside a class. The same method signature can be defined in different classes. If we define a method that exist in the super class then we override the super class method. The terminology for this is **method overriding**. This is different from method overloading. Method overloading happens with methods with the same name different signature. Method overriding happens with same name, same signature between inherited classes.

The return type can cause the same problem we saw above. When we override a super class method the return type also must be the same. In fact if that is not the same, the compiler will give you an error.

Method overriding is related dynamic linking, or runtime binding. In order for the Method Overriding to work, the method call that is going to be called can not be determined at compilation time. It will be decided at runtime, and will be looked up in a table.

```
{
1  MyClass obj = new SubOfMyClass();
2
3  MyClass obj = new MyClass();
4
5  obj.myMethod(); // -- During compilation, it is not known what
reference the 'obj' has, MyClass or SubOfMyClass
}
```

In the above example 'obj' reference has the type MyClass on both line 1 and line 3. However the 'obj' reference points two different objects. On line 1 it references SubOfMyClass object, on line 3 it references MyClass object. So on line 5 which method will be called, method define in MyClass, or the method that defined in its subclasses. Because the 'obj' reference can point to object and all its sub object, and that will be known only at runtime, a table needs to be kept with all the possible method address to be called.

Also another rule is that when you do an override, the visibility of the new method that overrides the super class method can not be reduced. The visibility can be increased, however. So if the super class method visibility is public, the override method can not be package, or private.

In the case of the exception the override method may throw can be the same as the super class or it can be one of that exception inherited class. So the common rule is that the override method must throw the same exception or it is any of its subclasses.

NOTE: A common mistake to think that if we can override methods, we could also override member variables. This is not the case, as member variables are not overridden.

```
{
1  MyClass obj = new SubOfMyClass();
2
3  MyClass obj = new MyClass();
4
5  String var = obj.myMemberVar;  // -- The myMemberVar is defined in
the MyClass object
}
```

In the above example, it does not count what object the 'obj' reference points to, because it was declared MyClass type on both line 1 and line 3, the variable in the MyClass object will be referenced. In real examples we rarely use public variables, but if you do keep in mind that Java does not support variable overriding.

Parameter Passing

We can pass in all the primitive data types or any object references to a method. An object cannot be passed to a method, only its references. All parameters (those are primitive types and object references) are passed by value. In other words if you change the passed in parameter values inside the method, that will have no effect on the original variable that was passed in. When you pass in an object reference to a method and then you change that inside the method, that will have no effect on the original object reference. However if you modify the object itself, that will stay after the method returns. Think about the object reference as a pointer to an object. If you change the object the reference points at, that will be permanent. For example:

```
1 {
2   int var1 = 10;
3   int var2 = 20;
4   ...
5   myMethod( var1, var2 );
6   ...
7   System.out.println( "var1="+var1 + "var2="+var2 );  // -- The
variable values did not change
8 }
9 ...
10 void myMethod( int var1, int var2 )
11 {
12   ...
13   var1 = 0;
14   var2 = 0;
15   ...
16 }
```

On line 7 the value of var1 is 10 and the value of var2 is 20. When the variables were passed in to the methods their values were copied. This is called passing the parameter by value. In java we do not represent an object directly, we represent an object through an **object reference**. You can think of an object reference as a variable having the address of the object. So the object reference is passed in by value, but the object itself is not. For example:

```
1 {
2   MyObjOne obj = new MyObjOne();
3   obj.setName("Christin");
4   ...
}
```