MESSAGE PASSING

Table of Contents

Message Passing

In an object-oriented program, objects cooperate with each other to produce the desired functionality. This implies that there is some form of communication occurring between the objects. In most object-oriented languages this communication is accomplished via messages and a messaging system. In figure 1, the two objects, `p` and `c`, are communicating with each other via the Java messaging system within the Java Virtual Machine (JVM).

Figure 1

In summary, an object-oriented program can be viewed as a set of communicating objects working together to provide a specified set of functionality.

The Structure of a Message

Messages have the following structure.

<ID> <selector> <arguments>

The *ID* identifies the recipient of the message, the *selector* defines the required action, and the *arguments* contain any information necessary for completion of the action. For example, consider sending a message to the painter to paint the house blue. The message can have the following structure.

***ID**:* *Painter*
***Selector**:* *Paint House*
***Arguments**:* *Colour Is Blue*

Message passing is expressed in Java via the **dot operator** (**.**). In fact, many of the well-known object-oriented languages, such as C++, use this operator. The previous message structure can now be rewritten as follows.

*<ID>**.**<selector>(<arguments>);*

This can be re-expressed as follows.

*<Name of Object>**.**<Name of Method>**(**<Parameters>**);***

Where the *ID* becomes the *name of the object*, the *selector* becomes the *name of the method* and the *arguments* become the *parameters*.

Consider the following example:

```
public class VRWordMain
{
        public static void main( String args[] )
        {
                Player p1, p2;

                p1 = new Player();
                p2 = new Player();

                p1.set( 35, 150 );  // send a message to p1

                p2.set( 25, 200 );  // send a message to p2
        }
} // VRWordMain
```

The class `VRWordMain` has sent messages to the objects `p1` and `p2`, indicating that they should change their age and height values (see figure 2).

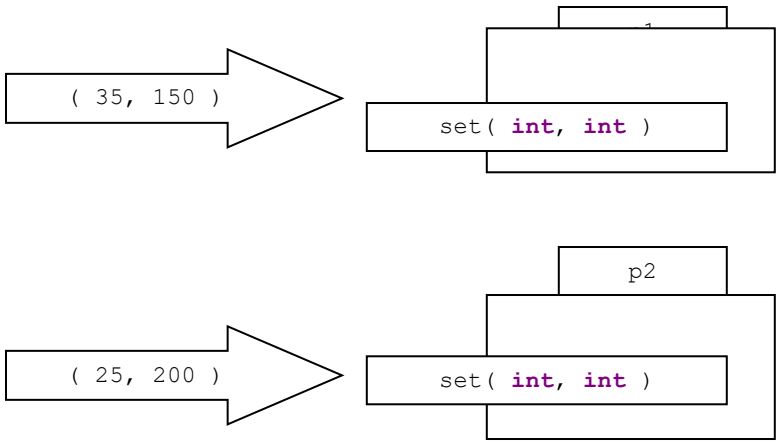

Figure 2

The Java syntax for message passing is similar to normal function calls thereby making it easier to understand and use.

Method Overloading

When an object receives a message, the *selector* is inspected and if a method with the same name exists within the object, the message is handled. However, the *arguments* complicate matters

since the message will be rejected if the formal parameters of the method do not have the same type and order as the *arguments*.

Consider the following example.

```
public class Player
{
	private int age, height;

	public void set( int newAge, int newHeight )
	{
		if( newAge > 0 )
			age = newAge;

		if ( newHeight > 0 )
			height = newHeight;
	} // set
} // Player

public class VRWordMain
{
	public static void main( String args[] )
	{
		Player p1, p2;

		p1 = new Player();

		p1.set( "35", 150 );		// send a message to p1
	}
} // VRWordMain
```

The call to the method `set( "35", 150 )` will not work since the first parameters is of type `String`, instead of `int`. The Java compiler will warn you of this problem as shown below.

```
VRWordMain.java:9: set(int,int) in Player cannot be applied to
(java.lang.String,int)
                p1.set( "35", 150 );    // send a message to p1
                  ^
1 error
```

This problem is resolved by *method overloading* where an object can respond to the same message selector in one or more ways i.e., an object can have one or more implementations for the same selector. Consider the following example:

```
public class Player
{
	private int age, height;

	public void set( int newAge, int newHeight )
	{
		if( newAge > 0 )
			age = newAge;

		if ( newHeight > 0 )
			height = newHeight;
	} // set
```

```
        public void set( String newAge, int newHeight )
        {
                int a;

                a = Integer.parseInt( newAge );

                set( a, newHeight );

                // This code can be optimised as follows
                //
                // set( Integer.parseInt( newAge ), newHeight );
        } // set
} // Player

public class VRWordMain
{
        public static void main( String args[] )
        {
                Player p1, p2;

                p1 = new Player();
                p2 = new Player();

                p1.set( "35", 150 );        // send a message to p1

                p2.set( 25, 200 );          // send a message to p2
        }
} // VRWordMain
```

Two methods called `set` have been provided in class `Player`. Since the parameter lists are different, they are distinct implementations of the same selector (see figure 3). Therefore, the method to be selected will depend on the actual parameter types. In this case, two calls to test have been made.

- The first call passes one `String` object and a single integer and so `set( String newAge, int newHeight )` will be used.
- The second call passes two integers and so `set( int newAge, int newHeight )` will be selected.

Method overloading occurs when a class has two or more methods with the same name but different parameter types.

Note that it is the parameter **types** and not the parameter **names** that distinguish one method from the other. Also, the **return** type is ignored. Consider the following code.

```
void setX( int x )
boolean setX( int x )
```

These two methods are considered the same, even though their return types are different.

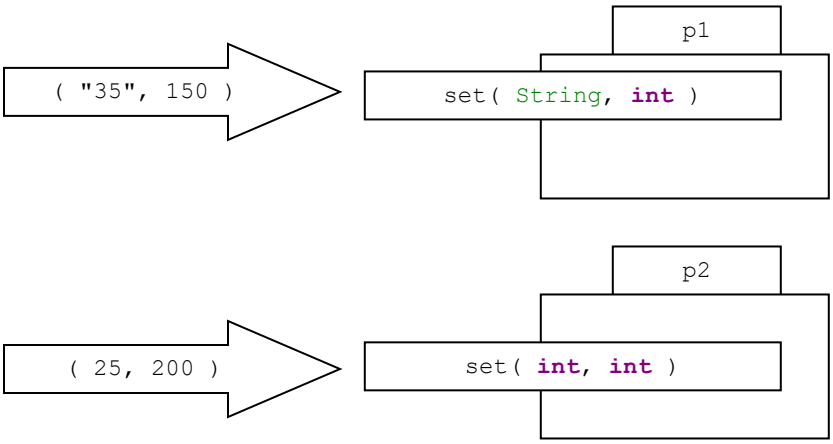

Figure 3

The main Method

To execute a Java program, there must be an entry point that the JVM can access. In Java this is called the **main** method. When the program is executed within the JVM, a message is sent to this method and the program then takes over from this point onwards. To simplify the JVM implementation, the return value and the parameter list of the **main** method have been standardised as follows.

```
public static void main( String args[] )
{
} // main
```

 Unlike C/C++, this declaration is fixed and must not be changed. Also, the **main** method **must** occur within a class.

The keyword **static** was described in the *Encapsulation* handout. The `args[]` parameter contains the list of arguments passed to the program at the command line.