http://www.python.org/dev/peps/pep-0253/). Template metaprogramming in C++ is a rather different approach: we use the fact that the C++ compiler will generate template code at compile time to carry out computation right then (Abrahams and Gurtovoy 2005). It is a technique with exciting possibilities that calls, however, for esoteric programming skills. In Java, metaprogramming, via reflection, is an integral part of the language, although Java code that uses reflection tends to be cumbersome.

A related issue is highlighted when we are faced with the problem of constructing a menu at runtime. A menu associates items with handlers that are invoked when the user selects the associated label. If we are able to refer to handlers by name, then we can construct a menu dynamically using code such as this:

```
CustomMenu new addList: #(
    #('red' #redHandler)
    #('green' #greenHandler)
    #('blue' #blueHandler)); startUpWithCaption:'Colors'.
```

In Smalltalk the delimiters `#( )` enclose arrays. We create a new menu with its items and handlers given by a list that contains label-handler pairs. The handlers are selectors (in real code, we would need to give the implementation for them). The handlers are prefixed by the character `#`, which denotes symbols in Smalltalk. We can think of symbols like strings. Symbols are typically used for class and method names.

Reflection allows us to implement the Abstract Factory design pattern in a succinct way (Alpert et al. 1998, pp. 43–44). If we want a factory class that instantiates objects of classes specified at runtime by the user, we can do the following:

```
makeCar: manufacturersName
    "manufacturersName is a Symbol, such as #Ford, #Toyota, or #Porsche."
    | carClass |
    carClass := Smalltalk
                    at: (manufacturersName, #Car) asSymbol
                    ifAbsent: [^ nil].
    ^ carClass new
```

After the user has given a manufacturer's name, we create the name of the class by appending the word `Car` to it. For `#Ford`, the class name will be `#FordCar`, for `#Toyota` it will be `#ToyotaCar`, and so on. Concatenation in Smalltalk is denoted by comma (`,`), and we want the resulting concatenated string to be made a symbol again, so we call its `asSymbol` method. All class names in Smalltalk are stored in the `Smallktalk` dictionary, the unique instance of the `SystemDictionary` class. Once we find the required class name, we return an instance of the class.

We have seen several examples of Smalltalk code, but not a class definition. In Smalltalk, classes are constructed in the same way as anything else: by sending the necessary message to the appropriate receiver. We start by completing the following template in the Smalltalk environment:

```
NameOfSuperclass subclass: #NameOfSubclass
    instanceVariableNames: ''
    classVariableNames: ''
```