Creators — observers — mutators

Problems with the terminology getter/setter

One usually hears this terminology: a *getter method* is a function that returns the value of a field (and changes nothing); a *setter method* is a procedure that stores a value in a field.

Some people automatically add getters and setters for all fields of a class. This can severely undermine the abstraction that a class behavior (as characterized by the set of public methods and their specifications) is designed for. You might as well make the fields public and remove the setters/getters. Resist the temptation to blindly insert getters/setters for all fields. Add them only if there is a good reason to do so, if they are actually part of the behavior you want of the class.

The getter/setter terminology has problems in that it influences the programmer to think that method specifications should mention fields, but fields are generally private and the user doesn't even know about them. Putting the specification "Set field x to parameter p." doesn't make sense from the user's point of view because the user doesn't know there is a field named x.

Specifications should be written from the caller's perspective, talking only about what the caller knows about the class. Here's an example. In class *javax.swing.JFrame*, procedure call *j.setTitle*(*String*) sets the title of the window associated with object j. The specification says just that; it does not mention the name of the field in which the title is stored (if there is one).

Also, refactoring to change the fields of a class may mean that a method is no longer a getter method, in that it doesn't retrieve the value of a field. Here's an example. Consider a class *Time* that has field *hour* for the hour of the day and field *minute* for the minute of the hour. We have getter method *getHour*(). We decide to replace the two fields by a single field *m* that contains the minute of the day, in the range 0..23*60-1. We have to change function *getHour*() accordingly, but it is no longer a getter method in the sense of returning the value of a field.

Creator/observer/mutator

Better is to consider methods to be in three categories: creators, observers, and mutators. This is based on the book "*Program Development in Java: Abstraction, Specification, and Object-Oriented*", by Barbara Liskov and John Guttag.

A *creator* is a method that creates (and returns) an object. There may be reasons for making all constructors private and providing a public function (or more than one) that returns an object of the class. That function could use a new-expression, thus creating an object, and return it. Note that a constructor is *not* a creator. Evaluation of a new-expression creates an object; all a constructor does is initialize fields.

An *observer* is a function that reports something about the state of the object without changing the state —it has no side effect. An example is function *size*() in class *ArrayList*. Function *size*() doesn't change anything; it just returns the number of elements in the *ArrayList*. Function *size*() need not be a getter function: we don't even know whether a field contains the number of elements or whether the size is obtained in another way.

Here's another possible observer function. Consider a class that maintains people's US social security numbers. Function *soc*() returns just the last four digits of the social security number, thus providing some security —the complete social security number is not available.

A *mutator* is a method that changes the state of an object (it may be a function or a procedure). Examples are procedure *setTitle*(*String*) in class *javax.swing.JFrame*. This function must do far more than change a field of an object (if it does that), for it must execute at least one call that puts the title on the associated window on the monitor.

Immutable classes (or objects)

Classes that contain no mutator methods are called *immutable* (or rather their objects immutable). *Immutable* means invariable, unalterable, not capable of change. Classes *String and* Integer are immutable. You cannot change the value of the string in a String object —but you can create new String objects. The advantage of immutable classes is that their objects can be shared freely by different code modules.