Kevlin Henney / kevlin@acm.org

Value added

THE TERM VALUE can refer to the desirability of a thing or to an amount. While I'm not sure that Java's type system is up to modeling desirability, we can readily see that calculating and storing amounts is a typical requirement of software.

Value objects[1] represent the simple, granular pieces of information you find in a system: strings, dates, money, dimensions, colors, etc. This is in contrast to entity objects—which capture the significant, persistent pieces of information real estate such as orders and customers—and to process and service objects such as Command objects,[2] servlets, and layout managers.

Patterns

A pattern language relates a set of practices. Patterns are arranged so they flow from one to another, with decisions taken to go from one to the next, forming a design around a particular model—in this case, value-based programming in Java. Figure 1 shows the patterns and their connections in the language.

Although the flow in the language is not linear, this column is. The following sections list the patterns in a kind of best-fit, partial ordering. Each pattern is presented in a brief problem/solution form, preceded by a sentence summarizing the intent. A simple date representation example runs throughout, but you can find these patterns applied in many Java systems, including the JDK.

Common Forces and Consequences

Common constraining forces and common resulting consequences are a feature of many of the patterns; other patterns work to offset these.

Values tend to make up a lot of the representation of and method traffic between other objects. As objects, this may lead to an increase in the number of classes at development time and the creation of more small objects at runtime. However, these consequences and their relative impact depend on the system being built. They must be weighed against an overall reduction in duplicated logic—and therefore system size and effect of change—and an increase in code clarity.

As part of other objects, values often require persistence; as part of remote method communication, they should be copied rather than stand as targets for independent communication. Implementing Serializable is a solution to both of these issues.

Threading is another common issue confronting the developer: To satisfy the principle of least astonishment, the class provider should ensure either that the effects of any modifier methods on classes are thread safe, or that class users are made aware of the lack of thread safety.

Whole Value

Provide distinct classes to express distinct domain specific value quantities clearly and safely.

Problem. How can you represent a primitive domain quantity in your system without loss of meaning? It is tempting, and indeed common, to represent value quantities in a program in the most fundamental units possible, e.g., integers and floating point numbers. However, this fails to communicate the understanding of the problem and its quantities into the solution,

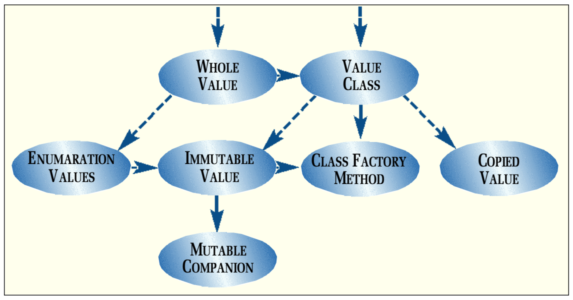

Figure 1. ***Patterns and their successors for supporting value-based programming in Java.***

Kevlin Henney is an independent consultant and trainer based in the UK. He can be contacted at kevlin@acm.org.

LISTING 1.

Year represented as a simple Whole Value.

```
public final class Year
{
    public static Year valueOf(int year)
        { return new Year(year); }
    public Year(int year) { value = year; }
    public int getValue() { return value; }
    private final int value;
}
```

and shows poor use of the checked type system.

Consider initializing a date object from the year, month, and day. The following code fragment arbitrates a more international (and logical) solution to the DD/MM/YYYY versus MM/DD/YYYY convention:

```
public final class Date ...
{
    public Date(int year, int month, int day) ...
    ...
}
```

However, an int is a pretty meaningless unit of currency: It does not define its units and it cannot be checked for sensible use. Compile-time errors should always be preferred to runtime errors—it can mean the difference between debugging before a release or after it:

```
Date right = new Date(year, month, day);
Date wrong = new Date(day, month, year);
Date alsoWrong = new Date(month, day, year);
```

Solution. Express the type of the quantity as a class. A Whole Value[3] recovers the loss of meaning and checking. A Whole Value class can range from a simple wrapper class—making the intent of the code clearer to both the human reader and the compiler—to one with more complete behavior. A Whole Value makes a method interface clearer and safer; whether it is used in the encapsulated implementation—as opposed to reverting to raw values—depends on other design choices and preferences.

Listing 1 demonstrates how a simple Year class elevates years to a checkable and named quantity.

Year could be made a little more sophisticated, acquiring the fuller trappings of a Value Class and some range checking. To make Date initialization safe, either one or both of months and days should also be represented by a Whole Value. Now, only the correct combination will compile:

```
new Date(Year.valueOf(year), Month.valueOf(month), day)
```

Whole Value allows us to enforce rules of combination that are not possible with simpler types such as int, double, or String. This can be likened to dimensional analysis in the physical sciences: It is plain nonsense to divide a measurement of distance by a measurement of temperature and assign the result as a measure of acceleration. A Whole Value can wrap up a more primitive type, e.g., PhoneNumber wrapping String; or bundle together simple attributes, e.g., Dimension wrapping ints for width and height.

Value Class

Define features for a class that represents a value type so that it conforms to expectations.

Problem. Having established that a value type needs implementing as a class, often because it is a Whole Value representing a domain specific quantity, what steps must be taken to define the class so that it meets user expectations?

The use of value objects revolves around their represented content rather than their state: Comparisons and ordering between value objects should depend on their content and not on their reference identity. This applies to their use as keys in associative collections as well as the more obvious comparisons for equality.

Solution. Override the methods in Object whose action should be related to content and not identity (e.g., equals) and, if appropriate, implement Serializable.

Field by field comparison is more appropriate for values than the default, identity-based Object.equals. Listing 2 shows equals defined with minimum casting and clutter.

If you override equals you should also override hashCode, ensuring that if two objects compare equal so do their hash values. Related to equality comparison is general relational comparison. Not all values can be meaningfully compared, e.g. Color, but where it makes sense, e.g., Date, provide a compareTo method. If you are using JDK 1.2 you can publicize this feature more explicitly by implementing the Comparable interface.

Value objects are normally the kind of object that should provide a printed form. Overriding Object.toString allows class users to take advantage of simple string concatenation syntax:

```
System.out.println("The date today is " + Date.today());
```

LISTING 2.

The implementation of equals for the Date Value class.

```
public final class Date ...
{
    ...
    public boolean equals(Object rhs)
    {
        return rhs instanceof Date && equals((Date) rhs);
    }
    public boolean equals(Date rhs)
    {
        return rhs != null && rhs.year == year &&
            rhs.month.equals(month) && rhs.day == day;
    }
    private int year, day;
    private Month month;
}
```

Creating instances of a Value Class can be simplified by providing Class Factory Methods as well as, or instead of, public constructors.

Enumeration Values

Represent a distinct and bounded set of values as objects.

Problem. How can you represent a fixed set of constant values and preserve type safety? It is commonly the case that a fixed set of options or indicators need representation. Even if they have no specific values they must still be distinct from one another.

For example, assuming you count months from January, do you number from 0 or from 1? A good case can be made for either one, which is exactly the problem with raw, meaningless ints. That said, there is a question of consistency over any API that thinks it is reasonable to start months from 0 and days from 1. Given that Java defaults int fields to 0—potentially hiding initialization problems—it seems best to opt for the 1 convention.

Now that the decision has been made, how can it be enforced? The simplest way to communicate this intent with the user is to use named int constants. Although this is preferable to inviting users to plug in magic numbers, it is still open to intentional or accidental abuse: int is checked only as an int and not as a month. Modeling month as a Whole Value offers a more expressive approach, but does not necessarily constrain the set of values to the correct ones.

Solution. Treat each constant as a Whole Value instance, declaring it as public static final in the Whole Value class. To e sure that the set of constant values remains fixed, prevent public construction by defining the constructor as private. Implementing the class as an Immutable Value prevents the possibility of the constant values being unexpectedly nonconstant.

Users of the Month class can now both name the month of their choice and receive the full support of the compiler's type checking. Also, because the type implicitly restricts the Enumeration Values, range checking is free.

Where the primitive value associated with each of the Enumeration Values has special significance—which is the case for months—the initialization of each constant should be explicit and fully under the control of the class developer (see Listing 3). Otherwise a no-arg constructor and a static counter can be used to assign each constant value a distinct number based on order of initialization.

LISTING 3.

Months **expressed as** Enumeration Values**.**

```
public final class Month ...
{
   public static final Month january = new Month(1);
   ...
   public static final Month december = new Month(12);
   public int getValue() { return value; }
   ...
   private Month(int month) { value = month; }
   private final int value;
}
```

LISTING 4.

A Class Factory Method **that converts from** int **to** Month**.**

```
public final class Month ...
{
   ...
   public static Month valueOf(int month)
      { return months[month - 1]; }
   private static final Month[] months =
      { january, ..., december };
   ...
}
```

Class Factory Method

Provide a convenient way to create an object that encapsulates the details of creation.

Problem. How can you simplify, and potentially optimize, construction of Value Class objects in expressions without resorting to intrusive new expressions? How do you provide for the opportunity to reuse existing objects that have the same value, i.e., Flyweight objects?[2]

Assuming that year and month are represented by Whole Value objects, the following code to initialize a Date looks syntactically clumsy and does not support the use of Enumeration Values for months:

```
Date clumsy = new Date(new Year(year), new Month(month), day);
```

Solution. Provide static methods to be used instead of (or as well as) ordinary constructors. The methods return either newly created Value Class objects or existing objects from a lookup table (see Listing 4).

For value types with a clear and commonly used string representation, it is common to provide a Class Factory Method that takes a String and returns a value object based on the parsing of that string. For example, Month may provide a conversion from the month name to the relevant Enumeration Value.

Even when it does not offer a lookup optimization, a Class Factory Method has benefits in promoting uniform usage and syntactic simplification. Class Factory Method is a common variation of the vanilla Factory Method pattern.[2]

Copied Value*

Describe how modifiable value objects should be passed around so as to avoid aliasing problems.

Problem. How can you pass a modifiable Value Class

*The previous article[3] referred to this as Cloneable Value, but the name Copied Value better captures the general intent of the pattern.

LISTING 5.

Passing a Copied Value **in and out of methods.**

```
class Order
{
  ...
  public void setDeliveryDate(Date newDeliveryDate)
    { deliveryDate = (Date) newDeliveryDate.clone(); }
  public Date getDeliveryDate()
    { return (Date) deliveryDate.clone(); }
  private Date deliveryDate;
}
```

object into and out of methods without permitting callers or called methods to affect the original object? Value objects are often used as attributes inside other objects. Subtle aliasing problems can arise if the attribute is assigned directly from a method argument when it is set or if it is returned directly when queried:

```
Order firstOrder, secondOrder;
...
Date date = new Date(year, month, day);
firstOrder.setDeliveryDate(date);
date.nextDay();
secondOrder.setDeliveryDate(date);
```

In essence, this is a violation of encapsulation: Although declared private, users can still interfere with the object's representation.

Solution. Implement the Cloneable interface or provide a copy constructor for the Value Class, and use a copy of the original whenever it needs to be passed (see Listing 5). This prevents sharing and preserves encapsulation, while still allowing value objects to be modified.

For values that are queried far more often than they are modified, an Immutable Value with a Mutable Companion offers an alternative leading to less object creation.

Immutable Value

Define a class for which instances have fixed state to avoid aliasing problems.

Problem. How can you share Value Class objects and guarantee no side effect problems? Copied Value describes the conventions for using a modifiable value object to minimize aliasing issues. However, this can be error prone and may lead to excessive creation of small objects, especially where values are frequently queried or passed around.

If objects are shared between threads they must be thread safe. Synchronization of state change incurs an overhead, but is essential in guarding against race conditions. Even where copying or synchronization are carefully attended to, there are still opportunities for undefined behavior: The integrity of an associative collection may be compromised if the state of an object used as a key is modified via an aliasing reference.

Solution. Set the internal state of the Value Class object at construction, and allow no subsequent modifications; i.e., provide only query methods and constructors, as shown in Listing 6. Declaring the fields final ensures this no-change promise is honored. This guarantee implies also that either the class itself must be final or its subclasses must also be Immutable Values.

The absence of any possible state changes means there is no reason to synchronize. Not only does this make Immutable Value objects implicitly thread safe, the absence of locking means that their use in a threaded environment is also efficient.

Sharing of Immutable Value objects is also safe and transparent in other circumstances; so, there is no need to copy an Immutable Value, and therefore no reason to implement Cloneable. Attributes are changed by replacing their referenced Immutable Value object with another holding a different value, rather than modifying the object that is already there.

There are complementary techniques for creating Immutable Value objects: Provide a complete and intuitive set of constructors; provide a number of Class Factory Methods; or provide a Mutable Companion.

Mutable Companion

Define a companion class to simplify complex manipulation of Immutable Value objects.

Problem. How can you simplify complex construction of an Immutable Value? Constructors provide a way of creating objects from a fixed set of arguments, but they cannot accumulate changes or handle complex expressions without becoming too complex, e.g., calculate the date 15 working days from today. Such requirements typically lead to expressions that create many temporary objects.

LISTING 6.

Date **expressed as an** Immutable Value**.**

```
public final class Date ...
{
  public Date(Year year, Month month, int day)
  {
    this.year  = year.getValue();
    this.month = month;
    this.day   = day;
  }
  public int getYear() { return year; }
  public Month getMonth() { return month; }
  public int getDayInMonth() { return day; }
  ...
  private final int year, day;
  private final Month month;
}
```

LISTING 7.

The DateManipulator Mutable Companion **to** Date**.**

```
public class DateManipulator ...
{
   public synchronized void set(
      Year year, Month month, int day)
   {
      this.year = year;
      this.month = month;
      this.day = day;
   }
   public synchronized Date toDate()
      { return new Date(year, month, day); }
   public int getYear() { return year.getValue(); }
   ...
   private Year year;
   private Month month;
   private int day;
}
```

Class Factory Methods may be able to optimize some of the creation and expressiveness issues, but they are still one off creation requests.

Solution. Implement a companion class that supports modifier methods and acts as a factory for Immutable Value objects. For convenience, the factory can stand not only as a separate class, but can also take on some of the roles and capabilities of the Immutable Value, e.g., StringBuffer and String. The modifiers, which should be synchronized, allow for cumulative or complex state changes, and a query method allows users to get access to the resulting Immutable Value (see Listing 7).

Although it may support many of the same methods as its associated Immutable Value, it is not related to it by inheritance: It cannot fulfill the same guarantees on immutability and therefore cannot be considered a subtype, hence its nonfamilial status as a companion. ■

References

1. **http://c2.com/cgi/wiki?ValueObject**
2. Gamma, E. et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison–Wesley, 1995.
3.Henney, K., "Patterns of value," *Java Report*, Vol. 5, No. 2, Feb. 2000, pp. 64–68.
4. Cunningham, W., "The CHECKS Pattern Language of Information Integrity," *Pattern Languages of Program Design*, J. Coplien and D. Schmidt, Eds., Addison–Wesley, 1995.

Effective Java / Steve Ball

Continued from page 72

column I'll address the first, by showing how to explicitly disable cloning for classes that have inherited a clone method with no checked exceptions.

The second trouble spot derives from a defect in the original design of the clone method: It is not possible to clone objects through Object references because the clone method is declared protected in Object and, furthermore, is not accessible through the Cloneable interface. Java's designers openly acknowledge the difficulties this imposes, stating,

"The sad fact of the matter is that Cloneable is broken, and always has been. The Cloneable interface should have a public clone method in it. This is a very annoying problem, and one for which there is no obvious fix."[5]

The inaccessibility of the clone method via Object references has implications for performing deep copying on container objects. Generic Object-based containers are prevented from cloning their elements because the clone method is inaccessible. In the words of the Java language developers again, "In fact, it's nearly impossible to do deep copies in Java, as the Cloneable interface lacks a public clone operation."[4]

Java's cloning mechanism has numerous defects and is unnecessarily awkward and error prone.[1] Not only that, but cloning has been even more "broken" by the addition of new language features such as final members and inner classes, with apparent disregard for their impact on a facility that already presents a number of problems.

Missing features in the language also make cloning more bothersome than it could be: Java does not support *covariant return types*, where an overriding method is permitted to return a subclass of the type returned by the method it overrides. This requires callers of the clone method to cast the result to retrieve the type information that was lost when the clone method returned.

The clone method is not supported for many common classes for which it would be useful: notably String and the wrapper classes Integer, Float, etc. Cloning instances of these classes through generic Object references is unnecessarily irregular.

Fortunately, all these problems may be overcome. Providing cloning support is an important part of designing classes of maximum usefulness and reliability. In a future column on cloning, I will present a mechanism that allows objects to be cloned via Object references, one that can be added to any of the Java API container classes that will cause them to perform a deep copy when cloned. You will then have all the weaponry you need to beat Java at its own game. ■

References

1. Ball, S., "Effective Cloning," *Java Report*, Vol. 5, No. 1, Jan. 2000, pp. 60–67.
2. Ball, S. "Supercharged Strings," *Java Report*, Vol. 4, No. 2, Feb. 1999, pp. 62–69.
3. Gosling, J., B. Joy, and G. Steele, *The Java Language Specification*, Addison–Wesley, 1996.
4. Sun Microsystems Bug Parade, **http://developer.java.sun.com/developer/bugParade/bugs/4103477.html**
5. **http://developer.java.sun.com/developer/bugParade/bugs/4228604.html**
6. Source code for the example classes, as well as the benchmarking application for timing the relative efficiency of construction vs. cloning, is available at **http://effectivejava.com/column/d-cloning**