COMP 110

Prasun Dewan[1]

9. Loan Case Study

In the representation section, we have studied a number of concepts, including testing, class members, structured objects, and polymorphism. These concepts were illustrated through objects which, by following certain programming patterns, enabled ObjectEditor to display them graphically. In this section, we gain more experience with these concepts through a case study.

Two-Way Dependencies

Let us move away from geometric objects to gain more experience with objects. Let us consider an object that represents a loan. Figure 1 shows how it is displayed and manipulated in the main edit window.

As the figure shows, the object has three `int` properties, which specify the loan amount or principal, the interest to be paid annually on it, and the monthly interest on it. These are not independent properties but different ways in which the loan can be viewed. In other words, given the value of one property, we can determine the values of the other two properties. All three properties are editable and changing one automatically causes corresponding changes in the other. This is shown by the three edit windows of Figure 1: in each window the selected property was set by the user and the other two properties were automatically computed. Thus, in the leftmost window, we entered the value of the principal, which resulted in automatic computation of the other two properties; and in the other two windows, we entered the values of other two properties.

This application is like a standard spreadsheet in that it defines dependencies among data items. In a standard spreadsheet, and the examples we saw in Chapter 2, the dependencies are one-way, that is, an item is either a dependent item or an independent item, and all dependencies are from dependent items to independent items. A dependent item cannot be changed directly; it is always changed indirectly by changing the items on which it depends. For instance, in the BMI example, the independent editable items are weight and height, and the dependent (read-only) item is the BMI value. In this example, on the other hand, the dependencies are two-way: each of these items is editable and depends on the other two items. Figure 2 shows the difference between one-way and two-way constraints.

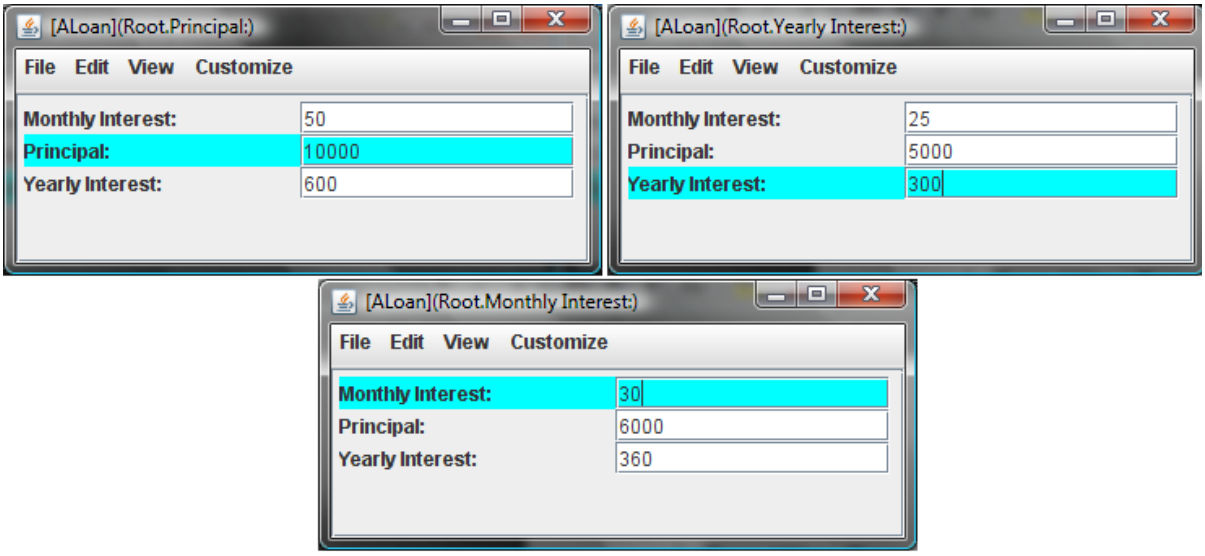


Figure 1. ALoan with three editable properties

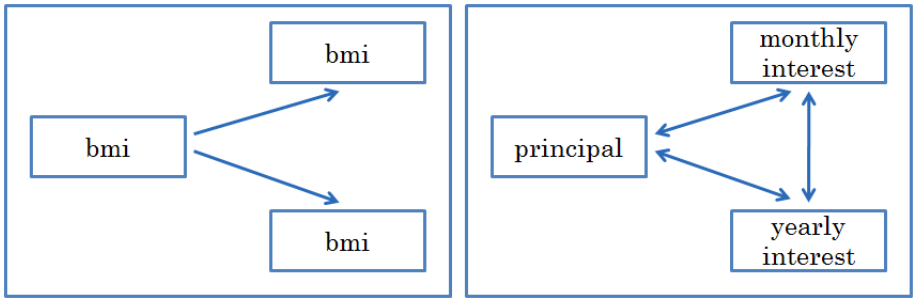


Figure 2. One-Way vs. Two-Way Dependencies

Top-down Programming

As it turns out, implementing the two-way constraints of this problem is only slightly more complicated than implementing the one-way constraints we saw earlier. However, this time, we will use a different approach to developing our solution. In the BMI example, we used the *bottom-up* approach of first creating the classes and then extracting the common interface implemented by them. Here, we will try to use the *top-down* approach of first creating the interface and then deriving implementation from it.

Level 1: Interface

The interface of the loan object can be derived from its edit windows displayed in Figure 1. Since all three properties are editable, we must define a getter and setter method for each of them, giving us the following interface:

```
public interface Loan {
      public int getPrincipal();
      public void setPrincipal(int newValue);
      public int getYearlyInterest();
      public void setYearlyInterest(int newVal);
      public int getMonthlyInterest();
      public void setMonthlyInterest(int newVal);
}
```

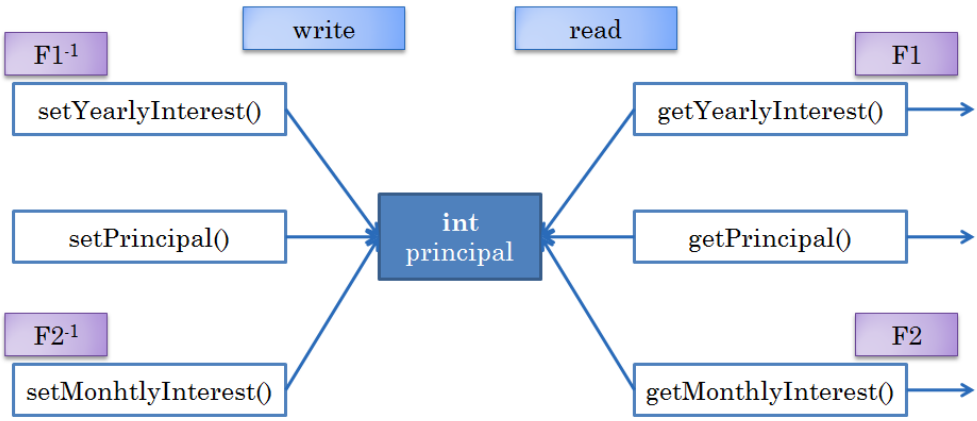


Figure 3. One-Way vs. Two-Way Dependencies

The type of these properties as `int` rather than `double` since we are interested only in whole number values for these properties, as shown in Figure 1.

Level 2: Representation

An implementation of the loan object must, as in the case of the BMI object, define one or more instance variables for storing the state of the object and use them to implement the methods specified in its interface. The set of instance variables that store the state of an object is called the *representation* of the object. As we saw in the BMI example, it is possible to create multiple representations for an object, depending on whether we want space efficiency or time efficiency. Let us put the constraint of space efficiency for this problem. We must then decide the smallest number of instance variables that can hold the state of the object.

As we see in the edit windows of figure, the value any one of these properties is sufficient to derive the values of the other two properties. Thus, one `int` instance variable is necessary and sufficient to store the state of this object. Let us arbitrarily choose this variable to be the loan principal:

```
int principal
```

We will refer to properties as *stored* or *computed* based on whether they are stored directly in an instance variable or computed from other properties. Thus, assuming the representation above for a loan, the principal is a stored property while the yearly and monthly interests are computed properties. All read-only properties such as the BMI value have been computed properties. In this example we see that editable properties can also be computed properties.

Level3: Algorithm

Before we describe the code of the methods that access this variable, let us work out a high-level algorithm for implementing them. The getter and setter methods for the principal are like those for other stored properties we have seen before, simply setting and returning, respectively, the instance variable in which it is stored. The getter methods for the computed properties pose no new challenge either, simply evaluating formulae that compute their return values from the principal, much as the getter method for BMI evaluated a formula computing its return value from weight and height. The

tricky parts of the implementation are the setter methods of these two properties. Unlike the editable properties we have seen so far, these properties to do not have instance variables associated with them. What they must do, then, is set the instance variable for the principal property by calculating the inverse of the formulae computed by their getter methods, that is, calculating the principal from their formal parameters. Figure 3 illustrates our algorithm, assuming FI and F2 are formulae that map the principal to the yearly and monthly interest, respectively, and $F1^{-1}$ and $F2^{-1}$ do the reverse.

Level 4: Class

We are now ready to implement the algorithm as a Java class, `ALoan`:

```
public class ALoan implements Loan {
      int principal;
      public int getPrincipal() {return principal;}
      public void setPrincipal(int newVal) {principal = newVal;}
      public int getYearlyInterest() {return principal*INTEREST_RATE/100;}
      public void setYearlyInterest(int newVal) {
            principal = newVal*100/INTEREST_RATE;}
      public int getMonthlyInterest() {return getYearlyInterest()/12;}
      public void setMonthlyInterest(int newVal) {
            setYearlyInterest(newVal*12);}
}
```

As we write the getter methods of the two computed properties, we see the need for the named constant, `INTEREST_RATE`. Where should we define it? We cannot define it in a particular method of this class, since multiple methods access it. The obvious thing to do, it seems, is declare it outside all methods as an instance variable of the class. Before we commit to this decision, let us look towards the future and anticipate other implementations of the Loan interface that use different representations of the loan. All of these implementations should use the same value of this constant. Therefore, it really belongs to the interface Loan, rather than this class.

This is the first example in which the public methods are not just called from outside the class, by ObjectEditor, but from inside the class to, implement other public methods. For instance, the method `getYearlyInterest` is called by `getMonthlyInterest`. The latter could have computed the yearly interest directly from the instance variables:

```
public int getMonthlyInterest() {
      return principal*INTEREST_RATE/(100 * 12);
}
```

but this results in repeating code for calculating the monthly interest. Thus, when we write a method, we should try and see if part of its code is already implemented by another method. Often, we think of public methods as being independent of each other, and miss out on such reuse.

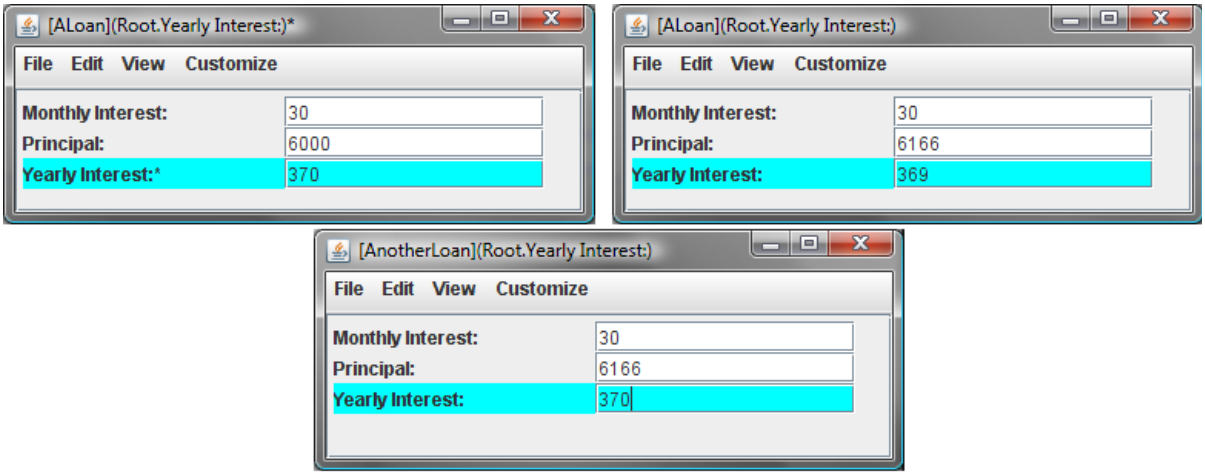


Figure 4. ALoan vs. AnotherLoan: (top-left) editing Yearly Interest in ALoan and (top-right) pressing Enter; (bottom) setting Yearly Interest in AnotherLoan

Mixing Top-down and Bottom-up Programming

Let us go back then and change the interface to add this constant:

```
public interface Loan {
      public final int INTEREST_RATE = 6;
      …
}
```

We must make it public to allow it to be visible in the classes that implement it. In fact, recall that all declarations in an interface must be public.

Going back and changing the interface after defining the class that implements it is inconsistent with our goal of top-down programming, since in top-down programming, the specification comes before the implementation. As this example shows, a pure top-down approach is very difficult to use, since it implies we understand the problem perfectly before we start implementing it. In fact, this is rarely the case, and as we have seen here, what we actually end up taking is an approach that mixes top-down and bottom-up programming, sometimes called *middle-out* or *iterative programming*.

We will see more examples of such programming when we further refine the interface and class.

Conversion Errors

In our implementation of the interface, could have used either the yearly interest or monthly interest as the representation of a loan. Interestingly, though, in this example, the representation we choose influences the exact functionality offered by the implementation.

Assume that the class `AnotherLoan` makes yearly interest the stored property and the other two properties the computed properties. Figure 4 shows the difference between the behaviors of `ALoan` and `AnotherLoan`.

Figure 4 (top-left) and (top-right) show interaction with an instance of `ALoan`, while Figure 4 (bottom) shows interaction with an instance of `AnotherLoan`. Figure 4 (top-left) and (top-right) show effect of editing Yearly Interest and pressing Enter, respectively. Notice that the value entered, 370, for this property is changed by the object to 369. This is because the following sequence of actions occurs when Enter is pressed:

1. The setter method for the property, `setYearlyInterest` is called, which converts its argument, 370, to the principal valyue 6166, using the forumala given above. Since the formula involves integer division, there is truncation of the result.
2. Now the getter methods of the two other properties, including the one we edited, are called to refresh the values of all properties. When `getYearlyInterest` is called, it converts the value principal value, 6166, back to an yearly interest value, using the formula given above. Again, because of integer division, there is truncation of the result.

Thus, there are two truncations, one in converting from the yearly interest to the principal, and one in the other direction, resulting in the set by the setter method of the property to be different from the value returned by its getter method. This, in turn, causes the entered value of this property to be different from the one used to refresh it. Thus, the yearly interest cannot effectively be set to certain values, such as 370 in this example.

This problem does not occur in Figure 4 (bottom), which shows the final result after entering the value 370 for yearly interest and pressing Enter. The entered value is maintained, since, in this version of the implementation, the yearly interest is a stored property rather than a computed one. On the other hand, in this implementation, the principal cannot be set to certain values, since it is a computed rather than stored property.

Thus, we may choose different representations, not just to tradeoff space for efficiency, but to tradeoff accuracy in one property for accuracy in another. For instance, if principal is the “primary” property , that is, it is the one whose value we wish to set precisely, we should use `ALoan`. If, on the other hand, yearly interest is the primary property, then we should use `AnotherLoan`.

A corrollary of this discussion is that if we are going to create a single representation, we must choose it wisely since it effects efficiency and nature of conversion errors.

Now that we completely understand the loan example, consider a related example motivated by the fact that often we have more than one loan to worry about - typically, a house loan and a car loan. It would be useful to display the two loans and their sum in a single window, and edit the two loans to do what-if calculations, as shown in the edit window of Figure 5.

Let us consider how we would define the object behind this edit window. So far, it was straightforward to derive the properties of an object once we had seen its edit window. This time, however, there is some ambiguity, as shown in Figure 5. We can create nine int properties of the object (one for each of the text fields of Figure 5) representing the car loan principal, car loan yearly interest, car loan monthly interest, house loan principal, and so on. Or we can create three properties, each of them a loan object,

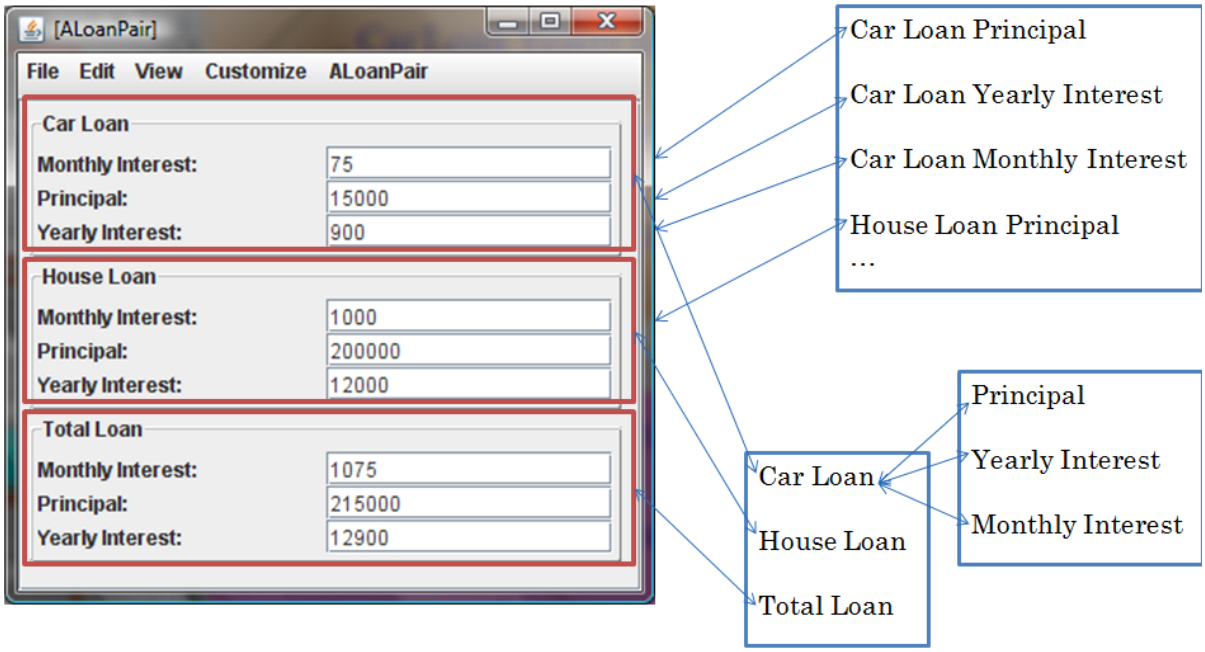


Figure 5. Primitive Vs Object Properties

which in turn has the three properties we saw before. Thus, in the first approach, we decompose the object into nine primitive properties, and in the second approach, we decompose it into three object properties.

The first approach may seem more comfortable, since it creates only primitive properties, which is what we have done in the examples so far. However, while this approach worked well for the previous examples, it would not be as suitable this time, because this example is far more complex. We would end up end up creating many more primitive properties – nine instead of the two or three properties we have created so far. Thus, we would end up doing fairly tedious coding, writing nine pairs of getter and setter methods. More important, this coding would be repetitive, as the principal, yearly interest, and monthly interest properties would be defined three times - once for each of the three loans.

The second approach seems to offer the hope of less coding. We have already written code to display and edit a single loan. If we decomposed our new object into three loan objects, rather than directly into the nine individual int values, then we can reuse our existing code. Visually, the edit window of this new object can be considered a composition of the three loan-object edit windows. The second approach allows us to code this example by simply composing existing code we have written to implement the loan object. Therefore, we will use this approach.

To use this approach, we must learn how to create properties that are objects. Like primitive properties, object properties are specified by getter and setter methods whose names are derived from the names of the properties. The tricky part is determining what to specify as the types of these properties. Since both classes and interfaces serve as types of objects, in this example, we must decide between the two classes `AnotherLoan` and `ALoan`, and the interface `Loan`, all of which type loan objects.

As mentioned before, in general, it is better to type a property or variable by an interface rather than a class because it lets us postpone our decision on the class of the object to which the property or variable is set. In this example, if we make the type of the car loan property the interface Loan, then we can set it either to an instance of `ALoan` or an instance of `AnotherLoan`. If, on the other hand, if we make its type, `ALoan`, then we can set it only to an instance of `ALoan`, not `AnotherLoan`, even though both kinds of objects offer the same interface.

We can now declare the headers of the getter and setter methods of these three properties in the interface of the new object:

```
public interface LoanPair {
      public Loan getCarLoan();
      public void setCarLoan(Loan newValue);
      public Loan getHouseLoan();
      public void setHouseLoan(Loan newValue);
      public Loan getTotalLoan();
}
```

As we can see, the getter and setter methods for object properties are not different from those of primitive properties – the only issue was the type of these properties, which does not arise for primitive properties, since they are uniquely typed.

The next step is to decide on the representation of the object. Let us again try to be as space efficient as possible. Home loan and car loan properties are independent editable properties, while total loan is a readable property, with one-way dependencies on the other two properties. Thus, this object is much like the BMI object; two Loan instance variables, storing the two independent properties, are necessary and sufficient. We can declare them as follows:

```
Loan carLoan;
Loan houseLoan;
```

The getter and setter methods for the car loan and house loan properties simply read and write these variables:

```
public Loan getCarLoan() {
      return carLoan;
}
public void setCarLoan(Loan newValue) {
      carLoan = newValue;
}
public Loan getHouseLoan() {
      return houseLoan;
}
public void setHouseLoan(Loan newValue) {
      houseLoan = newValue;
}
```

The getter method of the total loan property is more complicated, and we will ignore it for now.

Notice that we did not initialized the two instance variables when we declared them. We did not other instance variables we have created so far either. For instance, the instance variable of class `ALoan` is declared as follows:

```
int principal;
```

We did not initialize the variable since we did not know of a suitable initial value, and relied on Java's choice of zero to be appropriate. We do not know initial values for the car loan and house loan either, and thus could rely on Java's choice of a default value for a `Loan` variable. As we saw before, if the type of the variable is an object type, Java does not store a default object of that type, and instead, stores the null value, which signifies the absence of an object value. Therefore, we must explicitly initialize the object variables.

```
Loan carLoan = new ALoan();
```

Programmer-defined Add

Let us now focus on the third property, total loan. Conceptually, this property is simply the addition of the other two properties:

```
public Loan getTotalLoan() {
  return houseLoan + carLoan;
}
```

The above code, however, will not work, because the + operator works does not know how to add two values of type Loan. Because this type was defined by us, it is our responsibility to also provide an operation to add two Loan objects:

```
public Loan add(Loan loan1, Loan loan2) {
      Loan retVal = new ALoan();
      retVal.setPrincipal(loan1.getPrincipal()
            + loan2.getPrincipal());
      return retVal;
}
```

The operation takes as arguments two loan objects, and returns a new loan object that is the "sum" of the two arguments, that is, an object whose principal, yearly interest, and monthly interest properties are the sum of the corresponding properties of `loan1` and `loan2`.

The variable, `retVal`, is a local variable whose value is returned by the method. Before returning it, the method initializes it to an instance of `ALoan`, and then sets its principal to the sum of the two principals, thereby also effectively summing the other two properties of the two loans, since all three properties are kept consistent in a loan object. We could have alternatively summed one of the other two properties:

```
public Loan add(Loan loan1, Loan loan2) {
      Loan retVal = new ALoan();
      retVal.setMonthlyInterest(loan1.getMonthlyInterest()
            + loan2.getMonthlyInterest());
      return retVal;
}
```

Moreover, we would have initialized `retVal` to an instance of `AnotherLoan` in either of the two implementations.

Constructors

Consider the code in the `add` operation.

```
public Loan add(Loan loan1, Loan loan2) {
      Loan retVal = new ALoan();
      retVal.setPrincipal(loan1.getPrincipal()
            + loan2.getPrincipal());
      return retVal;
}
```

In the method, there are separate statements for creating a loan object:

```
Loan retVal = new ALoan();
```

and initializing its state:

```
retVal.setPrincipal(loan1.getPrincipal()
      + loan2.getPrincipal());
```

Java's constructors allow us to combine these two steps in one statement, resulting in a much more succinct version of the add method that does not even require us to declare `retVal`:

```
public Loan add(Loan loan1, Loan loan2) {
      return new ALoan(loan1.getPrincipal()
            + loan2.getPrincipal());
}
```

To support the new version of the add method, we can add the following constructor to class `ALoan`:

```
public ALoan(int initPrincipal) {
      setPrincipal(initPrincipal);
}
```

The constructor takes as argument the initial value of the principal, and calls `setPrincipal` to set its value.

The constructor could take any of the three properties as a parameter – we have chosen to pass it the principal, because it is the primary property, that is, the property in terms of which we wish other

properties to be computed. By the same argument, the constructor for `AnotherLoan` should take the yearly interest as a parameter:

```
public AnotherLoan(int initYearlyInterest) {
      setYearlyInterest(initYearlyInterest);
}
```

Initialization using Setter Methods

Notice that in the example above (and the previous two examples of constructors), we called the setter method to initialize the instance variables. We could have initialized the variables directly:

```
public ALoanPair (Loan initCarLoan, Loan initHouseLoan) {
      carLoan = initCarLoan;
      houseLoan = initHouseLoan;
}
```

In these examples, it does not make a difference which approach we take. Consider, `AnotherBMISpreadsheet`, where it makes a difference. If we directly initialize the two independent variables, `weight` and `height`:

```
public AnotherBMISpreadsheet(
            double initHeight, double initWeight) {
      height = initHeight;
      weight = initWeight;
}
```

we do not initialize, the dependent variable `bmi`, which retains its default value. As a result, the first time we call `getBMI`, it will return this value. On the other hand, if we call the setter methods to initialize, as is done below:

```
public AnotherBMISpreadsheet(
            double initHeight, double initWeight) {
      setHeight(height);
      setWeight(weight);
}
```

the methods will automatically set `bmi` to a value that is consistent with `initHeight` and `initWeight`. The first time we call `getBMI`, it is this value that is returned, rather than the default value.

Thus, in general, it is better to invoke setter methods to initialize variables, since they automatically initialize variables that depend on the one being initialized.

Class Methods

In which class(es) should we implement the `add` operation. We can implement it in either the class `ALoanPair`, or the two classes, `ALoan`, and `AnotherLoan`. Just as the addition of two int values belongs to the implementation of int, rather than every class that use int, the addition of two `Loan`

objects belongs in a class that implement `Loan`, rather than every class, such as `ALoanPair`, that uses `Loan`. As it does not access any instance variable, it should be a class method of `Loan`:

```
public static Loan add(Loan loan1, Loan loan2) {
      Loan retVal = new ALoan();
      retVal.setPrincipal(loan1.getPrincipal()
            + loan2.getPrincipal());
      return retVal;
}
```

We can now invoke it from `ALoanPair` to complete our definition of `getTotalLoan`:

```
public Loan getTotalLoan() {
      return ALoanPair.add (houseLoan, carLoan);
}
```

Polymorphism

In the implementation of `add` we followed the principle of using interfaces rather than classes to type formal parameters. This example helps us better understand benefits of this principle. We can pass actual parameters to this method that are instances of both `ALoan` and `AnotherLoan`. Thus, the same method can be used to add two instances of `ALoan`, two instances of `AnotherLoan`, and an instance of `ALoan` and an instance of `AnotherLoan`. If we used classes to type these variables, we would need to create three different implementations of this operation, one for each of these three combinations:

```
public Loan add(ALoan loan1, ALoan loan2) {
      return new ALoan(loan1.getPrincipal() + loan2.getPrincipal());
}
public Loan add(AnotherLoan loan1, AnotherLoan loan2) {
      return new ALoan(loan1.getPrincipal() + loan2.getPrincipal());
}
public Loan add(ALoan loan1, AnotherLoan loan2) {
      return new ALoan(loan1.getPrincipal() + loan2.getPrincipal());
}
```

Thus, if we use interfaces as the types of variables, we write one piece of code to process instances of multiple classes. Such kind of reuse is called polymorphism; and an operation that has one or more formal parameters that can be assigned actual parameters of different types is called a polymorphic operation.

Interfaces are only one mechanism for supporting polymorphism. Later, when we will see that inheritance offers another such mechanism.

Polymorphism Vs Overloading

A polymorphic operation should be distinguished from an overloaded operation. In both cases, the operation can be applied to values of different types. The difference is that in the case of a polymorphic operation, one implementation of the operation is created for processing different types of values; while in the case of an overloaded operation, multiple implementations are created. Thus, our original,

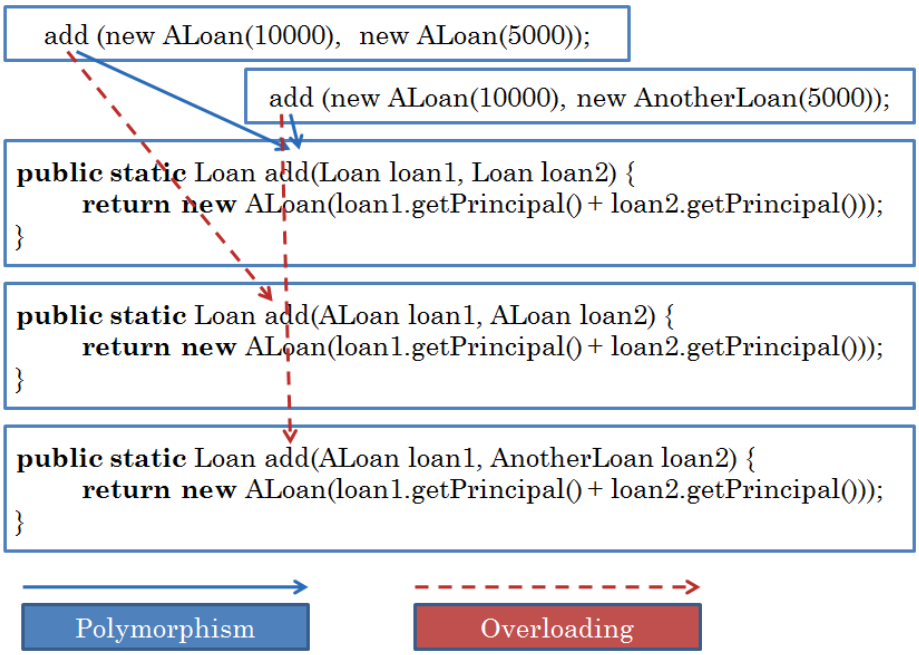


Figure 6. Polymorphism vs. Overloading

version of add is a polymorphic operation, while the new version is an overloaded one with three different implementations. We cannot tell, however, from how it is used, whether it is polymorphic or overloaded. For instance, in both cases, the following are legal calls:

```
add (new ALoan(10000), new ALoan(5000));
add (new AnotherLoan(), new AnotherLoan(5000));
add (new ALoan(10000), new AnotherLoan(5000));
```

We must look at the implementation to tell the difference (Figure 6).

Multi- Type Problems

We have seen in this chapter an example that simultaneously uses two different object types - the types `Loan` and `LoanPair`. As we saw, we could have used only one type, by creating a version of `LoanPair` that defines nine primitive properties instead of three Loan properties. We motivated the use of both types by arguing that defining `LoanPair` in terms of `Loan` allowed us to reuse existing implementations of the latter. What if the type Loan did not exist and we were asked to implement `LoanPair`? Since there is no existing code to reuse, does it mean we should define nine primitive properties in this class?

The answer, again, is no. By defining `Loan` specially for this application, we still get reusability, since we use the implementation of `Loan` three times in `LoanPair`, once for each of the three loans. If we were to directly implement loan calculation code in `LoanPair`, we would be repeating this code, once for each loan.

Put another way, by using `Loan` in `LoanPair`, we are using a *higher-level* type there, that is, a type that matches more closely the entity, loan, defined by the problem domain. As a result, we not only gain in reusability, but also other benefits of modularity such as code understandability.

It is tempting to define the smallest number of types required to implement an application, because of the thought required to identify each type. Because of the advantage mentioned above, it is important, in fact, to take the opposite approach of creating as many types as possible to solve a particular problem.

Steps in Top-Down Programming

Based on the discussion on the Loan example, we can now summarize some of the steps in the process of coding a top-down solution to a problem:

1. Identify as many of the types (interfaces) in the problem domain as possible.
2. For each type, identify the implementations that are needed to efficiently and accurately solve the problem.
3. For each implementation, identify a representation and associated algorithm.
4. Code the representation and algorithm in a class.

The number of steps we take depends on the complexity of the problem. For simple problems, we may omit steps 3 and 4, and directly code interfaces and classes. Also, the sequence in which we perform these steps depends on how well we understand the problem – the more we understand it, the higher the top-down component in our approach.

Summary

- It is possible for properties of an object to have two-way dependencies among them, especially when they are different views of a common state.
- It is possible to have multiple representations of such an object that are equally space efficient but calculate of the values of its properties to different accuracy.
- The process of solving a problem developing can be broken down into several stages: define one or more interfaces, and for each interface, develop one or more representations, algorithms, and classes.
- Top-down programming carries out these steps in the order given above, and bottom-up programming performs them in reverse order.
- Variables and properties should be typed by interfaces rather than classes so that they can be set to objects of different classes that implement the same interface.
- An uninitialized object variable is set to the null value, on which it is illegal to invoke any method.
- A class can define a constructor to initialize the instance variables of an object with values determined by the creator of the object.
- If the class does not define any constructor, Java automatically creates one for it.
- Each time a new object is instantiated, new copies of the instance variables declared in its class are created for the object.

- Class variables are shared by all class and instance methods of the class in which they are declared.
- Class (instance) variables and methods are called class (instance) members.
- Instance members are not visible in class methods and cannot be used in the expressions initializing class variables.
- Class members are visible throughout the class in which they are declared.

Exercises

1. Distinguish between read-only, editable, stored and computed properties.

2. Consider the following class:

```
public class StaticPractice {
      static int s = 0, n = 0;
      int a = 0;
      public StaticPractice (int v) {
            s = s + v;
            n = n + 1;
            a = s/n;
      }
      public static int getS() {
            return s;
      }
      public static int getN() {
            return n;
      }
      public int getA1() {
            return a;
      }
      public int getA2() {
            return s/n;
      }

}
```

   Classify the methods of this class into constructors, instance methods, and class methods. Also classify the variables of the class into instance and class variables.

   Suppose we create three instances of this class, passing to the constructors the values 10, 20, and 30, respectively. What will be the value of the four properties in each of these three instances? Use ObjectEditor to actually create the three instances. Refresh each edit window after all three windows have been created. Now verify if your calculations were right. Explain the values of the properties for each of the instances. (The names of the variables are deliberately obscure so that you trace the program rather than guess the results based on the names.)

3. Which of the methods of the various classes we saw in Chapter 2: `ABMICalculator`, `ABMISpreadsheet`, etc, could have been class methods?

4. Assume two implementations `DollarMoney` and `PoundsMoney`, of the following interface:

```
public interface Money {
      public int getPounds();
      public int getDollars();
}
```

DollarMoney (PoundMoney) has a constructor to initialize the money to a dollar (pound) amount; and the two methods above return this amount in pounds and dollars, respectively. Consider the following class, which is meant to represent the total money a person has in UK and USA. Identify and correct the errors and violations of style rules covered in this chapter. Also, classify the methods in the class into polymorphic and non-polymorphic, and overloaded and non-overloaded.

```
public class UK_USA_Assets implements Money {
      static PoundMoney assets;
      public UK_USA_Assets (DollarMoney usaAssets, PoundMoney ukAssets) {
            assets = add (usaAssets, ukAssets);
      }
      public DollarMoney add (DollarMoney arg1, DollarMoney arg2)  {
            return new DollarMoney (arg1.getDollars() + arg2.getDollars());
      }
      public DollarMoney add (DollarMoney arg1, PoundMoney arg2)  {
            return new DollarMoney (arg1.getDollars() + arg2.getDollars());
      }
      public DollarMoney subtract (Money arg1, Money arg2)  {
            return new DollarMoney (arg1.getDollars() + arg2.getDollars());
      }
      public static int getPounds() {
            return assets.getPounds();
      }
      public int getDollars() {
            return assets.getDollars();
      }
}
```

5. Extend the temperature interface of Chapter 4, problem 9, so that the Fahrenheit property is editable rather than read-only.

6. Implement the interface in a class, `ACentigradeTemperature`, that represents the temperature as a centigrade value and provides a constructor to initialize it to a value specified in centigrade. Thus:

```
new ACentigradeTemperature(100)
```

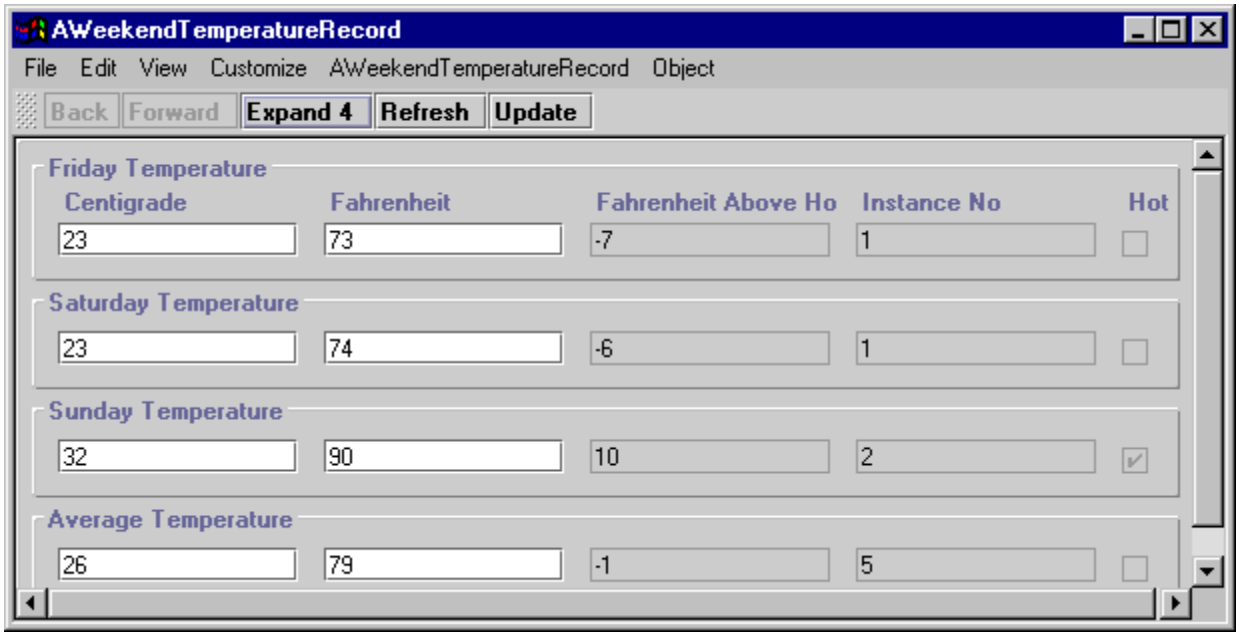
returns a new temperature representing 100 degree centigrade.

7. Provide another implementation of the interface that represents the temperature as a Fahrenheit value and provides a constructor to initialize it to a value specified in centigrade. Thus:

```
new AFahrenheitTemperature (212)
```

returns a new temperature representing 212 degree Fahrenheit.

8. Implement an object, displayed below, defining the temperatures recorded during some weekend. It has four properties: the first three specify the temperatures recorded on the three days of the weekend, Friday, Saturday, and Sunday; and the fourth displays the average of the temperatures of these three days.



You will not (by default) get a tabular display of the kind shown above; instead you will get each item displayed on a separate line, which is acceptable. The display has been customized so that it takes less space.

It is up to you which representation you use for a particular temperature, but you should use instances of both `ACentigradeTemperature` and `AFahrenheitTemperature` to represent the temperatures. That is, one to three of the four temperatures should be represented using instances of `ACentigradeTemperature` and the remaining should be represented using instances of `AFahrenheitTemperature`. (Can you tell from the figure that Friday and Saturday temperatures use different representation?)

It is also up to you whether you compute the average of the three temperatures inside in the temperature classes (`ACentigradeTemperature` & `AFahrenheitTemperature`) or in the user of these classes (`AWeekendTemperatureRecord`).

9. Draw diagrams giving the physical and logical structure of the object above.

10. What are the consequences, on the users of `AWeekendTemperature`, of using a particular representation of a temperature?