Chapter 7
Inheritance

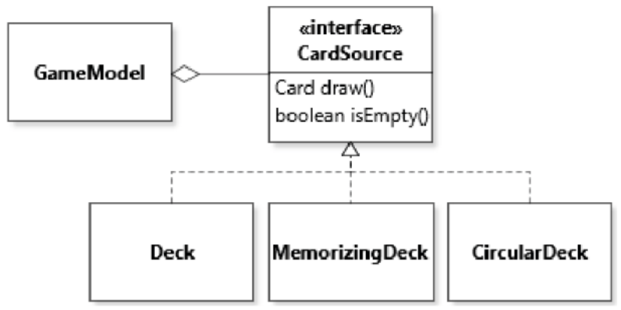
This chapter covers:

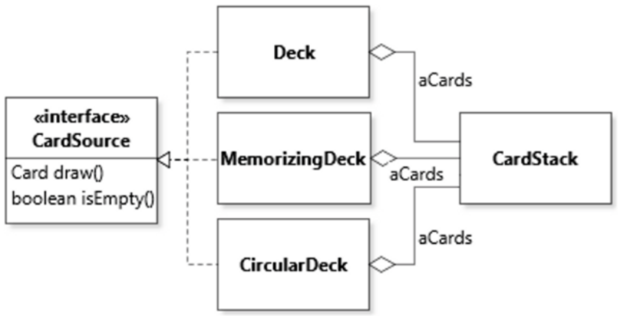
Concepts and Principles: Code reuse, extensibility, Liskov Substitution Principle;
Programming Mechanisms: Inheritance, subtyping, downcasting, object initialization, super calls, overriding, overloading, abstract classes, abstract methods, final classes, final methods;
Design Techniques: Inheritance-base reuse, class hierarchy design;
Patterns and Antipatterns: TEMPLATE METHOD.

7.1 The Case for Inheritance

Polymorphism helps make a design extensible by decoupling client code from the concrete implementation of a required functionality.

This design is extensible because in principle the GameModel can work with any card source. As discussed in Chapter 3, in Java polymorphism relies intrinsically on the language's subtyping mechanism. The key to supporting various options for a CardSource is the fact that the different concrete implementations of the service are subtypes of the CardSouce interface type.



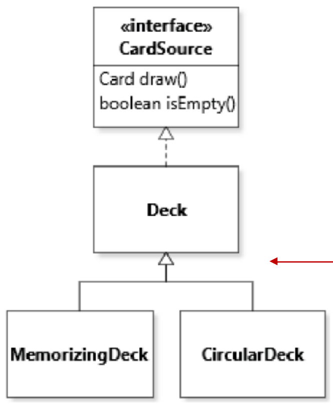


DUPLICATED CODE†
Services defined by the CardSource interface are similar, and likely to be implemented in similar ways.

- In all cases the implementation of method isEmpty() is simply a delegation to aCards.isEmpty()
- In all cases the implementation of method draw() pops a card from aCards: the only difference between the three options is small variants for the remainder of the implementation of draw (e.g., to re-insert the card in the deck in the CardStack of CircularDeck).

→ code reuse: Inheritance

The key idea of inheritance is to define a new class (the subclass) in terms of how it adds to (or extends) an existing base class (also called the superclass). Inheritance avoids repeating declarations of class members because the declarations of the base class will automatically be taken into account when creating instances of the subclass.



In class diagrams, inheritance is denoted by a solid line with a white triangle pointing from the subclass to the superclass.

7.2 Inheritance and Typing

```
public class MemorizingDeck extends Deck
{
  private CardStack aDrawnCards;
}
```

To understand the effect of inheritance in code, it is important to remember that a class is essentially a template for creating objects. Defining a subclass `MemorizingDeck` as an extension of a superclass `Deck` means that when objects of the subclass are instantiated, the objects will be created by using the declaration of the subclass and of the declaration of the superclass. The result will be a single object. The run-time type of this object will be the type specified in the new statement. However, just as for interface implementation, inheritance introduces a suptyping relation. For this reason, an object can always be assigned to a variable of its superclass (in addition to its implementing interfaces).

```
Deck deck = new MemorizingDeck();
CardSource source = deck;
```

a new object of run-time type MemorizingDeck is created and assigned to a variable named deck of compile-time type Deck.

The code declares a variable of type CardSource and assigns the value deck to it. The compile-time type of deck is Deck, which is a subtype of CardSource. For this reason, the compiler allows the assignment. At run time, it will turn out that the concrete type of deck is MemorizingDeck. However, because MemorizingDeck is a subtype of both Deck and CardSource, there is no problem.

Difference between run-time type and compile-time(or static) type

In Java, once an *object* is created, its run-time type remains unchanged. All the variable reassignments accomplish in the code above is to change the type of the *variable that holds a reference to the object*. The run-time type of an object is the most specific type of an object when it is instantiated. It is the type mentioned in the `new` statement, and the one that is represented by the object returned by method `getClass()`.

```
public static boolean isMemorizing(Deck pDeck)
{ return pDeck instanceof MemorizingDeck; }

public static void main(String[] args)
{
  Deck deck = new MemorizingDeck();
  MemorizingDeck memorizingDeck = (MemorizingDeck) deck;
  boolean isMemorizing1 = isMemorizing(deck); // true
  boolean isMemorizing2 = isMemorizing(memorizingDeck); // true
}
```

At the first line of the `main` method an object is created that is of run-time type `MemorizingDeck` and assigned to a variable of type `Deck`. As stated above, the run-time type of this object remains `MemorizingDeck` throughout the execution of the code. However, at the following line the static type of the variable that stores the original object is `MemorizingDeck`, and within the body of method `isMemorizingDeck` it is `Deck` (a formal parameter is a kind of variable, so the type of a parameter acts like a type of variable). Because the run-time type of the object never changes, the value stored in both `isMemorizing1` and `isMemorizing2` is **`true`**.

Downcasting

When using inheritance, subclasses typically pro- vide services in addition to what is available in the base class. For example, a class `MemorizingDeck` would probably include the definition of a service to obtain the list of cards drawn from the deck:

```
public class MemorizingDeck extends Deck
{
  public Iterator<Card> getDrawnCards() { ... }
}
```

Because of the typing rules discussed in Chapter 3, it is only possible to call methods that are applicable for a given static type. So if we assign a reference to an object of run-time type `MemorizingDeck` to a variable of type `Deck`, then we will get a *compilation error* if we try to access a method of the subclass:

```
Deck deck = new MemorizingDeck();
deck.getDrawnCards();
```

Downcasting involves some risks because a downcast encodes an assumption that the run-time type of the object referred to in the variable is of the same type as (or a subtype of) the type of the variable. In a way the code above would be a little bit like writing:

```
assert deck instanceof MemorizingDeck;
```

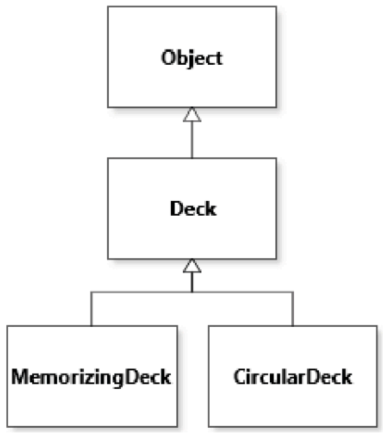
If the assumption is wrong, then the execution of the code cannot proceed, and the downcast will raise a `ClassCastException`. For this reason, downcasting code will often be protected by control structures to assert the run-time type of an object, such as:

```
if( deck instanceof MemorizingDeck )
{
  return ((MemorizingDeck)deck).getDrawnCards();
}
```

Singly-Rooted Class Hierarchy

Java supports single inheritance, which means that a given class can only declare to inherit from (i.e., extend) a single class.

Classes in Java are organized into a single-rooted class hierarchy. If a class does not declare to extend any class, by default it extends the library class `Object`. Class Object thus constitutes the root of any class hierarchy in Java code. The complete class hierarchy for variants of Deck thus includes class Object. Because the subtyping relation is *transitive(A->B, B->C, then A->C)* objects of class `MemorizingDeck` can thus be stored in variables of type Object, etc.



7.3 Inheriting Fields

When creating a new object, this object will have a field for each field declaration in the class named in the new statement, and each of its superclasses, transitively.

```
pulic class Deck implements CardSource
{
  private final CardStack aCards = new CardStack();
  ...
}

public class MemorizingDeck extends Deck
{
  private final CardStack aDrawnCards = new CardStack();
  ...
}
```

Objects created with the statement new `MemorizingDeck();` will have two fields: aCards and aDrawnCards. It does not matter that the fields are private. Accessibility is a static concept, meaning that it is only relevant for the source code. The fact that the code in class `MemorizingDeck` cannot access (or see) the field declared in its superclass does not change anything about the fact that this field is inherited [but not visible]. For the fields to be accessible to subclasses, it is possible to set their access modifier to `protected` instead of `private`, or to access their value through a getter. Type members declared to be protected are only accessible within methods of the same class, classes in the same package, and subclasses in any package.

Field Initialization

The general principle in Java is that the fields of an object are initialized "top down", from the field declarations of the most general superclass down to the most specific class (the one named in the new statement). In our example, `aCards` would be initialized, then `aDrawnCards`. This order is achieved simply by the fact that *the first instruction of any constructor is to call a constructor of its superclass (If the superclass declares a constructor with no parameter, this call does not need to be explicit)*, and so on. For this reason, the order of constructor calls is "bottom up".

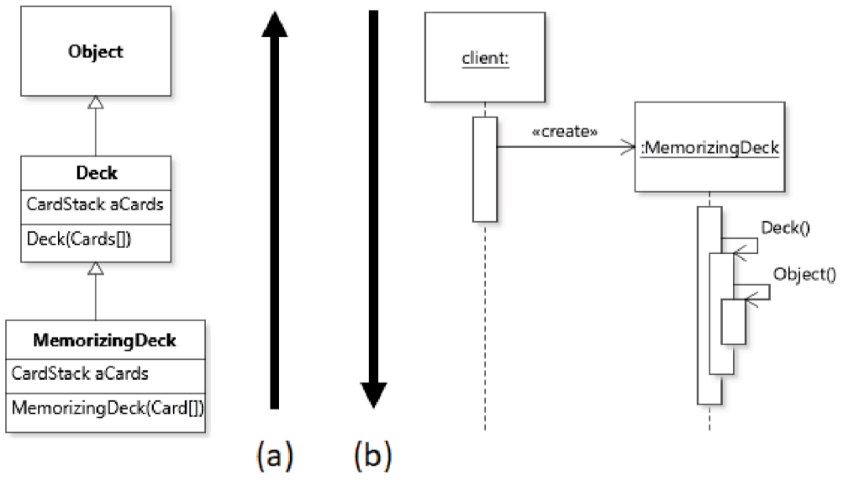


Fig. 7.5 Order of constructor call (a) and object construction (b). The calls to the constructors of a superclass are self-calls.

Example

```
public class MemorizingDeck extends Deck
{
  private final CardStack aDrawnCards = new CardStack();

  public MemorizingDeck(Card[] pCards)
  {
    /* Automatically calls super() */
    ...
  }
}
```

It is also possible to invoke the constructor of the superclass explicitly, using the **super**(...) call. However, if used, this call must be the first statement of a constructor.

```
public class Deck
{
  private final CardStack aCards = new CardStack();

  public Deck(){} // Relies on the field initialization

  public Deck(Card[] pCards)
  {
    for( Card card : pCards)
    {  aCards.push(card); }
  }
}

public class MemorizingDeck extends Deck
{
  private final CardStack aDrawnCards = new CardStack();

  public MemorizingDeck(Card[] pCards)
  { super(pCards);  }
}
```

Difference between using super (…) and new statement

```
public MemorizingDeck(Card[] pCards)
{
  new Deck(pCards);
}
```

It calls the default constructor of Deck, then also creates a new Deck instance, different from the instance under construction, immediately discards the reference to this instance, and then completes the initialization of the object.

7.4 Inheriting Methods

By default, methods of a superclass are applicable to instances of a subclass.

```
MemorizingDeck memorizingDeck = new MemorizingDeck();
memorizingDeck.shuffle();
```

Note:
An instance (i.e., non-static) method is just a different way to express a function that takes an object of its declaring class as its first argument.

Case 1: non-static

```
public class Deck implements CardSource
{
  private CardStack aCards = new CardStack();

  public void shuffle()
  {
    // The 'this' keyword is optional in this case. It is used
    // here to contrast with the alternative below.
    this.aCards.clear();
    this.initialize();
  }

  private void initialize()
  { /* Adds all 52 cards to aCard in random order */ }
}
```

- The function is invoked by specifying the target object before the call: `memorizingDeck.shuffle()`.
- In this case we refer to the `memorizingDeck` parameter as the implicit parameter. A reference to this parameter is accessible through the `this` keyword within the method.

Case 2: static

```
public class Deck implements CardSource
{
  private CardStack aCards = new CardStack();

  public static void shuffle(Deck pThis)
  {
    pThis.aCards.clear();
    pThis.initialize();
  }

  private void initialize()
  { /* Adds all 52 cards to aCard in random order */ }
}
```

The function is invoked by specifying the target object as an explicit parameter, so after the call: `shuffle(memorizingDeck)`. In this case to clear any ambiguity it is usually necessary to specify the type of the class where the method is located, so `Deck.shuffle(memorizingDeck)`.

What this example illustrates is that methods of a superclass are automatically applicable to instances of a subclass because instances of a subclass can be assigned to a variable of any supertype.

Overriding

When multiple implementations are applicable, the run-time environment selects the *most specific one based on the run-time type of the implicit parameter.*
Because the selection of an overridden method relies on run-time information, the selection procedure is often referred to as dynamic dispatch, or dynamic binding. It is important to remember that type information for variables is ignored for dynamic dispatch.

```
Deck deck = new MemorizingDeck();
Card card = deck.draw();
```

The method `MemorizingDeck.draw()` would be selected, even though the static (compile-time) type of the target object is `Deck`.

```
public class Deck implements CardSource
{
  private CardStack aCards = new CardStack();

  public Card draw()
  { return aCards.pop(); }
}
```

```
public class MemorizingDeck extends Deck
{
  private CardStack aDrawnCards = new CardStack();

  public Card draw()
  {                                  private → compile time error
    Card card = aCards.pop();
    aDrawCards.push(card);
    return card;
  }
}
```

Solution
(a) declare aCards to be protected
(b) super

```
public class MemorizingDeck extends Deck
{
  private CardStack aDrawnCards = new CardStack();

  public Card draw()
  {
    Card card = draw(); // Problematic
    aDrawCards.push(card);
    return card;
  }
}
```

Because the call to draw() within MemorizingDeck.draw() will be dispatched on the same object, the same method implementation will be selected, endlessly. The result will be a *stack overflow* error, because the method will recursively call itself without a termination condition.

```java
public class MemorizingDeck extends Deck
{
  public Card draw()
  {
    Card card = super.draw();
    aDrawCards.push(card);
    return card;
  }
}
```

Annotating Overridden Methods

For a method to effectively override another one, it needs to have the same **signature** as the one it overrides.
If a method annotated with `@Override` does not actually override anything, a compilation error is raised.

7.5 Overloading methods

The main thing to know about overloading is that the selection of a specific overloaded method or constructor is based on the number and *static* types of the explicit arguments. The selection procedure is to find all applicable methods and to select the most specific one.

```java
public class MemorizingDeck extends Deck
{
  private CardStack aDrawnCards = new CardStack();

  public MemorizingDeck()
  { /* Version 1: Does nothing besides the initialization */ }

  public MemorizingDeck(CardSource pSource)
  { /* Version 2: Copies all cards of pSource into
     * this object */ }

  public MemorizingDeck(MemorizingDeck pSource)
  { /* Version 3: Copies all cards and drawn cards of pSource
     * into this object */ }
}
```

not same(overloading)

```java
MemorizingDeck memorizingDeck = new MemorizingDeck();
Deck deck = memorizingDeck;

Deck newDeck1 = new MemorizingDeck(memorizingDeck);
Deck newDeck2 = new MemorizingDeck(deck);
```

version 3

version 2 *Because Deck is a subtype of CardSource but NOT a subtype of MemorizingDeck*

In contrast, with inheritance, the cards in the deck are not stored in a separate deck, but rather referred to from a field *inherited* from the superclass. In terms of methods, `shuffle()`, `isEmpty()`, and `draw()` are also inherited from the superclass, so they do not need to be redefined to delegate the call, as in composition. In our example we only need to override `shuffle()` and `draw()` to account for the memorization. Method `isEmpty()` can be directly inherited and still do what we want. In the code of the overridden methods, the delegation to another object is replaced by a super call, which executes on the *same* object.

```
public class MemorizingDeck extends Deck
{
  private final CardStack aDrawCards = new CardStack();

  public void shuffle()
  {
    super.shuffle();
    aDrawnCards.clear();
  }

  public Card draw()
  {
    Card card = super.draw();
    aDrawnCard.push(card);
    return card;
  }
}
```

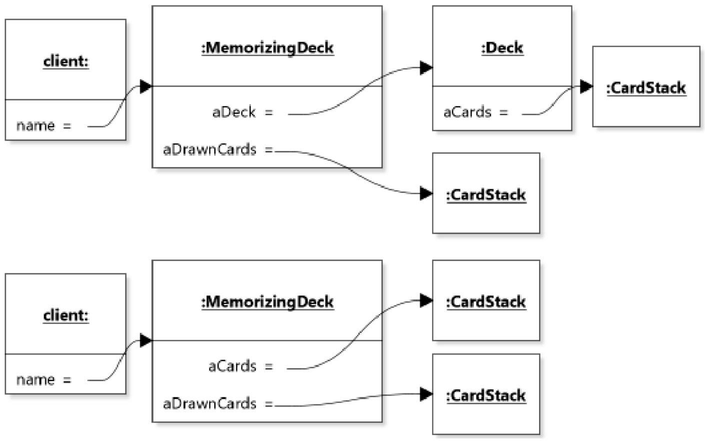


Fig. 7.6 Two implementations for `MemorizingDeck`: Composition-based (top), and inheritance-based (bottom)

The main difference between the two approaches is the number of `Deck` objects involved (see Figure 7.6). The composition-based approach provides a solution that requires combining the work of two objects: a basic `Deck` object and a “wrapper” (or “decorator”) object `MemorizingDeck`. Thus, as discussed in Section 6.4, the identity of the object that provides the full `MemorizingDeck` set of features is different from that of the other object that provides the basic card handling services of the deck. In contrast, the use of a `MemorizingDeck` subclass creates a *single* `MemorizingDeck` object that contains all the required fields.

In most situations, it may be possible to realize a design solution using either inheritance or composition. Which option to choose will ultimately depend on the context. Composition-based reuse generally provides more run-time flexibility. This option should therefore be favored in design contexts that require many possible configurations, or the opportunity to change configurations at run-time. At the same time, composition-based solutions provide fewer options for detailed access to the internal state of a well-encapsulated object. In contrast, inheritance-based reuse solutions tend to be better in design contexts that require a lot of compile-time configuration, because a class hierarchy can easily be designed to provide privileged access to the internal structure of a class to subclasses (as opposed to aggregate and other client classes).

7.7 Abstract Classes

There are often situations where locating common class members into a single superclass leads to a class declaration that it would not make sense to instantiate. As a running example for this section and the next, I continue to develop the concept of command objects as introduced in Section 6.8. Let us assume that for a card game application we decide to apply the COMMAND pattern and use the following definition of the command interface.

```
public interface Move
{
  void perform();
  void undo();
}
```

This interface is pretty self-explanatory. A “move” represents a possible action in the game. Calling `perform()` on any subtype of `Move` performs the move, and calling `undo()` undoes the move. The class diagram of Figure 7.7 shows a hypothetical application of the COMMAND pattern. Following a common naming convention, classes that implement the interface include the name of the interface as a suffix (e.g., `DiscardMove` represents the move that discards a card from the deck).

At a glance the diagram reveals a redundancy: each concrete command class stores an aggregation to an instance of `GameModel`, which the implementation of `perform()` and `undo()` will rely on when executing the respective commands. In terms of source code, this would look very similar: a field of type `GameModel` (called `aModel` in the diagram). As pointed out in Section 7.1, avoiding CODE DUPLICATION† is the very motivation for inheritance, so we should “pull up” the field `aModel` into a common superclass. However, there is a big difference between the `Deck` class

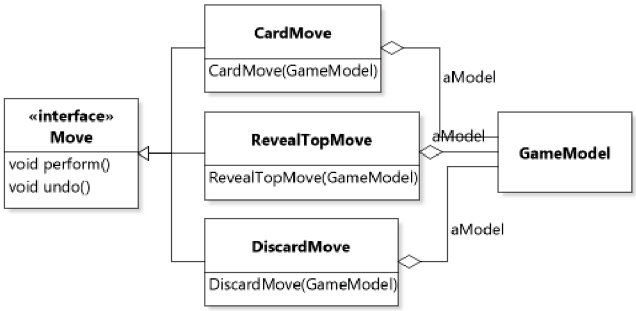


Fig. 7.7 Abstract and concrete commands

example of Section 7.1 and the command example discussed here. With a `Deck` base class and various subclasses that specialize it, *it makes sense to instantiate the base class*. If we want an instance of a `Deck` with no frills, we do **new** `Deck()` and we have what we want. In the case of commands, what would be the base class? One option is to arbitrarily select one concrete command and use it as the base class, as illustrated by the diagram of Figure 7.8.

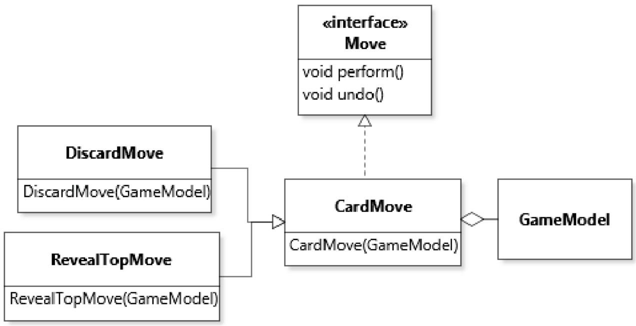


Fig. 7.8 Abuse of inheritance: the members of the base class end up being completely redefined instead of specialized

Although this could work, it is not good design. An important principle of inheritance is that a subclass should be a "natural" subtype of the base class that extends the behavior of the base class. In our case, a `DiscardMove` is not really a specialized version of a `CardMove`, they are just two different moves. First, `CardMove` may define non-interface methods that make no sense for users of its subclass (e.g., `getDestination()` to get the destination when moving a card). Second, this idea is risky, because `DiscardMove` and `RevealTopMove` automatically inherit the `perform()` and `undo()` methods of class `CardMove`, which need to be completely overridden to implement the actual move we want. If we forget to implement one (`undo()` for example), then calling `perform()` will do one thing, and calling `undo()` will undo something else! These types of bugs can be very hard to catch. I will return to the issue of design ideas that abuse inheritance in Section 7.10. To use inheritance properly here, we need to create an entirely new base class, and have all "actual" commands inherit it, as shows in Figure 7.9.

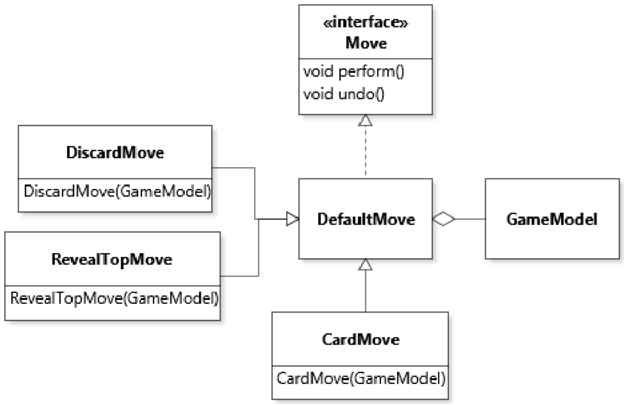


Fig. 7.9 Inheritance with additional base class

Now we avoid the hack of having subclasses that morph their superclass into something entirely different. However, at the same time we have a problematic situation: what is a `DefaultMove`, really? What would the implementation of `perform()` and `undo()` actually do? Here, even using some sort of default behavior seems questionable, because that would bring us back to the idea of using a base class that is not conceptually a base for anything. A key observation to move forward is that our new base class represents a purely abstract concept that needs to be refined to gain concreteness. This design situation is directly supported by the *abstract class* feature of a programming language. Technically, an abstract class represents a correct but *incomplete* set of class member declarations.

In Java a class can be declared abstract by including the keyword **`abstract`** in its declaration. It is also a common practice to prefix the identifier for an abstract class with the word `Abstract`. Hence, in our design the `DefaultMove` should be called `AbstractMove`, and its definition would look like this:

```
public abstract AbstractMove implements Move
{
  private final GameModel aModel;

  protected AbstractMove(GameModel pModel)
  { aModel = pModel; }

  ...
}
```

Declaring a class to be abstract has three main consequences:

- The class cannot be instantiated, which is checked by the compiler. This is a good thing because abstract classes should represent abstract concepts that it makes no sense to instantiate. Another typical example, besides abstract commands, would be something like an `AbstractFigure` in a drawing editor. Unlike concrete figures (rectangles, ellipses), an abstract figure has no geometric representation, so in most designs something like that would be likely to end up as an abstract class.
- The class no longer needs to supply an implementation for all the methods in the interface(s) it declares to implement. This relaxing of the interface contract is type-safe because the class cannot be instantiated. However, any concrete (i.e., non-abstract) class will need to have implementations for all required methods. What this means in our case is that, even though `AbstractMove` declares to implement `Move`, we do not have to supply an implementation for `perform()` and `undo()` in the abstract class. However, this assumes that non-abstract subclasses of `AbstractMove` will supply this missing implementation.
- The class can declare new abstract methods using the same **`abstract`** keyword, this time placed in front of a method signature. In practice this means adding methods to the interface of the abstract class, and thereby forcing the subclasses to implement these methods. The usage scenario for this is a bit specialized, and I will cover it in detail in Section 7.9. However, for now, we can just say that in most designs abstract methods are typically called from within the class hierarchy: by methods of the base class, by methods of the subclasses, or both.

Note that because abstract classes cannot be instantiated, their constructor can only be called from within the constructors of subclasses. For this reason it makes sense to declare the constructors of abstract classes **`protected`**. In our running example, the constructor of `AbstractMove` would be called by the constructor of subclasses to pass the required reference to the `GameModel` “up” into the base class:

```
public class CardMove extends AbstractMove
{
  public CardMove(GameModel pModel)
  {
    super(pModel);
  }
}
```

7.8 Revisiting the DECORATOR Design Pattern

In Section 6.4, we saw how we can use the DECORATOR pattern to add feature elements, or “decorations”, to an object at run-time. The key idea of the DECORATOR is to define these decorations using wrapper classes and composition as opposed to subclasses. Figure 7.10 reproduces Figure 6.8, which shows a class diagram of the sample application of DECORATOR to the `CardSource` design context.

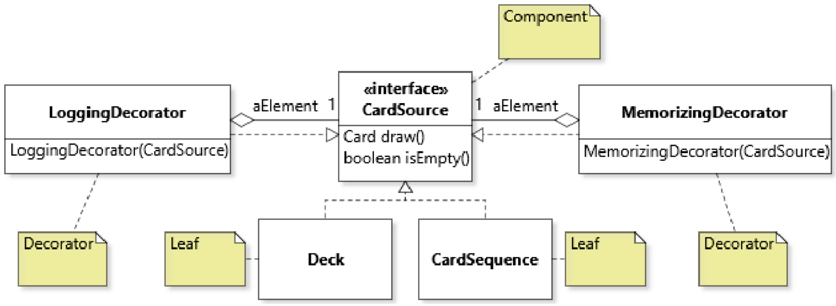


Fig. 7.10 Class diagram of a sample application of DECORATOR

When a design involves multiple decorator types, as in this example, each decorator class will need to aggregate an object to be decorated. This introduces the kind of redundancy that inheritance was designed to avoid. Thus, we can use inheritance to "pull up" the `aElement` field into an abstract decorator base class, and define concrete decorator subclasses that then only need to deal with the specific decoration. This solution, shown in Figure 7.11, is a good illustration of a design that combines ideas of composition (as seen in Chapter 6) and inheritance. Specifically, a decorator object is of a subtype that *inherits* the `aElement` field, which is then used to *aggregate* the instance of `CardSource` that is being decorated.

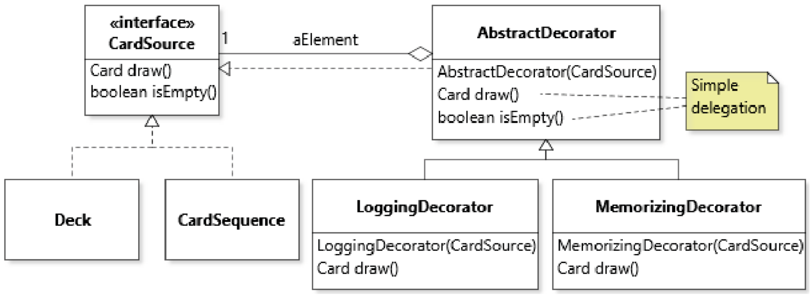


Fig. 7.11 Class diagram of a sample application of DECORATOR that uses inheritance

With this design, the `AbstractDecorator` includes default delegation to the decorated element.

```
public class AbstractDecorator implements CardSource
{
  private final CardSource aElement;

  protected AbstractDecorator(CardSource pElement)
  { aElement = pElement; }

  public Card draw() { return aElement.draw(); }

  public boolean isEmpty() { return aElement.isEmpty(); }
}
```

It is worth noting that the `aElement` field is private. This means concrete decorator classes will not have access to it. This level of encapsulation is workable because normally in the DECORATOR, decorated elements are only accessed through the methods of the component interface. In this case, subclasses can simply use the implementation of the interface methods they inherit from `AbstractDecorator` to interact with the decorated object. As an example, the following is a basic implementation of a `LoggingDecorator` that outputs a description of the cards drawn to the console.

```
public class LoggingDecorator extends AbstractDecorator
{
  public LoggingDecorator(CardSource pElement)
  { super(pElement); }

  public Card draw()
  {
    Card card = super.draw();
    System.out.println(card);
    return card;
  }
}
```

Class `LoggingDecorator` does not supply an implementation of `isEmpty()` because the one it inherits, which delegates the call to `aElement`, does what we want. As for `draw`, the method is redefined to do a basic draw operation using the inherited method, print the card, then return it to complete the require behavior.

7.9 The TEMPLATE METHOD Design Pattern

One potential situation with inheritance is when some common algorithm applies to objects of a certain base type, but a part of the algorithm varies from subclass to subclass. To illustrate this situation, let us go back to the design context of creating and managing moves in the Solitaire application, as discussed above in Section 7.7 and illustrated in Figure 7.9 (with `DefaultMove` renamed to `AbstractMove`). In this context we also assume that `aModel`'s access modifier is **`protected`**.

Let us assume that calling method `perform()` on moves *of any type* should accomplish three actions: *1)* Add the move to an undo stack, possibly located in the `GameModel`;[9] *2)* Perform the actual move; *3)* Log the move by writing out a description of what happened. This algorithm can be described with the following code, which could be in any concrete subclass of `AbstractMove`:

[9] This is not necessarily the best idea from a separation of concerns standpoint, but I use this example for simplicity. The Code Exploration section points to what might be a better alternative for accumulating commands.

```
public void perform()
{
  aModel.pushMove(this);
  /* Actually perform the move */
  log();
}
```

In this code, the first statement of method `perform()` pushes the current move object onto a command stack located in the game model. The block comment corresponds to the actual implementation of the move, which would vary from move to move. The final statement implements some logging of the move, for example by printing the name of the command class to the console. Let us assume the same approach is used for `undo()`, with moves being popped instead of pushed. Because parts of the code are common, it will benefit from being "pulled up" to the `AbstractMove` superclass for two main reasons:

- So that it can be reused by all concrete `Move` subclasses, thereby avoiding DUPLICATED CODE†;
- So that the design is robust to errors caused by inconsistently re-implementing common behavior. Specifically, we want to prevent the possibility that a developer could later declare a new concrete subclass of `Move` and supply it with an implementation of method `perform()` that does not do steps 1 and 3, for example.

Because the implementation of `perform()` needs information from subclasses to actually perform the move, it cannot be completely implemented in the superclass.[10] The solution to this problem is to put all the common code in the superclass, and to define some "hooks" to allow subclasses to provide specialized functionality where needed. This idea is called the TEMPLATE METHOD design pattern. The name relates to the fact that the common method in the superclass is a "template", that gets "instantiated" differently for each subclass. The "steps" are defined as non-private[11] methods in the superclass. The code below further illustrates the solution template for the TEMPLATE METHOD:

[10] Although, technically, it would be possible to have a SWITCH STATEMENT† in `perform()` that checks the concrete type of the object using **`instanceof`** or `getClass()` and executes the appropriate code for all commands, this would introduce a dependency cycle between the base class and its subclasses, and completely destroy the benefits of polymorphism. A bad idea of epic proportions.

[11] The "step methods" can have default, public, or protected visibility depending on the design context. However, they cannot be private because private methods cannot be overridden. This constraint makes sense because private methods are technically not visible outside of their class, and overriding requires method signature matching across classes.

```java
public abstract class AbstractMove implements Move
{
  protected final GameModel aModel;

  protected AbstractMove(GameModel pModel)
  { aModel = pModel; }

  public final void perform()
  {
    aModel.pushMove(this);
    execute();
    log();
  }

  protected abstract void execute();

  private void log()
  { System.out.println(getClass().getName()); }
}
```

In this code example, the implementation of method `perform()` introduces two new concepts related to inheritance: *final* methods (and classes) and *abstract method declarations* in classes.

Final Methods and Classes

In Java, declaring a method to be **`final`** means that the method cannot be overridden by subclasses. The main purpose for declaring a method to be **`final`** is to clarify our intent, as designers, that a method is not meant to be overridden. One important reason for preventing overriding is to ensure that a given constraint is respected. Final methods are exactly what is needed for the TEMPLATE METHOD, because we want to ensure that the template is respected for all subclasses. By declaring the `perform()` method to be **`final`**, subclasses cannot override it with an implementation that would omit the call to `pushMove` or `log()`.

The use of the **`final`** keyword with methods has an effect that is different from the use of the same keyword with fields and local variables (see Section 4.6). The use of **`final`** with fields limits how we can assign values to variables, and does not involve inheritance, dynamic dispatch, or overriding.

The **`final`** keyword can also be used with classes. In this case, the behavior is consistent with the meaning it has for methods: classes declared to be **`final`** cannot be inherited. Inheritance in effect broadens the interface of a class by allowing extensions by other classes. As demonstrated in Figure 7.8 and as will be further discussed in the next section, inheritance is a powerful mechanism that can easily be misused. A good principle to follow with inheritance is “design for inheritance or else prohibit it” [1, Item 19]. In other words, inheritance should be used to support specific extension scenarios (as the one illustrated in this section), or not used at all. Because by default, it is possible to inherit from a class, the mechanism needs to be explicitly disabled to prohibit its use. Generally, stating that a class cannot be inherited tends to make a design more robust because it prevents unanticipated effects caused by inheritance. In our current example, we could decide to make immediate subclasses of `AbstractMove` **`final`** to make it clear to other readers of the code that the class hierarchy should not be extended through inheritance beyond a single level of concrete move subclasses.

Although performance is not a major aspect of this book, it is also worth noting that declaring classes and methods to be final can also have some positive implications for the execution speed of a program, because the absence of dynamic dispatch for final classes means that code can be optimized to run more quickly.

Abstract Methods

In the implementation of the `perform()` template method in `AbstractMove`, the second step is to perform the actual move. Within class `AbstractMove`, this step is meaningless given that an abstract move does not represent any concrete move we could perform. For this reason, we need to leave out the actual execution of the move. However, we cannot leave this step out *entirely*, because as part of our template we do need to specify that executing the move needs to happen, and needs to happen specifically after the move is pushed to the move stack and before the move execution is logged. In our design we thus specify that this computation needs to happen by calling a method. However, because in Java all methods that are called need to be declared, we must add a new method declaration. In this example I called it `execute()`, because we cannot give it the same name as the template method (this would result in a recursive call). Because we do not have any implementation for `execute()`, we can defer the implementation to the subclasses. This is allowed because `AbstractMove` is declared to be **`abstract`**, so there is no issue if the class's interface is not fully implemented. Although it sometimes makes sense to declare abstract methods to be public, here I declare `execute()` to be protected because the only classes that really need to see this method are the subclasses of `AbstractMove` that must supply an implementation for it.

Summary of the Pattern

The declaration of class `AbstractMove`, above, illustrates the key ideas of the solution for TEMPLATE METHOD. The following points are also important to remember about the use of the pattern:

- The method with the common algorithm in the abstract superclass is the *template method*; it calls the concrete and abstract *step methods*;
- If, in a given context, it is important that the algorithm embodied by the template method be fixed, it could be a good idea to declare the template method **`final`**, so it cannot be overridden (and thus changed) in subclasses;
- It is important that the abstract step method has a different signature from the template method for this design to work. Otherwise, the template method would

recursively call itself, quite possibly leading to a stack overflow; following the advice of Section 7.5 about avoiding unnecessary overloading, I would recommend actually using a different name in all cases.

- The most likely access modifier for the abstract step methods is **`protected`**, because in general there will likely not be any reason for client code to call individual steps that are intended to be internal parts of a complete algorithm. Client code would normally be calling the template method;
- The steps that need to be customized by subclasses do not necessarily need to be abstract. In some cases, it will make sense to have a reasonable default behavior that could be implemented in the superclass. In this case it might not be necessary to make the superclass abstract. In our example, there is a default implementation of `log()` that can be overridden by subclasses. In a different context, it might make more sense to declare this method abstract as well.

When first learning to use inheritance, the calling protocol between code in the super- and subclasses can be a bit confusing because, although it is distributed over multiple classes, the method calls are actually dispatched to the *same target object*. The sequence diagram in Figure 7.12 illustrates a call to `perform()` on a `DiscardMove` instance. As can be seen, although it is implemented in subclasses, the call to the abstract step method is a self-call.

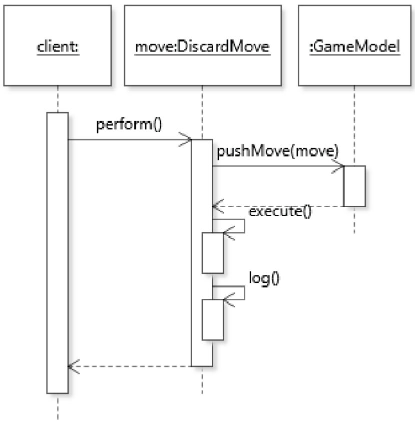


Fig. 7.12 Call sequence in the TEMPLATE METHOD

7.10 Proper Use of Inheritance

Inheritance is both a code reuse and an extensibility mechanism. This means that a subclass inherits the declarations of its superclass, but also becomes a subtype of its superclass (and its superclass's superclass, and so on). To avoid major design flaws, inheritance should only be used for *extending* the behavior of a superclass. As such, it is bad design to use inheritance to restrict the behavior of the superclass, or to use inheritance when the subclass is not a proper subtype of the superclass.

Restricting What Clients of Base Classes Can Do

As an example of a design idea to limit what a superclass can do using our running example of a deck of cards, let us say that in some design context we need to have decks of cards that should never be shuffled. Given that we already have a class (`Deck`) that defines everything we need to instantiate a deck, would it not be easy to simply subclass `Deck` to somehow "deactivate" the shuffling:

```
public class UnshufflableDeck extends Deck
{
  public void shuffle() {/* Do nothing */}
}
```

or,

```
public class UnshufflableDeck extends Deck
{
  public void shuffle()
  { throw new OperationNotSupportedException(); }
}
```

Actually, both versions are a bad design decision because they conflict directly with the use of polymorphism, which supports calling operations on an object independently of the concrete type of the object. Let us consider the following hypothetical calling context:

```
private Optional<Card> shuffleAndDraw(Deck pDeck)
{
  pDeck.shuffle();
  if( !pDeck.isEmpty() )
  { return Optional.of(pDeck.draw()); }
  else
  { return Optional.empty(); }
}
```

This code will compile and, given the interface documentation of `Deck`, should do exactly what we want. However if, when the code executes, the run-time type of the instance passed into `shuffleAndDraw` happens to be an `UnshufflableDeck`, the code will either not work as expected (first variant, the deck silently does not get shuffled), or raise an exception (second variant). There is clearly something amiss here.

The intuition that inheritance should only be used for extension is captured by the *Liskov substitution principle* (LSP). The LSP basically states that subclasses should not restrict what clients of the superclass can do with an instance. Specifically, this means that methods of the subclass:

- Cannot have stricter preconditions;
- Cannot have less strict postconditions;

- Cannot take more specific types as parameters;
- Cannot make the method less accessible (e.g., from **`public`** to **`protected`**);
- Cannot throw more checked exceptions;
- Cannot have a less specific return type.

This list seems like a lot of things to remember when designing object-oriented software, but the whole point of the principle is that once we have assimilated its logic, we no longer need to remember specific elements in the list. In addition, all situations except for the first two points are prevented by the compiler. Nevertheless, at first some of these points can seem a bit counter-intuitive, so let us consider concrete scenarios.

Remembering our interface `CardSource`, which has method `isEmpty()` and a method `draw()` with the precondition `!isEmpty()`, we can design a new subclass of `Deck` that draws the highest of the two top cards on the deck, and replaces the lowest one back on top of the deck.

```
public class Deck implements CardSource
{
  protected final CardStack aCards = new CardStack();

  public Card draw() { return aCards.pop(); }
  public boolean isEmpty() { return aCards.isEmpty(); }
}

public class DrawBestDeck extends Deck
{
  public Card draw()
  {
    Card card1 = aCards.pop();
    Card card2 = aCards.pop();
    Card high = // identify highest card between card1 and card2
    Card low = // identify lowest card  between card1 and card2
    aCards.push(low);
    return high;
  }
}
```

This code is simple and easy to understand, which can be a symptom of good design. There is a catch, however: for this solution to work, the deck needs to have at least two cards in it. How can we deal with this? One solution is to rework the code of the `draw` override to handle both cases:

```
public Card draw()
{
  Card card1 = aCards.pop();
  if( isEmpty() ) { return card1; }
  Card card2 = aCards.pop();
  ...
}
```

However, this code is not as elegant as the first version, it is more costly to test, etc. Given that we know about design by contract (see Section 2.8), why not simply declare the precondition that, to call `draw()` on an instance of `DrawBestDeck`, there needs to be at least two cards in the deck?

```
public class DrawBestDeck extends Deck
{
  public int size() { return aCards.size(); }

  public Card draw()
  {
    assert size() >=2;
    ...
  }
}
```

This makes the precondition stricter (i.e., less tolerant), and thus violates the LSP. This is because the mere presence of the subclass makes code like this unsafe for cases where a deck has only one card.

```
if( deck.size() >=1 ) { return deck.draw(); }
```

Actually, in our specific example, the idea is bad for an additional, if less fundamental, reason. Because the inteface `CardSource` does not have a `size()` method, it is not possible to check the precondition polymorphically. To check that a call to `draw()` respects all preconditions, it would thus be necessary to do:

```
Deck deck = ...
Optional<Card> result = Optional.empty();
if( deck instanceof DrawBestDeck &&
    ((DrawBestDeck)deck).size() >= 2 ||
      !deck.isEmpty() )
{ result = Optional.of(deck.draw()); }
```

Ultimately, we solved nothing by trying to make the overridden version of `draw()` simpler, because the size check simply migrated to the client, acquiring some additional complexity in the process.

Similarly, the LSP is the reason why Java does not allow overriding methods to take more specific types as parameters or have less specific types as return types. Let us say we add a method `init(Deck)` to the interface of `Deck` that re-initializes the target instance to contain exactly the cards in the argument. If we have a `MemorizingDeck` in our class hierarchy, we might be tempted to override `init` in `MemorizingDeck` to initialize both the cards and the drawn cards, as illustrated in Figure 7.13.

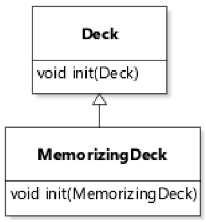


Fig. 7.13 Invalid attempt at method overriding that does not respect the LSP

Although this code will compile, it will actually create an *overloaded* version of `init` as opposed to an *overridden* version, as perhaps expected. This is extremely confusing. The systematic use of the `@Override` annotation (see Section 7.4) would help flag this as a problem, but otherwise, the code would lead to very mysterious executions, given that the result of both calls to `init` would be different in the code below:

```
Deck deck = new Deck();
MemorizingDeck memorizingDeck = new MemorizingDeck();
Deck mDeck = memorizingDeck;
deck.init(memorizingDeck); // Calls MemorizingDeck.init
deck.init(mDeck); // Calls Deck.init
```

The reason for this seemingly strange behavior is again to ensure the design respects the LSP. If clients of the base class `Deck` can call `init` and pass in instances of *any* subtype of `Deck`, then it would be limiting what they can do to require the clients to only pass certain subtypes to `init` (like `MemorizingDeck`).

The case of return types simply inverses the logic: if a method returns an object of a certain type, it should be possible to assign it to a variable of that type. If subclasses can redefine methods to return more general types, then it would no longer be possible to complete the assignment. For example, let us say that in a (bad) design, a developer adds a version of `Deck` that contains a joker card, and that objects that represent jokers are not subtypes of `Card`, but merely subtypes of the `Object` root type. Then calls like this:

```
Card card = deck.draw();
```

would become problematic if some versions of method `draw()` can return objects that are not subtypes of `Card`.

The classic example of a violation of the LSP is the so-called Circle–Ellipse problem, wherein a class to represent a circle is defined by inheriting from an `Ellipse` class and preventing clients from creating any ellipse instance that does not have equal proportions. This violates the LSP because clients that use an `Ellipse` base class can set the height to be different from the width, and introducing a `Circle` subclass would eliminate this possibility:

```
Ellipse ellipse = getEllipse();
// Not possible if ellipse is an instance of Circle
ellipse.setWidthAndHeight(100, 200);
```

How to avoid the Circle–Ellipse problem in practice will, as usual, depend on the context. In some cases, it may not be necessary to have a type `Circle` in the first place. For example, in a drawing editor, user interface features could be responsible for assisting users in creating ellipses that happen to be circles, while still storing these as instances of `Ellipse` internally. In cases where a type `Circle` can be useful, it might make sense to have different `Circle` and `Ellipse` classes that are siblings in the type hierarchy, etc.

Subclasses That Are Not Proper Subtypes

Inheritance accomplishes two things (see Section 7.2):

- It reuses the class member declarations of the base class as part of the definition of the subclass;
- In introduces a subtype–supertype relation between the subclass and the superclass.

To use inheritance properly, it has to make sense for the subclass to need *both* of these features. A common abuse of inheritance is to employ it only for reuse, and overlook the fact that subtyping does not make sense:

> Inheritance is appropriate only in circumstances where the subclass really is a *subtype* of the superclass. In other words, a class *B* should extend a class *A* only if an "is-a" relationship exists between the two classes. [1, p. 92]

Some well-known acknowledged violations of this principle include the library type `Stack` (which inappropriately inherits from `Vector`), and `Properties` (which inappropriately inherits from `Hashtable`). When subtyping is not appropriate, composition should be used.

Insights

This chapter introduced inheritance as a mechanism to support code reuse and extensibility.

- Use inheritance to factor out implementation that is common among subtypes of a given root type and avoid DUPLICATED CODE†;
- UML class diagrams can describe inheritance-related design decisions effectively;
- To the extent possible, use the services provided by a subclass through polymorphism, to avoid the error-prone practice of downcasting;
- Even in the presence of inheritance, consider keeping your field declarations private to the extent possible, as this ensures tighter encapsulation;
- Subclasses should be designed to complement, or specialize, the functionality provided by a base class, as opposed to redefining completely different behavior;
- Use the `@Override` annotation to avoid hard-to-find errors when defining overriding relations between methods;
- Because it can easily lead to code that is difficult to understand, keep overloading to a minimum. Overloading is best avoided altogether when the parameter types of the different versions of a method are in a subtyping relation with each other;
- Inheritance- and composition-based approaches are often viable alternative when looking for a design solution. When exploring inheritance-based solutions, consider whether composition might not be better;