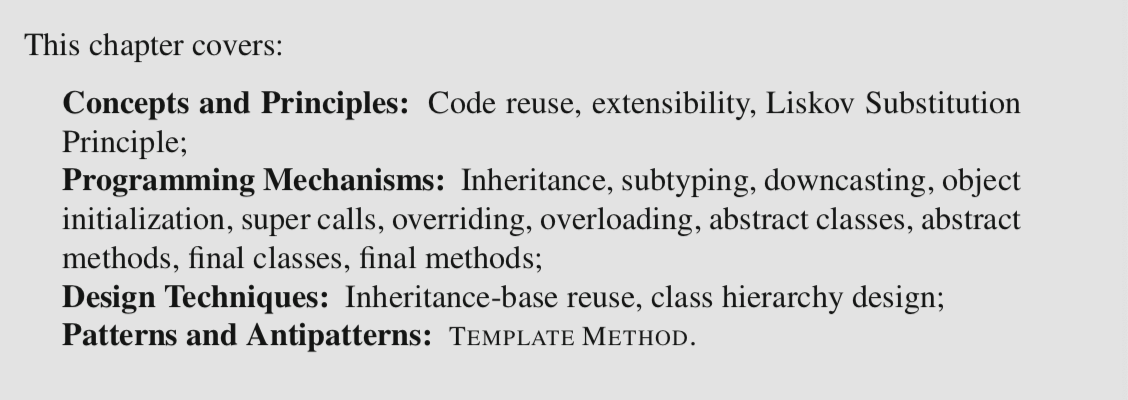
Chapter 7

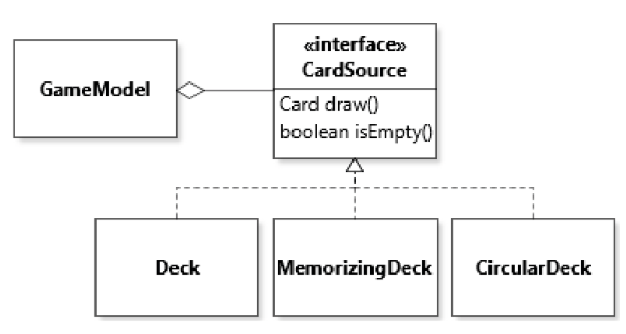
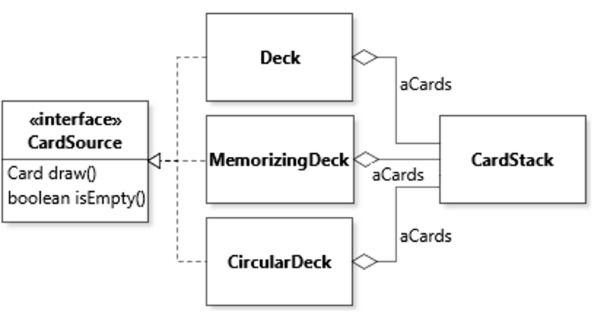
Inheritance



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.



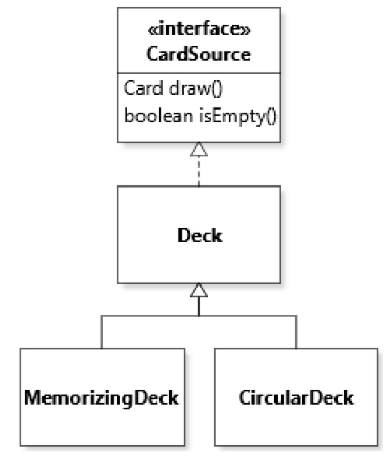
DUPLICATED CODE†

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).

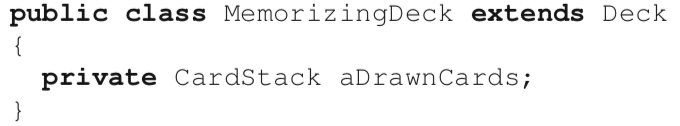
🡪 code reuse: Inheritance

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.



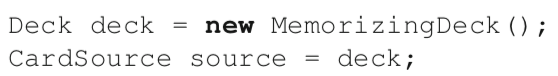
In class diagrams, inheritance is denoted by a solid line with a white triangle pointing from the subclass to the superclass.

7.2 Inheritance and Typing



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).

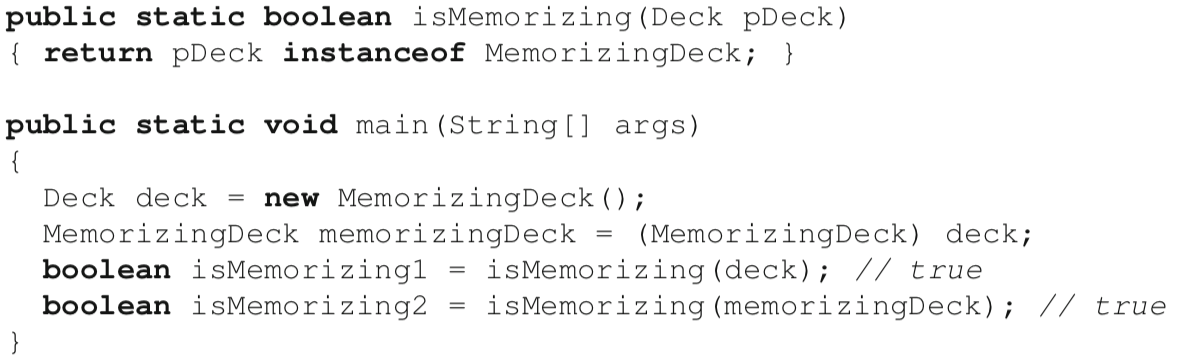
a new object of run-time type MemorizingDeck is created and assigned to a variable named deck of compile-time type Deck.

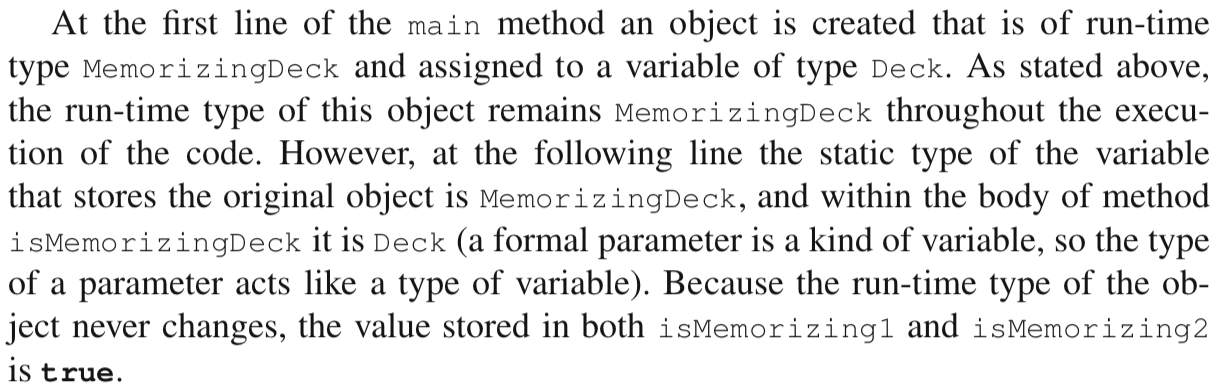


The code declares a variable of type CardSource and assigns the value deck to it. The compile-time type of deck is Deck, which is a subtype of CardSource. For this reason, the compiler allows the assignment. At run time, it will turn out that the concrete type of deck is MemorizingDeck. However, because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

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

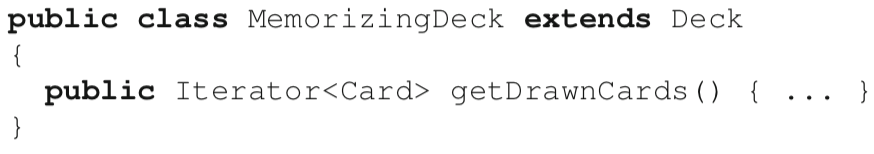
In Java, once an *object* is created, its run-time type remains unchanged. All the variable reassignments accomplish in the code above is to change the type of the *variable that holds a reference to the object*. The run-time type of an object is the most specific type of an object when it is instantiated.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





**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:



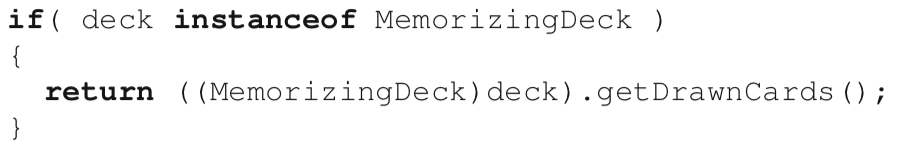
Because of the typing rules discussed in Chapter 3, it is only possible to call methods that are applicable for a given static type. So if we assign a reference to an object of run-time type MemorizingDeck to a variable of type Deck, then we will get a *compilation error* if we try to access a method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



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:



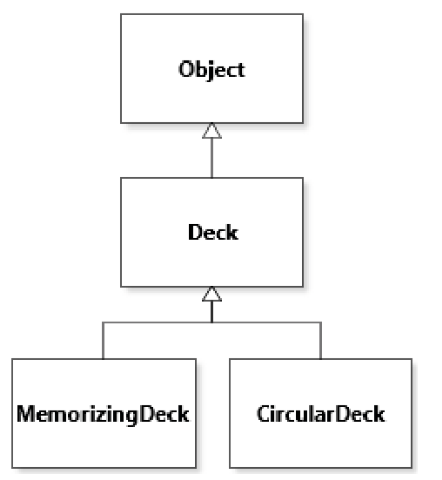
If the assumption is wrong, then the execution of the code cannot proceed, and the downcast will raise a ClassCastException. For this reason, downcasting code will often be protected by control structures to assert the run-time type of an object, such as:



**Singly-Rooted Class Hierarchy**

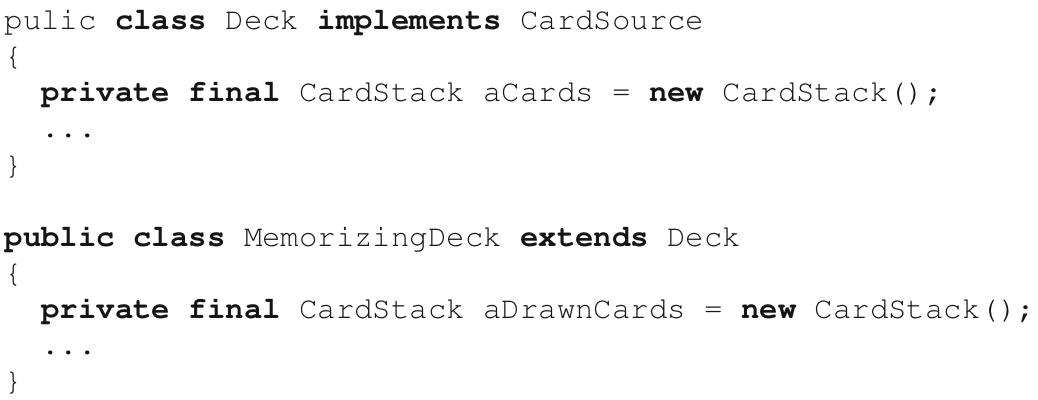
Java supports single inheritance, which means that a given class can only declare to inherit from (i.e., extend) a single class.

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.



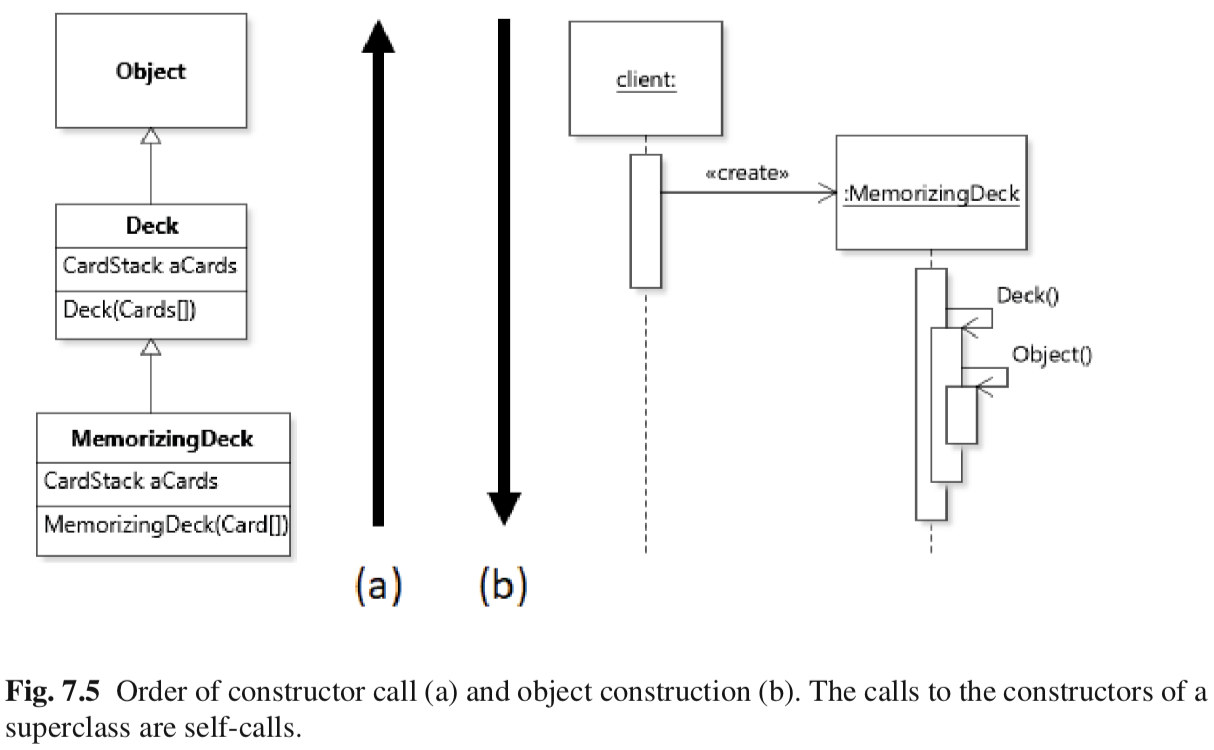
7.3 Inheriting Fields

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.

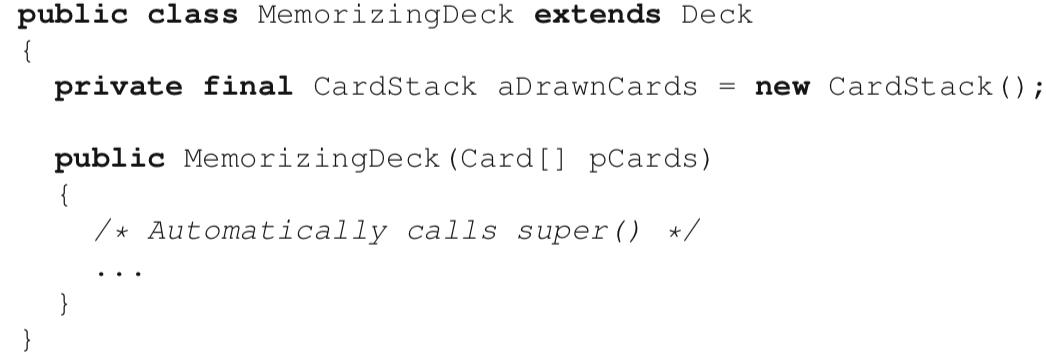


Objects created with the statement new MemorizingDeck(); will have two fields: aCards and aDrawnCards. It does not matter that the fields are private. Accessibility is a static concept, meaning that it is only relevant for the source code. The fact that the code in class MemorizingDeck cannot access (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.

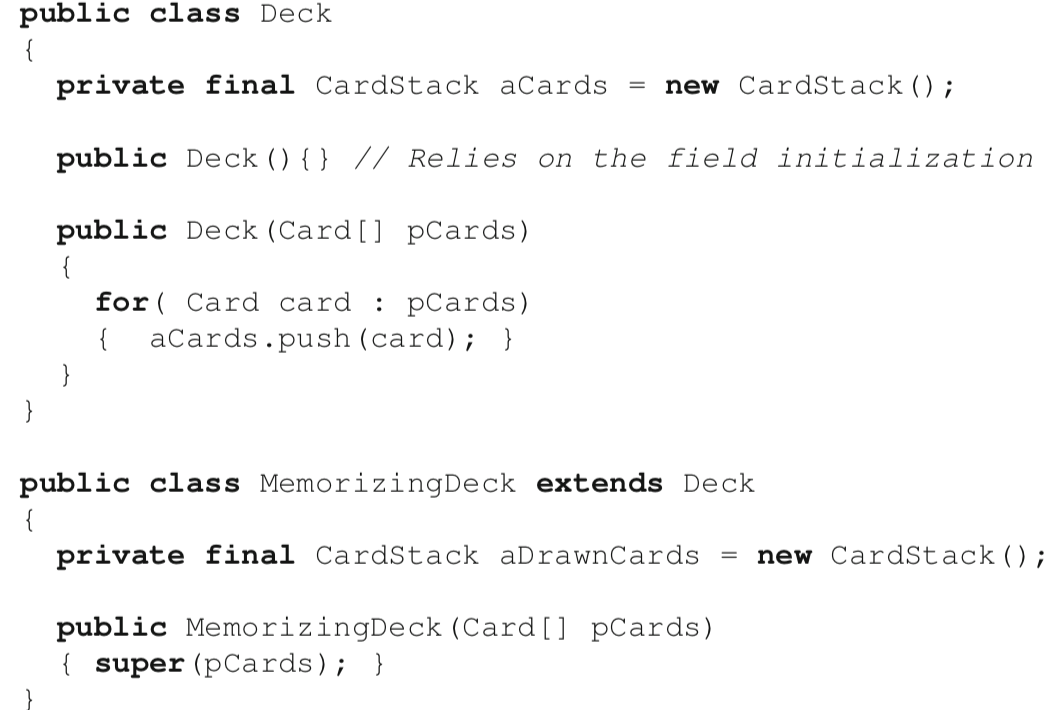
**Field Initialization**

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”.

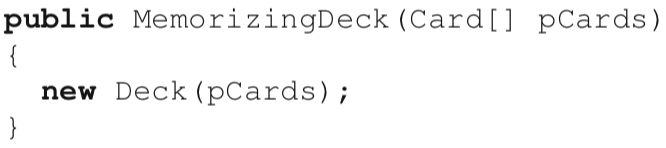
Example



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.



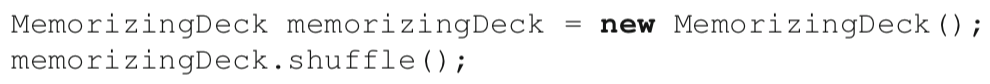
Difference between using super(…) and new statement



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

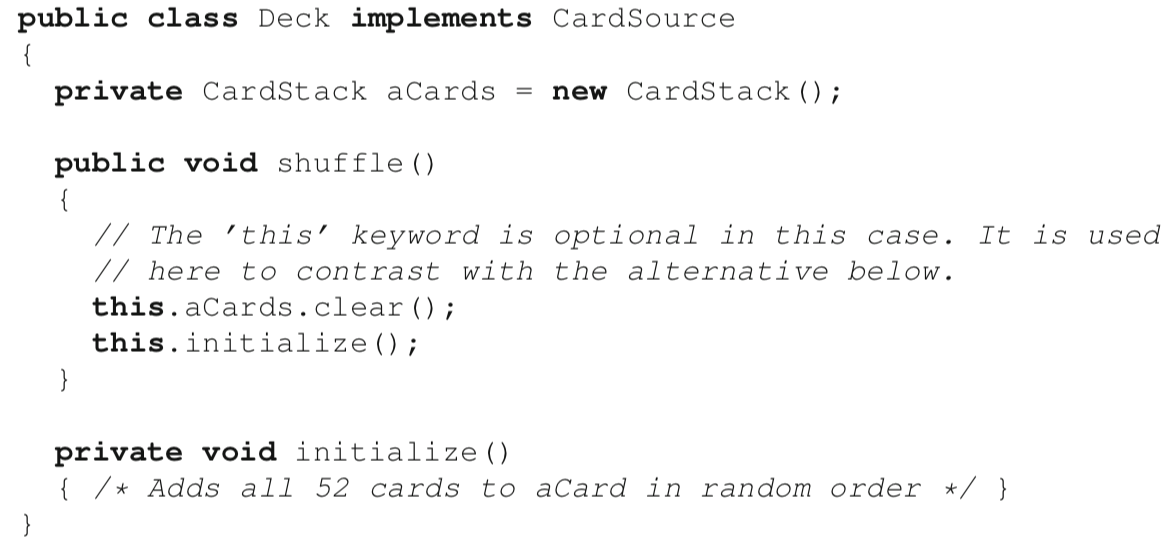
By default, methods of a superclass are applicable to instances of a subclass.



Note:

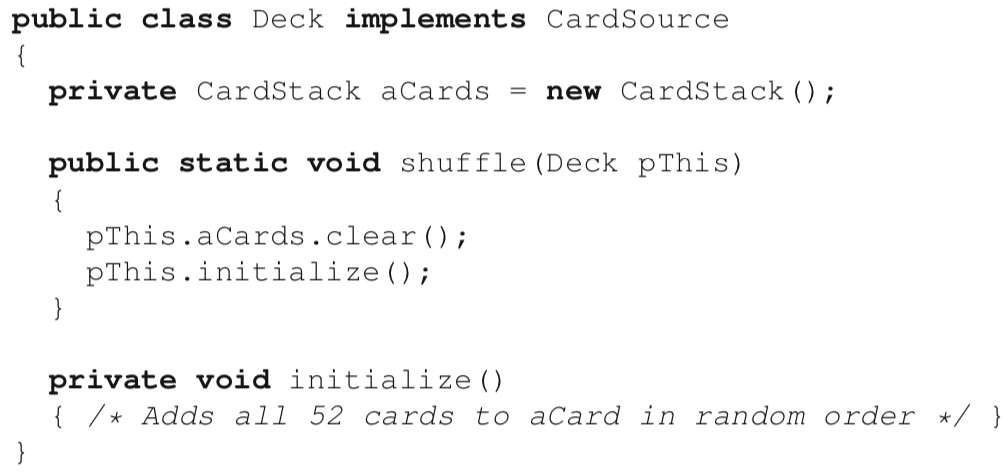
An instance (i.e., non-static) method is just a different way to express a function that takes an object of its declaring class as its first argument.

Case 1: non-static



* 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



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.

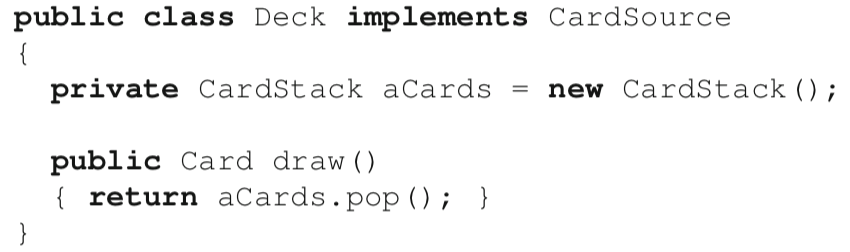
**Overriding**

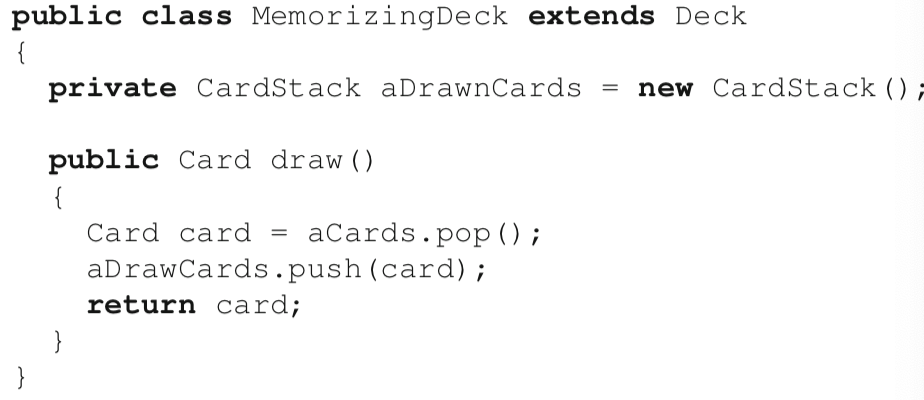
When multiple implementations are applicable, the run-time environment selects the *most specific one based on the run-time type of the implicit parameter.*

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.



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



****

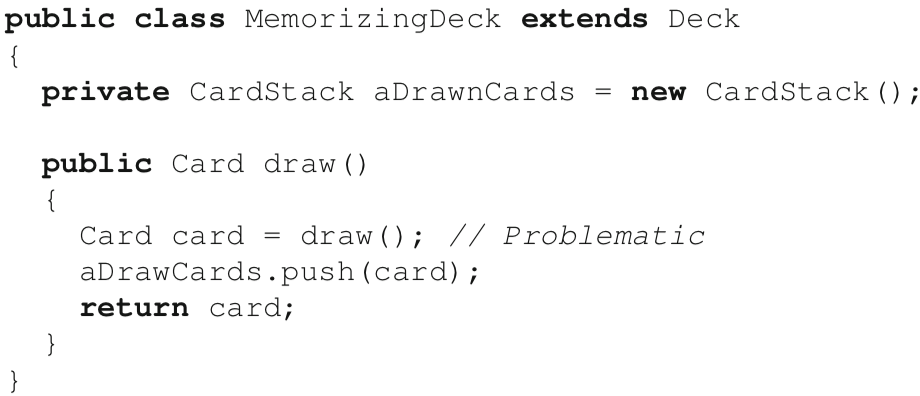
private 🡪 compile time error

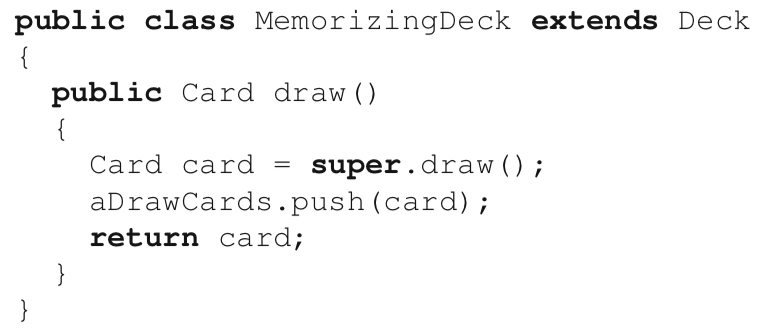
Solution

(a) declare aCards to be protected

(b) super

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.





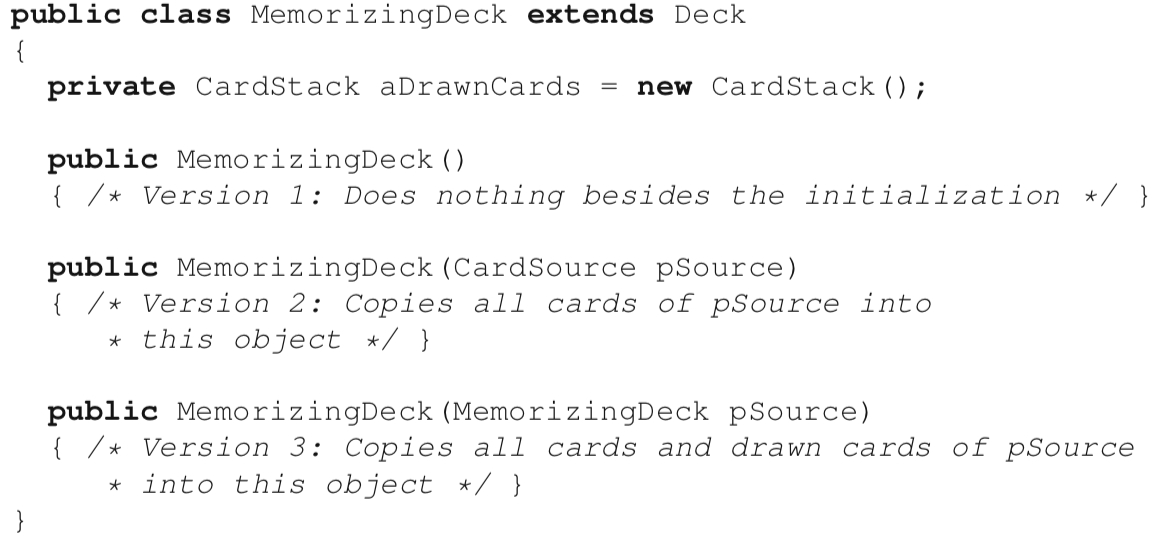
**Annotating Overridden Methods**

For a method to effectively override another one, it needs to have the same **signature** as the one it overrides.

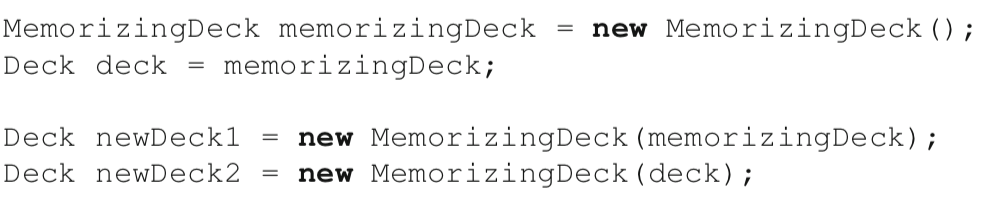
If a method annotated with @Override does not actually override anything, a compilation error is raised.

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.



not same(overloading)



version 3

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