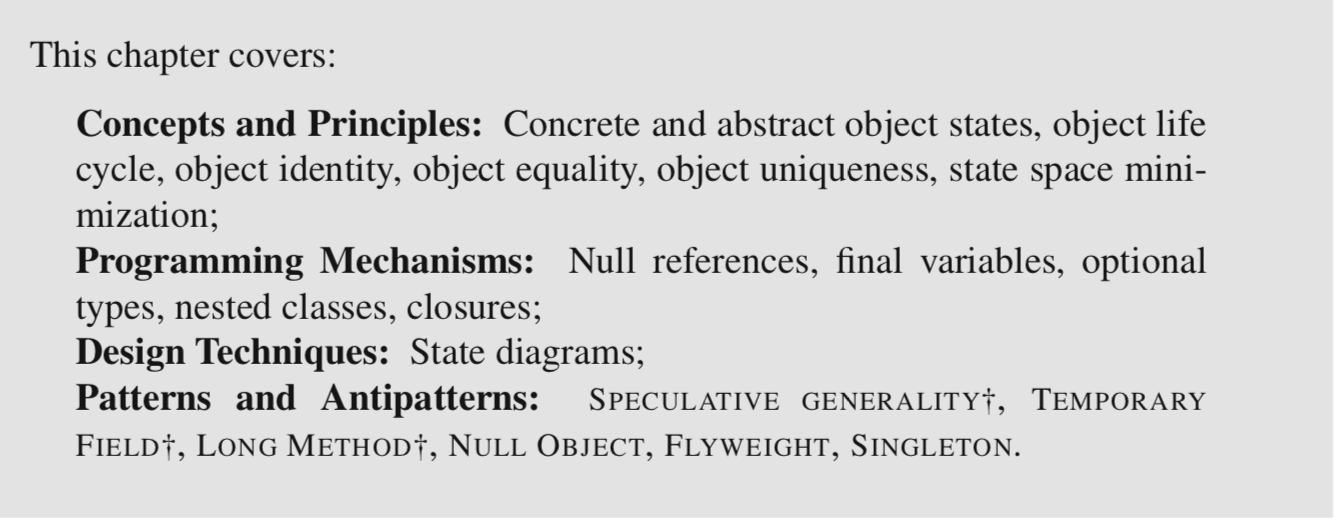
Chapter 4

Object State



4.2 Defining Object State

Informally, the state of an object refers to the particular pieces of information the object represents at a given moment. It is generally useful to distinguish between **concrete state** and **abstract state**. The concrete state of an object is the collection of values stored in the object’s fields.

We usually refer to *the set of possible states for a variable or object* as its **state space**. As soon as objects have fields of reference types, the cardinality (or size) of the state space explodes dramatically.

e.g. The state space of a Deck instance includes all possible permutations of any number of cards in the deck, a number in the range of 2.2×1068.

An abstract state is *an arbitrarily-defined subset of the concrete state space*.

[Examples]

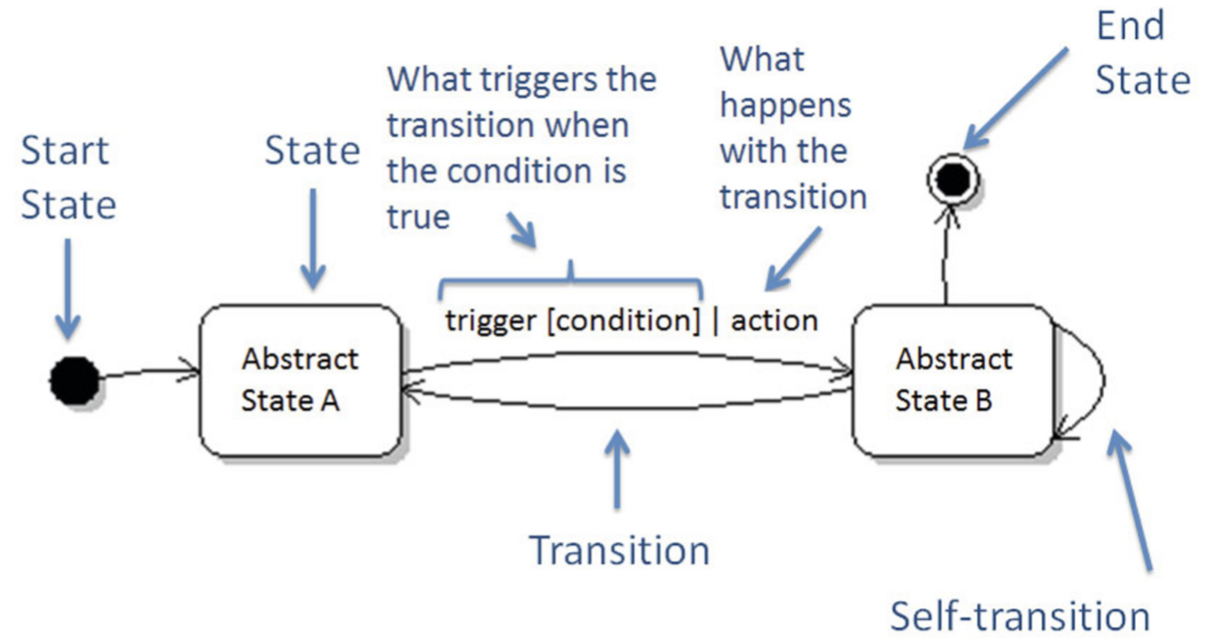
* Player: “even score” could be an abstract state for a Player instance that groups the roughly 231 states that represent a score that is an even number.
* Deck: “three kings” could represent any possible configuration of the deck where exactly three cards of rank Rank.KING are present.

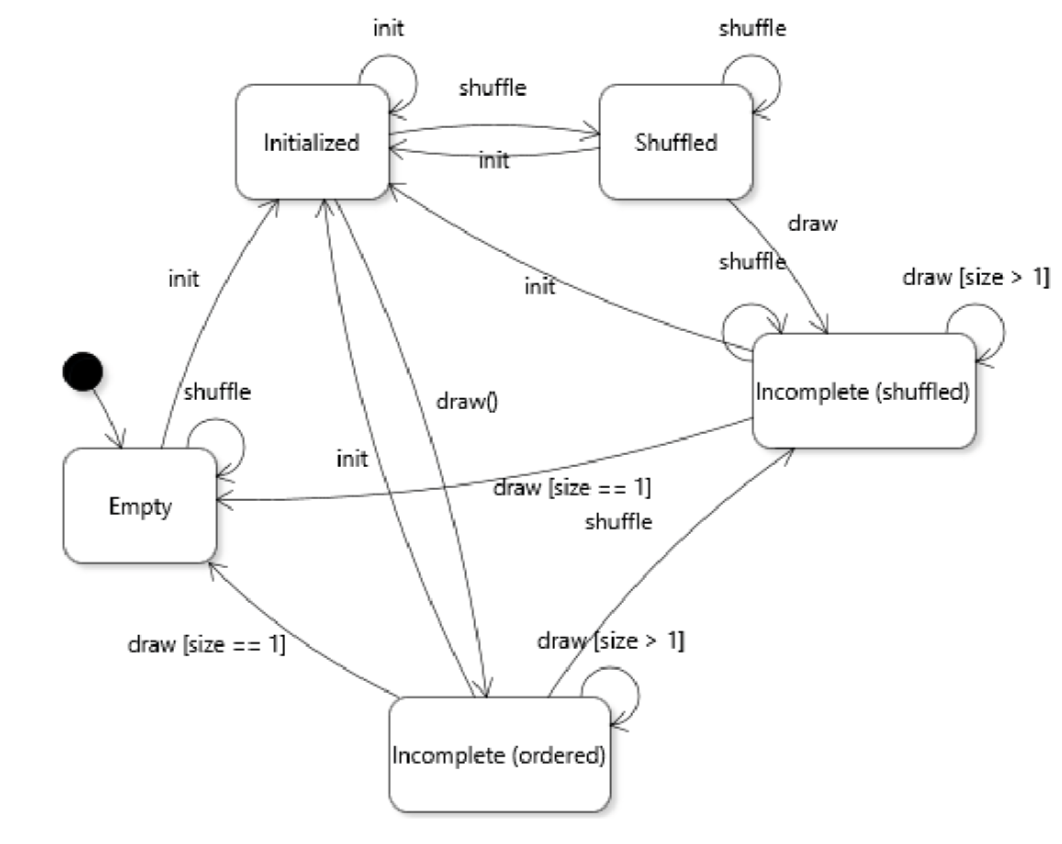
Some objects do not have any state. For example, function objects (3.4), often do not have any associated state. In this case, we talk about **stateless** objects; objects that have state as **stateful** objects. Another property of objects that is related to their state is **mutability** (2.6). In the case of immutable objects, the boundary between statefulness and statelessness becomes blurry, because in practice they only have a single state.

This chapter is concerned with objects that are both mutable and stateful.

4.3 State Diagrams

*UML state diagrams are useful to represent how objects can transition from one abstract state to another during their lifetime as a reaction to external events (typically, method calls)*. They represent a **dynamic view** of a software system.





4. 4 Designing Object Life Cycles

Objects with a complex life cycle are difficult to use, difficult to test, and their design and implementation is error-prone. A good design principle to avoid objects with complex life cycles is thus to **minimize the state space of objects** to what is absolutely necessary for the object to conduct its business.

Invalid and Useless States

Some states in the state space might simply be a consequence of how an object is designed or implemented, without there being any use for an object in that state in a given software system. For example, in a game of Solitaire, there is no use for an unshuffled deck of cards.

In some cases, eliminating some states from the life cycle of an object may seem like reducing the versatility of a class (“what if we need this one day?”). This is an argument that can be made in some situations. However, it is always important to consider the cost (in terms of software developer time spent understanding code, writing tests, and fixing bugs). Often, this kind of SPECULATIVE GENERALITY† is not worth the cost.

Unnecessary Stateful Information

To the extent possible, information should not be stored in an object unless it uniquely con- tributes to the intrinsic value represented by the object. A violation of this principle often constitutes an instance of the TEMPORARY FIELD† antipattern.

4. 5 Nullability

A null reference in the state of an object could be interpreted to mean:

1. That a variable is correctly but temporarily un-initialized, but is expected to become initialized in a different abstract state for the object. For example, in class Deck, we could assign to the field aCards the value null until the deck is shuffled;

2. That a variable is incorrectly initialized because the programmer overlooked a path through the code where the variable had to be initialized;

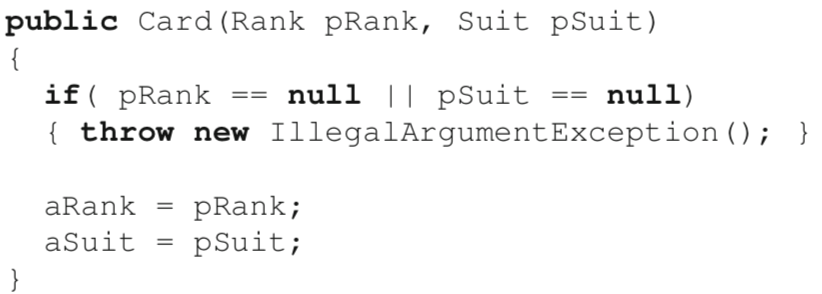
3. That the value is a flag that correctly represents the absence of a useful value in the normal life cycle of an object;

4. That the value is some sort of other flag that must be interpreted in a special way.

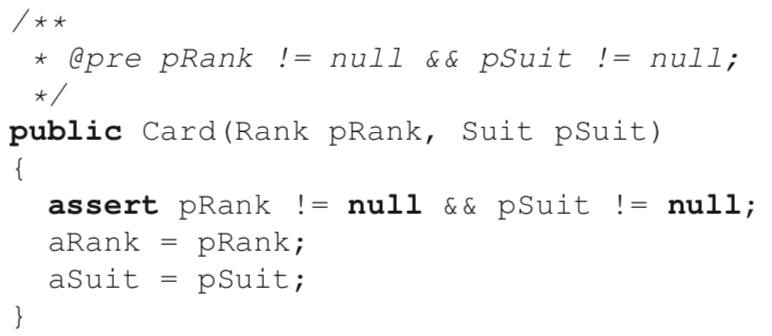
No Need to Model Absent Values

If it is possible to design a class to avoid any abstract state where a certain variable does not have a value, it is greatly desirable to design the class to forbid this eventuality.

[Input Validation]

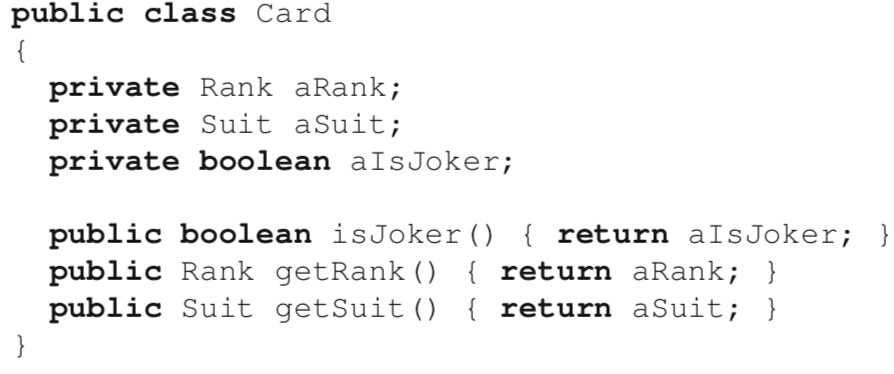


[Design by Contract]



Modeling Absent Values

In many situations, the domain concept we are trying to model will require that we make a provision for the fact that there may not be a value. As an example, let us consider a variant of class Card where an instance can also represent a “joker”. In many card games, a joker is a special card that has no rank and no suit.



* Null references: We could just ignore the advice offered in this section and assign null to aRank and aSuit. This means it would be possible to call (for example) card.getRank().ordinal() on a joker, and get a NullPointerException. Not good.
* Bogus values: We could assign an arbitrary, meaningless value for the rank and suit of a joker (e.g., ace of clubs). However, this is both confusing and dangerous. A part of the code could erroneously request the rank of a joker, and receive the value Rank.ACE, which makes no sense.
* Special values of an enumerated type: We could add an INAPPLICABLE enumerated value to both Rank and Suit, and assign these values to the corresponding fields for instances of Card that represent jokers. In my opinion, this solution is slightly better than the two above, but it still has some clear weaknesses.

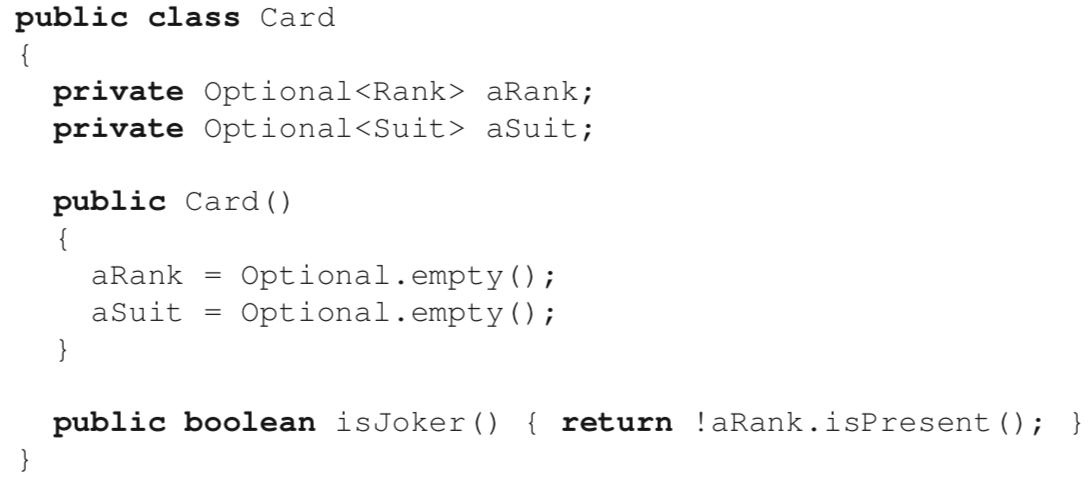
First, it is a conceptual abuse of the idea of enumerated types, where each value is enumerated. Although technically that is what it is, conceptually INAPPLICABLE is not a valid value in the enumeration, but rather a flag that indicates that we do not have a value.

Second, although we have four ranks and 13 suits, this solution will yield five and 14 enumerated values for each type, respectively. This discrepancy will muck up any code that relies on the ordinal values of these types (such as the initialization of a deck of cards), and introduce many opportunities for off-by-one errors.

Optional Types

The Optional type is a generic type that acts as a wrapper for an instance of type T, and which can be empty. To make a value of type T optional for a variable, we declare this variable to be of type Optional<T>.

To represent the absence of a value of the variable, we use the value returned by Optional.empty().



To create an instance of Optional that represents an actual value, we do Optional.of(value) if value is not (ever) expected to be null, and Optional.ofNullble(value) if value can be null (in which case Optional.empty() will be stored instead). To get the value wrapped by an instance of Optional, we call get().

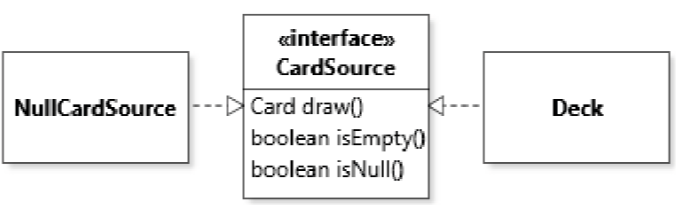
However, is that the two fields no longer have the types likely to be desired by the client code. While the client will probably be interested in working with values of type Rank and Suit, the fields of the class now store values of type Optional<Rank> and Optional<Suit>. To get around this issue, two main alternative are possible:

Change the interface of class Card so that getRank() and getSuit() return Optional<Rank> and Optional<Suit>, respectively. This requires client code to call get() everywhere the actual instance is needed, which can be cumbersome.

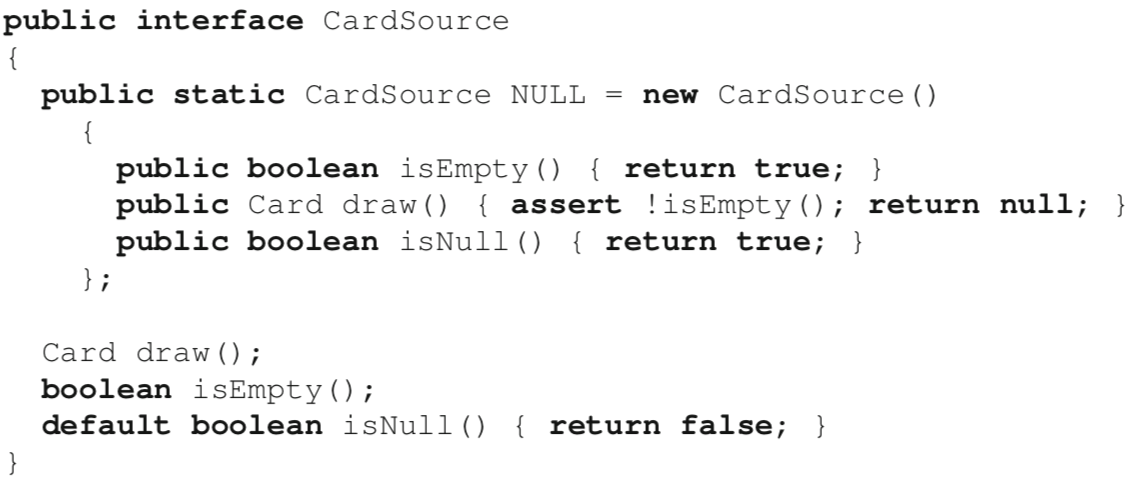
Unwrap the optional within getRank() and getSuit(), which preserves the interface but requires clients to ensure that they do not call the methods on a card that represents a joker (something that could be specified using design by contract, for example). This last solution is starting to look a lot like the use of null references, but it is technically safer, because calling get() on an empty instance of Optional will raise an exception immediately when the value is misused, as opposed to potentially propagating a null reference through the execution of the code.

The NULL OBJECT Design Pattern

The main idea of NULL OBJECT is to create one special object to represent the absent value, and to test for absence using a polymorphic method call.



With the features of Java 8, a NULL OBJECT solution can be implemented very efficiently by only modifying the interface at the root of the type hierarchy. First, we use a default method to avoid having to change all card source classes to implement isNull simply to return true. In Java, *default methods are methods that have an implementation in the interface which is applicable to instances of all implementing types.* To have a minimal impact on the rest of the code, we can also implement the NULL OBJECT as a constant in the interface by using an anonymous class.



With this solution, there is no longer a need for a separate NullCardSource class. Client code that must indicate an absent card source can simply use the reference available through CardSource.NULL instead. Because a NULL CardSource behaves just like any other card source, many special cases can be avoided.

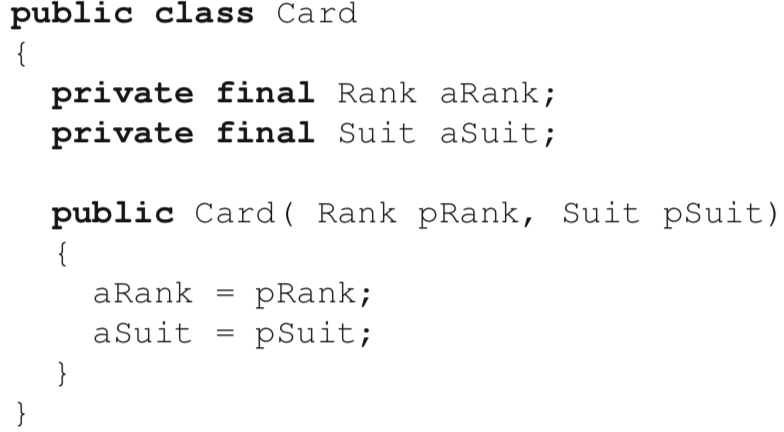
It is worth noting how this includes the call to draw: because calling draw on the NULL card source will automatically violate the precondition; the fact that we return a null reference afterwards is inconsequent.

In fact, in an ideal application of the NULL OBJECT, a check method such as isNull() should not even be necessary.

In our case, for example, because all client code that works with a card source must check for emptiness first, obtaining a NULL card source is indistinguishable from obtaining an empty one. For this reason, in practice it is very likely that we could dispense with the isNull() method altogether, and leverage polymorphism to its fullest to yield a simple and clean design without corner cases.

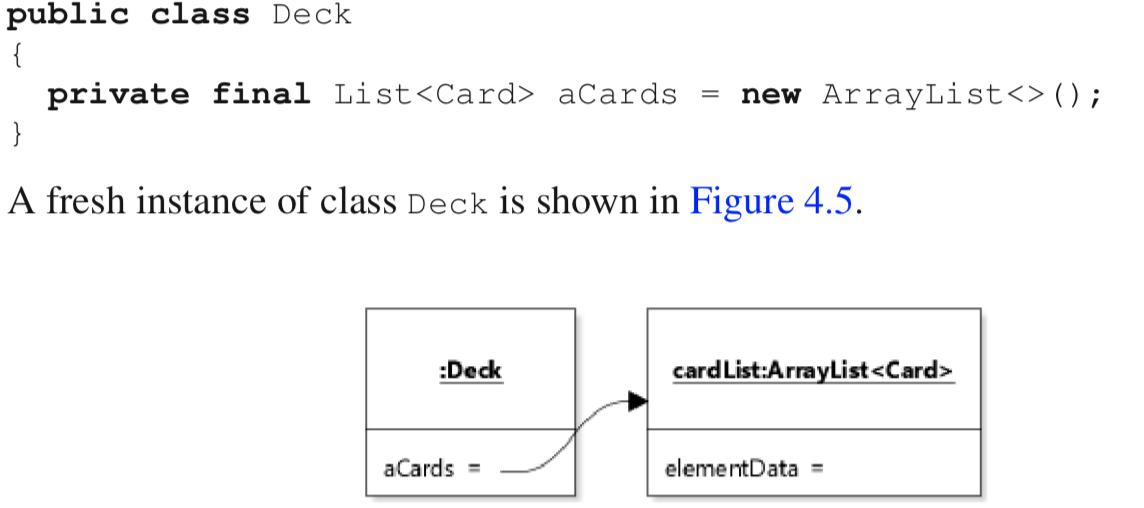
4.6 Final Fields and Variables

Because object state is just an abstraction of **the combination of values taken by the fields of an object**, the way to realize the principle in practice is to limit the number of ways in which the field values can be updated. 🡪 Prevent changing the value of a field after initialization, so that the value of the field remains constant.



Fields can be assigned a value only once, either in the initializer part of their declaration, or directly in the constructor (as in the example).

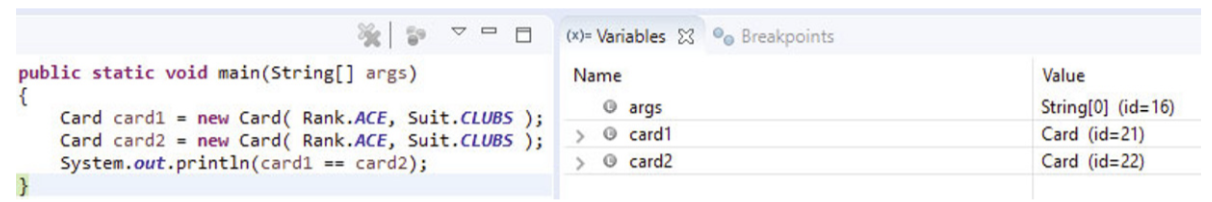
Note: For reference types, the value stored in a variable is a reference to an object. So, although it is not possible to reassign a final field, it is certainly possible to change the state of the object referenced (if the object is mutable).

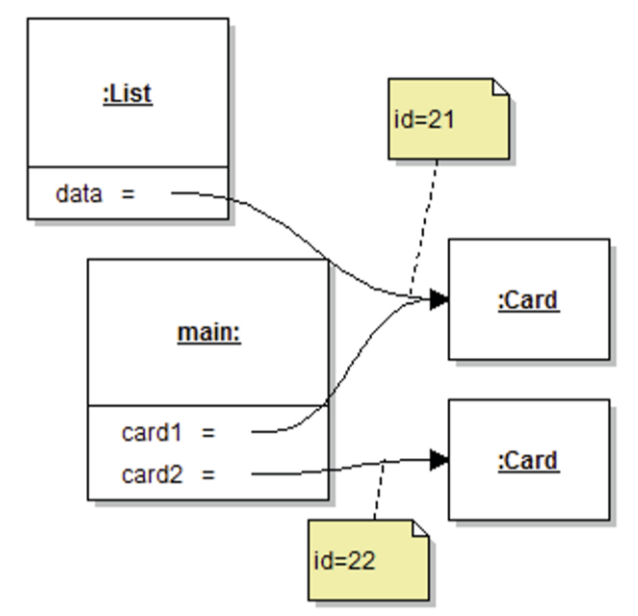


The discussion above was concerned mainly with **instance variables**. However, **local variables** (including method parameters) can also be declared to be final. [Local variables only exist for the duration of the execution of code in their scope.]

4.7 Object Identity, Equality, and Uniqueness

* Object Identity

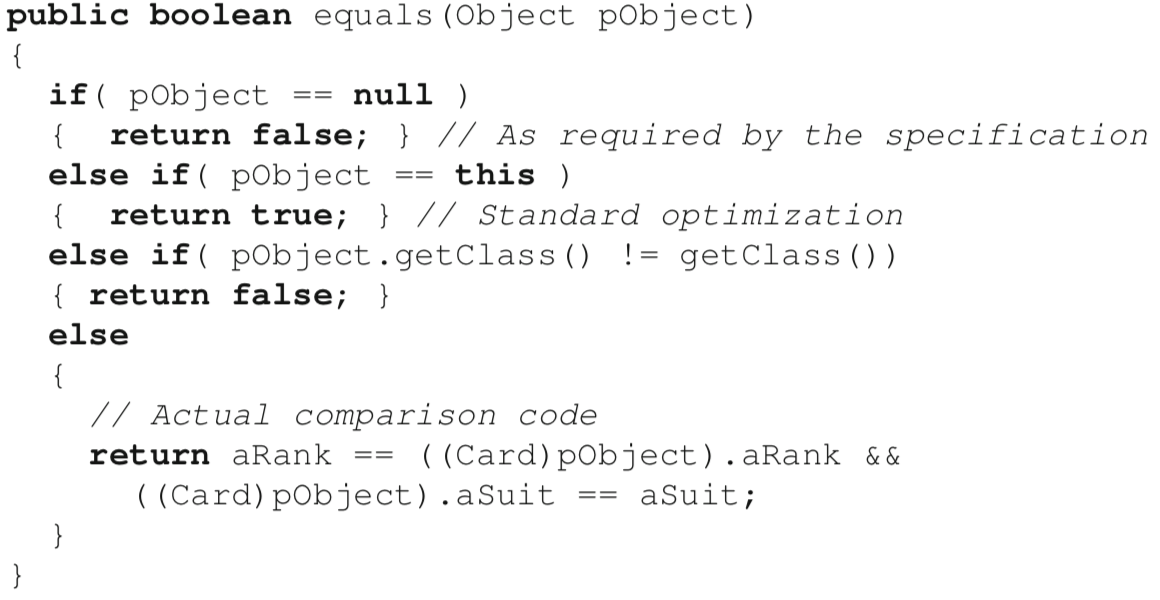




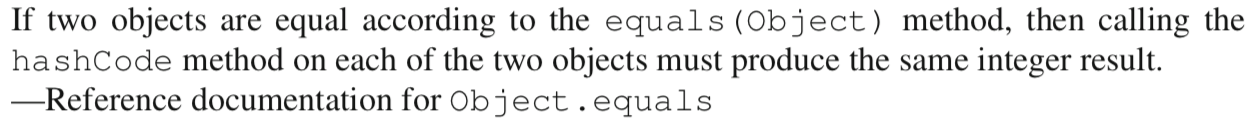
== operator returns true if the two operands evaluate to the same value. In the case of values of reference types, “the same value” means referring to the same object (identity). So here the statement returns false because, although both cards represent an ace of clubs, they are references to different objects.

* Object Equality

We can specify what it means for two objects of a class to be equal by overriding the equals(Object) method of the Object class. The default implementation of the equals method defines equality as identity. In other words, if the equals method is not redefined for a class, a.equals(b) is practically the same as a == b. [Except in the case where a == null. The == operator will correctly compare null values, but if a is null, a.equals(b) will raise a NullPointerException.]



[Convention] Any class that overrides equals must also override hashCode()



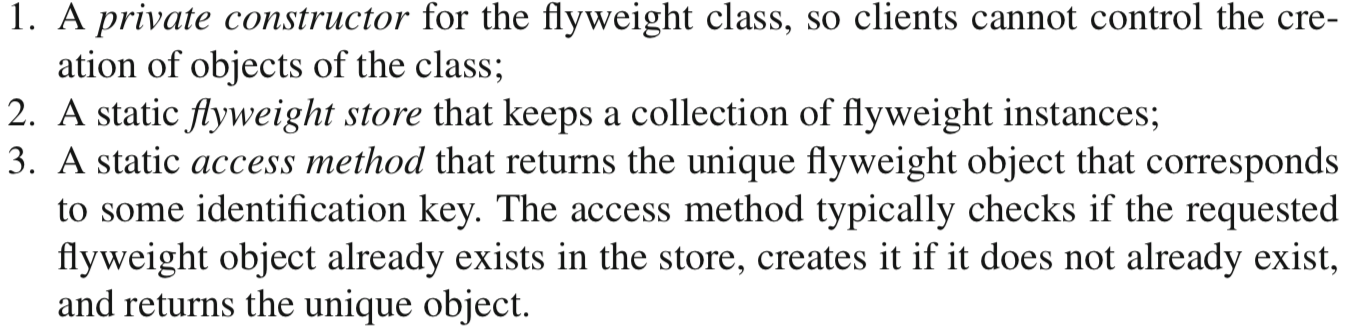
* Object Uniqueness

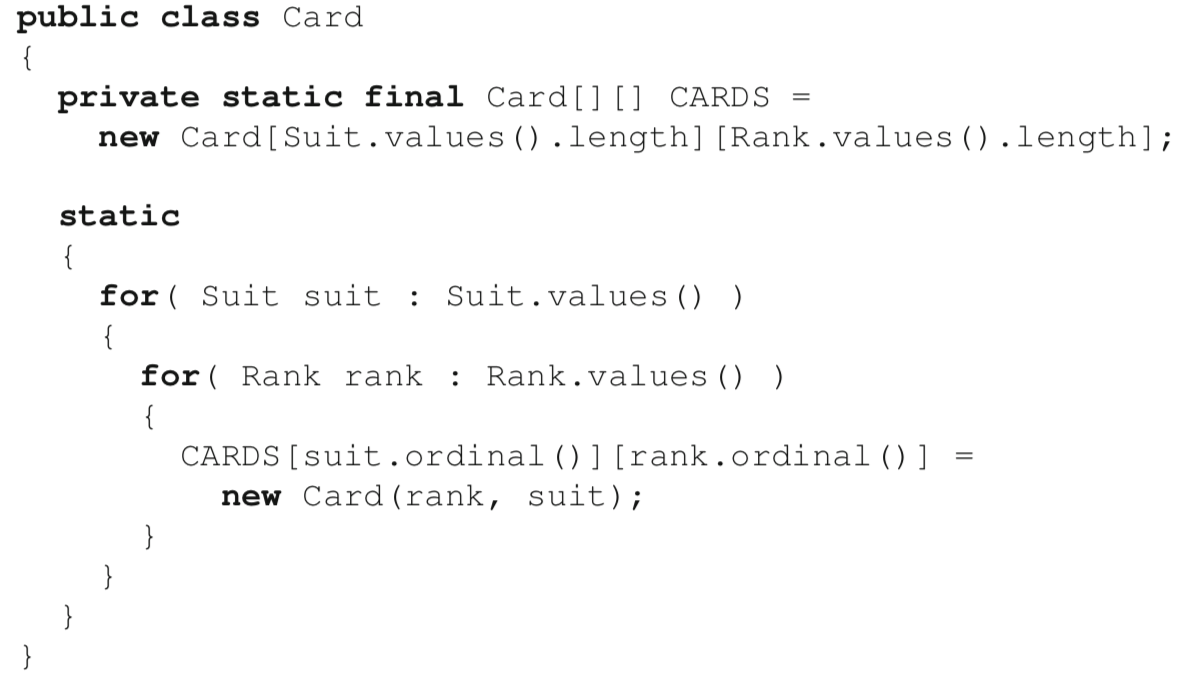
Objects of a class are unique if it is not possible for two distinct objects to be equal. If the objects of a class can be guaranteed to be unique, then we **no longer need to define equality**, because in this specific case, equality become equivalent to identity and we can compare objects using the **== operator**.

4. 8 The FLYWEIGHT Design Pattern

[Goal] Cleanly manage collections of low-level immutable objects. 🡪ensure uniqueness

[Context] Instances of a class are heavily shared in a software system.





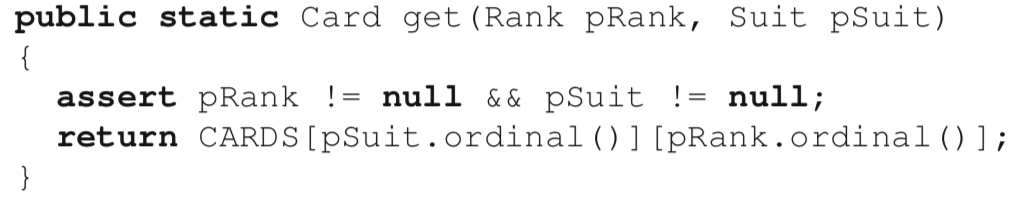
private

Static initializer block

A block of code that executes once, when the class is first loaded in the run-time environment.

The code above is only correct if it is placed within the scope of class Card because

CARDS is private.



access method

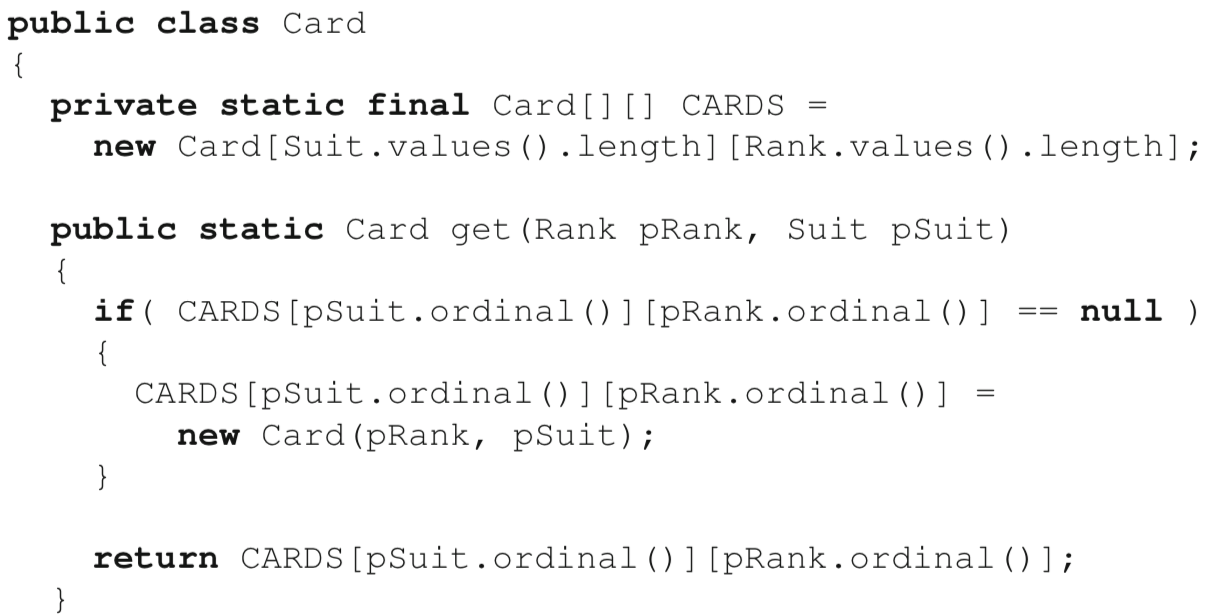
Note: The identification cannot be an instance of the flyweight object itself.



This would mean that to obtain a flyweight object of class Card, it would be necessary to already have that object. Because the only way to get a flyweight object should be through its access method, this scheme leads to an **infinite cycle**, and is thus flawed.

An important concern when implementing the FLYWEIGHT pattern is whether to pre-initialize the flyweight store, or whether to do this lazily, by creating objects as they are requested through the access method. The answer is context-dependent. In general, in cases where there exists a small and finite set of flyweights, it may make sense to pre-initialize them (as in the example). In other cases, additional logic must be added to the access method to **check if the object exists in the collection and, if not, create it based on the key.** In this latter case, the access method needs to be able to access all the information it needs to create the flyweight instance that corresponds to the requested key.

Instances are lazily created

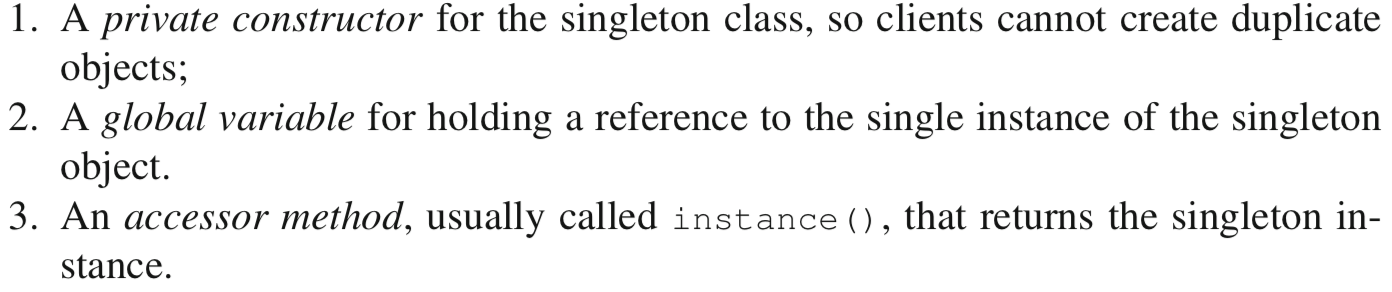


Note: that flyweight objects should be immutable to ensure that their uniqueness can be preserved, and that they can be shared and compared throughout the code.

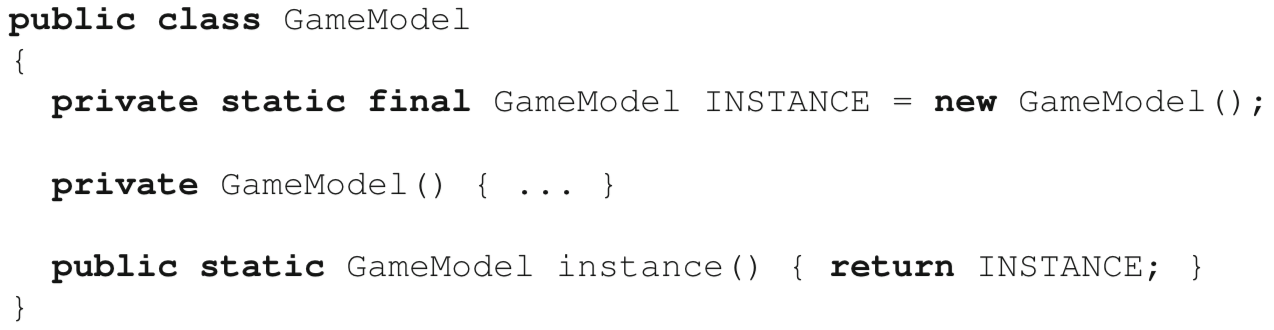
4. 9 The SINGLETON Design Pattern

[Goal] Ensure that there is only one instance of a given class at any point in the execution of the code.

[Context] The need to manage an instance that holds, in one place, a large or crucial amount of information that different parts of the code might be interested in.



e.g. the instance that represents the aggregated state of the game: the deck of cards and the various piles of cards in the game in progress



Difference between FLYWEIGHT and SINGLETON

SIGLETON attempts to guarantee that there is a *single instance* of a class, as opposed to *unique instances* of a class. Singleton objects are typically *stateful and mutable*, whereas flyweight objects should be *immutable*.

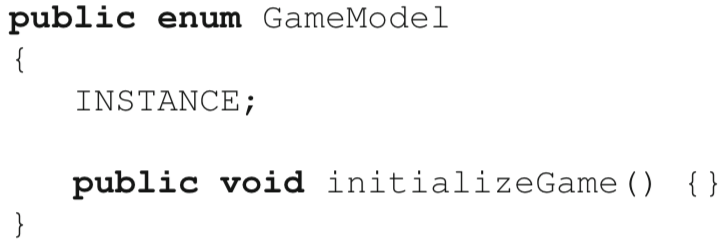
Note: A typical mistake when implementing the SINGLETON pattern is to store a reference to an instance of the class in a static field called INSTANCE or something like it, without taking proper care to prevent client code from independently creating new objects.

Solution 1

Private constructor

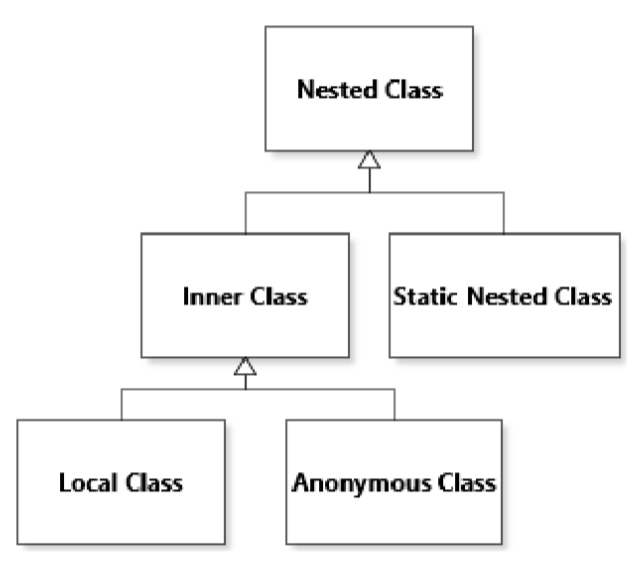
Solution 2

Enum type



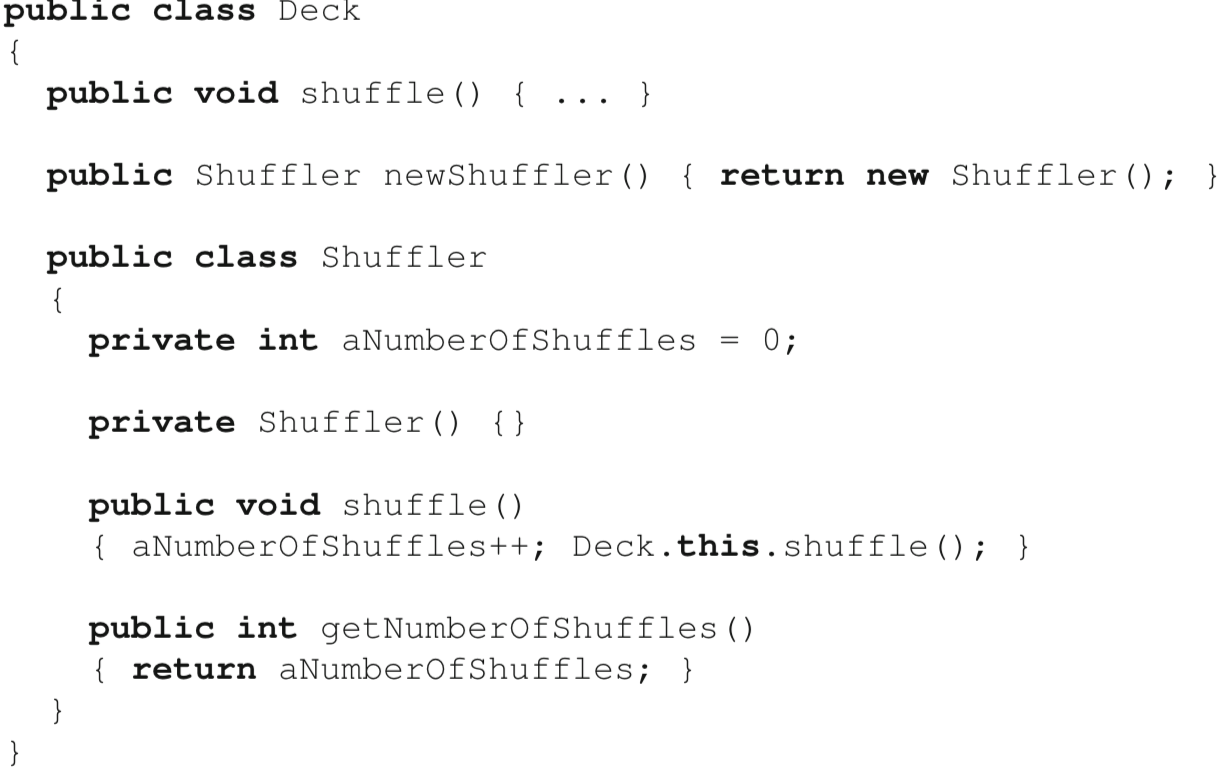
Comments: This technically works because the compiler will prevent the instantiation of enumerated types. Although this approach is represented as “preferred” in Effective Java, it is not without detractors. My conclusion is that this strategy is a trick that uses a programming mechanism (enumerated types) for an intent other that originally designed, and as such can be confusing. Here the type GameModel is not a finite set of values that represent different game models, which is what one would initially expect when seeing an enum designation. I thus recommend sticking to a private constructor to ensure the single instance constraint.

4. 10 Objects of Nested Classes



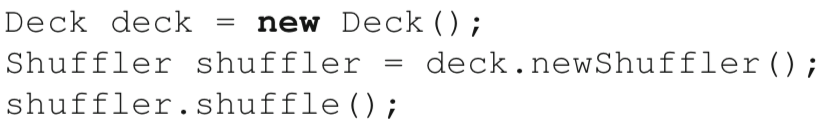
* Inner Class

Inner classes are declared within another class, and used to provide additional behavior that involves an instance of the enclosing class, but which for some reason we do not want to integrate into the enclosing class. As an example, let us say we want the option to remember how many times a certain Deck instance was shuffled.



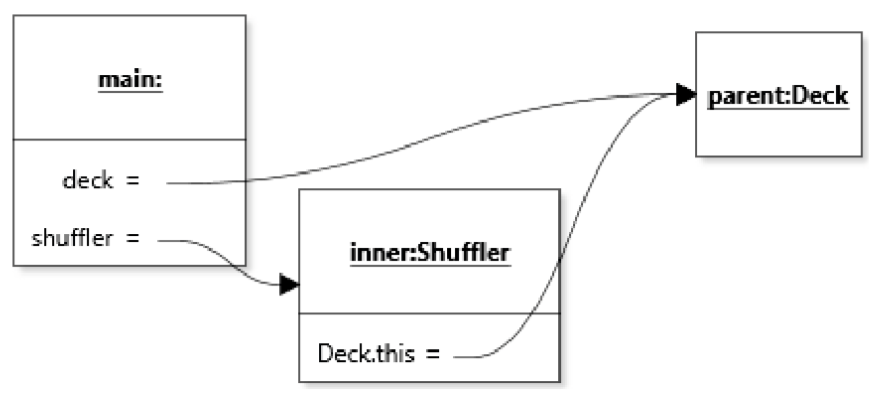
outer instance

Instances of an inner class automatically get a reference to the corresponding instance of their enclosing class (called the outer instance). The outer instance for an inner class is the object that was the implicit parameter of the method where the instance of the inner class was created.



The second line calls method newShuffler() on the instance of Deck referred to by variable deck. This method creates a new instance of Shuffler, whose outer instance will be the one referred to by deck. Within an inner class, the outer instance can be accessed through a qualified name that consists of the name of the class of the outer instance, followed by this. So, in our case, Deck.this refers to the outer instance of shuffler.

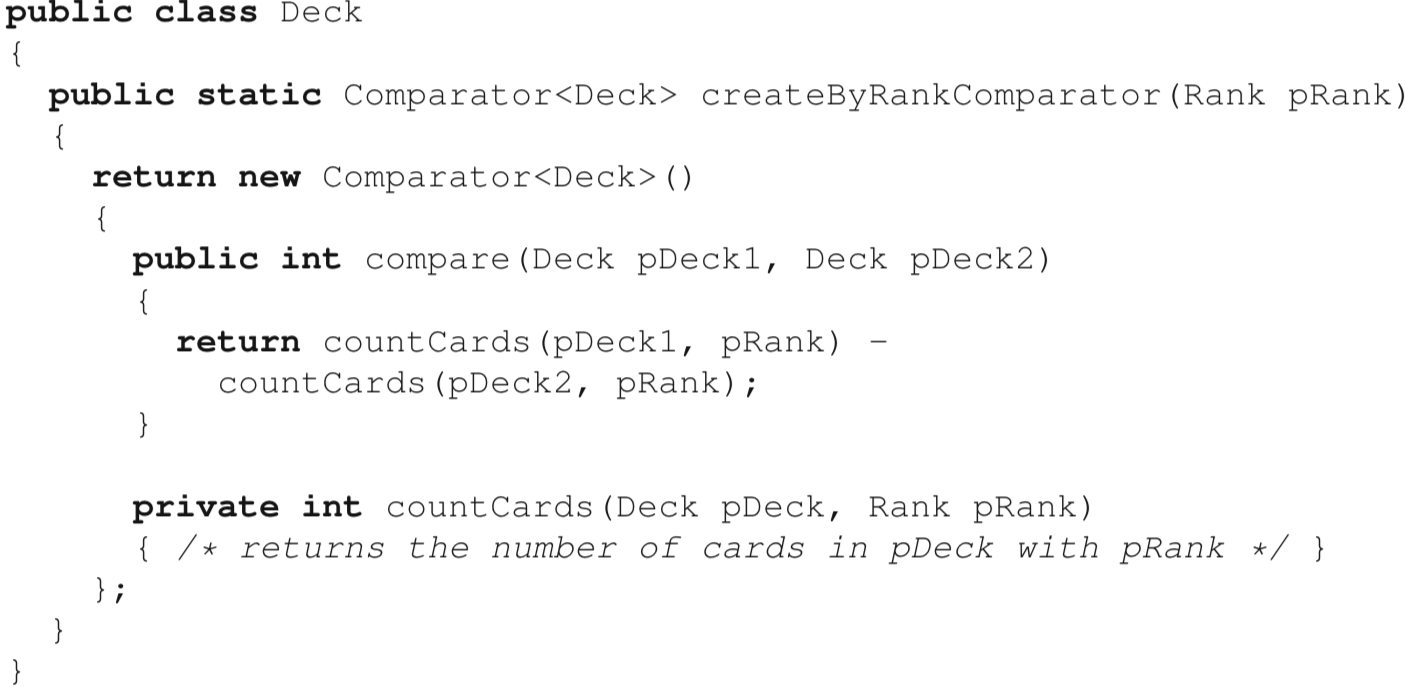
On the third line, it is the method shuffle() of the Shuffler instance that is called, but when this method executes, it then calls the shuffle() method of class Deck on the deck instance using Deck.this.



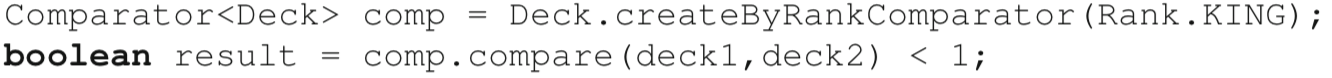
Java also allows the declaration of static nested classes. The main difference between static nested classes and inner classes is that static nested classes are not linked to an outer instance. As such, they are mostly used for encapsulation and code organization.

* Anonymous Class

Just like inner classes, local and anonymous classes also have implicit access to additional state information through a reference to their outer instance.



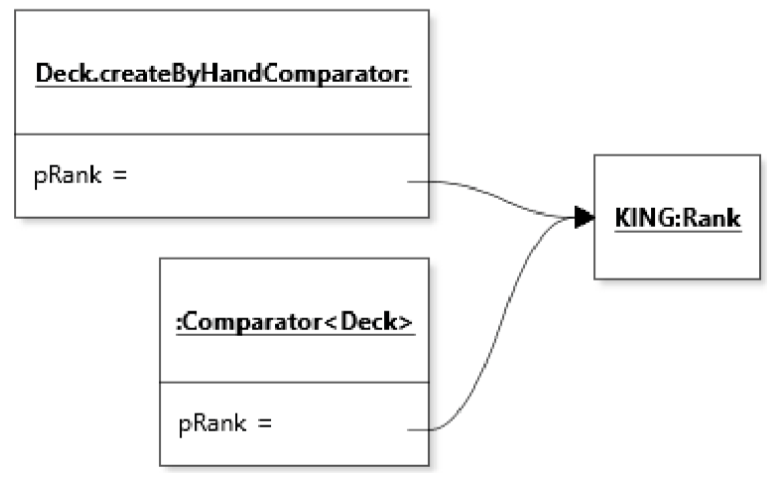
factory method



The code of method compare *declared inside the anonymous class* has access to the parameter pRank of createByRankComparator, which is *a separate method in a separate class*.

“Referring to variables in the parent method from an anonymous class”

When the compiler creates the definition of the anonymous class, it also (invisibly) adds fields to the anonymous class, and copies references to each of the local variables referenced in the code of the anonymous class’s method into a field. Thus, once an object of the anonymous class is created, the references to the local variables are now stored in fields of the same name in the anonymous class.



A method definition together with references to its local environment variables is commonly called a **closure**. In Java, anonymous classes and lambda expressions are not closures in the strict sense because they cannot modify the variables they reference.