Object and classes

------------------

NOTE

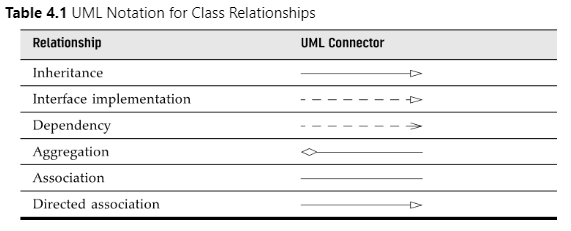
* Declaring a reference variable does not create an object
* A *final method*  cannot be overridden in subclass

Object and Class characteristics

* Class : blueprint, ex: car
* behaivior : method
* state : car engine, tire, door
* All objects that are instances of the same class share a family resemblance by supporting the same ***behavior***.
* Next, each object stores information about what it currently looks like. This is the object’s **state**.
* However, the state of an object does not completely describe it, because each object has a distinct **identity**.

Relationship between classes

* Dependence
* Aggregation
* Inheritance



NOTE

* It is important to realize that an object variable doesn’t actually contain an object. It only **refers** to an object.

Encapsulation

* The key to making encapsulation work is to have methods *never* directly access instance fields in a class other than their own.
* Programs should interact with object data *only* through the object’s methods.
* Sometimes, it happens that you want to get and set the value of an instance field. Then you need to supply *three* items:

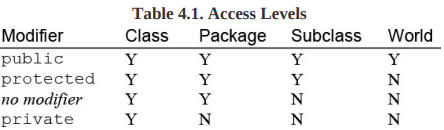
• A private instance field;

• A public field accessor method;

• A public field mutator method.

Mutator and Accessor methods

* In contrast, methods that only access objects without modifying them are sometimes called accessor methods. For example, LocalDate.getYear and GregorianCalendar.get are accessor methods.
* GregorianCalendar.add are mutator methods



Constructor

* A constructor can only be called in conjunction with the new operator.
* A constructor has the same name as the class.
* A class can have more than one constructor.
* A constructor can take zero, one, or more parameters.
* A constructor has no return value.
* A constructor is always called with the new operator.

NullPointerExceptions

Implicit and Explicit parameters

* public void raiseSalary(double byPercent)

{

   double raise = salary \* byPercent / 100;

   salary += raise;

}

* The raiseSalary method has two parameters.
* The first parameter, called the implicit parameter, is the object of type Employee that appears before the method name.
* The second parameter, the number inside the parentheses after the method name, is an explicit parameter. (Some people call the implicit parameter the target or receiver of the method call.)
* **this** refers to the implicit parameter.
* public void raiseSalary(double byPercent)

{

   double raise = **this**.salary \* byPercent / 100;

**this**.salary += raise;

}

NOTE

* Be careful not to write accessor methods that return references to mutable objects.
* As a rule of thumb, always use **CLONE** whenever you need to return a copy of a mutable field.
* class Employee

{

   private **Date** hireDay;

   . . .

   public **Date** getHireDay()

   {

      return hireDay; // BAD

   }

   . . .

}

Private fields

* When implementing a class, we make all instance fields private because public data are dangerous.

Final instance fields

* the field may not be modified again.
* private **final** String name;

Static fields & methods

* If you define a field as static, then the field is not present in the objects of the class.
* You can think of static fields as belonging to the class, not to the individual objects.

Static constants

* public static final double PI = 3.14

Private methods

* private methods are useful in certain circumstances.

NOTE

* As mentioned several times, it is never a good idea to have public fields, because everyone can modify them. However, public constants (that is, final fields) are fine. Since out has been declared as final, you cannot reassign another print stream to it:

Static methods

* Static methods are methods that do **NOT OPERATE ON OBJECTS**. For example, the pow method of the Math class is a static method.
* Static methods are the methods in Java that can be called without creating an object of class. They are referenced by the class name itself or reference to the object of that class
* Use static methods in two situations:

• When a method doesn’t need to access the object state because all needed parameters are supplied as explicit parameters (example: Math.pow).

• When a method only needs to access static fields of the class (example: Employee.advanceId).

Define static methods in the following scenarios only

* If you are writing utility classes and they are not supposed to be changed.
* If the method is not using any instance variable.
* If any operation is not dependent on instance creation.
* If there is some code that can easily be shared by all the instance methods, extract that code into a static method.
* If you are sure that the definition of the method will never be changed or overridden. As static methods cannot be overridden.

Factory methods

Main methods

Method parameters

* Two kind of method parameter: primitive types and object references
* Java use call by value
* The term ***call by value*** means that the method gets just the value that the caller provides.
* In contrast, ***call by reference*** means that the method gets the *location* of the variable that the caller provides. Thus, a method can *modify* the value stored in a variable passed by reference but not in one passed by value.
* A method cannot modify a parameter of a primitive type (that is, numbers or boolean values).
* A method can change the *state* of an object parameter.
* A method cannot make an object parameter refer to a new object.

You have seen that it is impossible for a method to change a primitive type parameter. The situation is different for object parameters.

Overloading

* var messages = new StringBuilder();
* var todoList = new StringBuilder('To do:\n');
* This capability is called *overloading*. Overloading occurs if several methods have the same name (in this case, the StringBuilder constructor method) but different parameters.

Default field initialization

* If you don’t set a field explicitly in a constructor, it is automatically set to a default value: numbers to 0, boolean values to false, and object references to null.
* Some people consider it **poor programming** practice to rely on the defaults.
* Certainly, it makes it harder for someone to understand your code if fields are being initialized invisibly.

Constructor with no arguments

* This constructor sets *all* the instance fields to their default values.

Initialization block

* The initialization block runs first, and then the body of the constructor is executed.
* // static initialization block

static {

nextId = generator.nextInt(10000);

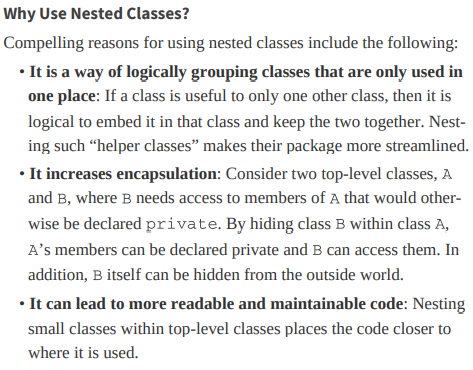
}

Object destruction and the finalize method

* cleanup code that may be needed when an object is no longer used
* Since Java does automatic garbage collection, manual memory reclamation is not needed, so Java does not support destructors.

NOTE

* Nested classes that are declared static are called **static** nested classes.
* Non-static nested classes are called **inner** classes.



Constructors: Canonical, Custom, and Compact

* The automatically defined constructor that sets all instance fields is called the canonical constructor.

Package access

* You have already encountered the access modifiers public and private. Features tagged as public can be used by any class.
* Private features can be used only by the class that defines them. If you don’t specify either public or private, the feature (that is, the class, method, or variable) can be accessed by all methods in the same package.

Documentation comments

* Class comments

/\*\*

\* A {@code Card} object represents a playing card,such

\* as 'Queen of Hearts'. A card has a suit (Diamond, Heart,

\* Spade or Club) and a value (1 = Ace, 2 . . . 10, 11 = Jack,

\* 12 = Queen, 13 = King)

\*/

public class Card

{

   . . .

}

* Method comments

/\*\*

\* Raises the salary of an employee.

\* @param  byPercent the percentage by which to raise the salary (e.g., 10 means 10%)

\* @return  the amount of the raise

\*/

public double raiseSalary(double byPercent)

{

   double raise = salary \* byPercent / 100;

   salary += raise;

   return raise;

}

* Field comments

/\*\*

 \* The 'Hearts' card suit

 \*/

public static final int HEARTS = 1;

**IMPORTANT NOTES!**

1. *Always keep data private.*

This is first and foremost; doing anything else violates encapsulation. You may need to write an accessor or mutator method occasionally, but you are still better off keeping the instance fields private. Bitter experience shows that the data representation may change, but how this data are used will change much less frequently. When data are kept private, changes in their representation will not affect the users of the class, and bugs are easier to detect.

1. *Always initialize data.*

Java won’t initialize local variables for you, but it will initialize instance fields of objects. Don’t rely on the defaults, but initialize all variables explicitly, either by supplying a default or by setting defaults in all constructors.

1. *Don’t use too many basic types in a class.*

The idea is to replace multiple *related* uses of basic types with other classes. This keeps your classes easier to understand and to change. For example, replace the following instance fields in a Customer class:

private String street;

private String city;

private String state;

private int zip;

with a new class called Address. This way, you can easily cope with changes to addresses, such as the need to deal with international addresses.

1. *Not all fields need individual field accessors and mutators.*

You may need to get and set an employee’s salary. You certainly won’t need to change the hiring date once the object is constructed. And, quite often, objects have instance fields that you don’t want others to get or set, such as an array of state abbreviations in an Address class.

1. *Break up classes that have too many responsibilities.*

This hint is, of course, vague: “too many” is obviously in the eye of the beholder. However, if there is an obvious way to break one complicated class into two classes that are conceptually simpler, seize the opportunity. (On the other hand, don’t go overboard; ten classes, each with only one method, are usually an overkill.)

Here is an example of a bad design:

public class CardDeck // bad design

{

   private int[] value;

   private int[] suit;

   public CardDeck() { . . . }

   public void shuffle() { . . . }

   public int getTopValue() { . . . }

   public int getTopSuit() { . . . }

   public void draw() { . . . }

}

This class really implements two separate concepts: a *deck of cards*, with its shuffle and draw methods, and a *card*, with the methods to inspect its value and suit. It makes sense to introduce a Card class that represents an individual card. Now you have two classes, each with its own responsibilities:

public class CardDeck

{

   private Card[] cards;

   public CardDeck() { . . . }

   public void shuffle() { . . . }

   public Card getTop() { . . . }

   public void draw() { . . . }

}

public class Card

{

   private int value;

   private int suit;

   public Card(int aValue, int aSuit) { . . . }

   public int getValue() { . . . }

   public int getSuit() { . . . }

}

1. *Make the names of your classes and methods reflect their responsibilities.*

Just as variables should have meaningful names that reflect what they represent, so should classes. (The standard library certainly contains some dubious examples, such as the Date class that describes time.)

A good convention is that a class name should be a noun (Order), or a noun preceded by an adjective (RushOrder) or a gerund (an “-ing” word, as in BillingAddress). As for methods, follow the standard convention that accessor methods begin with a lowercase get (getSalary) and mutator methods use a lowercase set (setSalary).

1. *Prefer immutable classes.*

The LocalDate class, and other classes from the java.time package, are immutable—no method can modify the state of an object. Instead of mutating objects, methods such as plusDays return new objects with the modified state.

The problem with mutation is that it can happen concurrently when multiple threads try to update an object at the same time. The results are unpredictable. When classes are immutable, it is safe to share their objects among multiple threads.

Therefore, it is a good idea to make classes immutable when you can. This is particularly easy with classes that represent values, such as a string or a point in time. Computations can simply yield new values instead of updating existing ones.

Of course, not all classes should be immutable. It would be strange to have the raiseSalary method return a new Employee object when an employee gets a raise.