# Alan Kay on formuleerinud

## **OOP 5 tähtsaimat printsiipi**

1. **Everything is an Object**. Iga objekt on omapärane muutuja – objekt (nagu iga muutuja) hoiab endas **andmeid**, mis määravad objekti oleku. Iga objektiga on aga lahutamatult seotud ka mingid funktsioonid (**meetodid**), mille kaudu saab objekti olekut (objekti käitumist!) muuta. Programmis võib objektiks kuulutada praktiliselt suvalise mõiste.
2. **Programm on objektide kogum, mis saadavad üksteisele sõnumeid**. Kui üks objekt soovib saata mõnele teisele objektile sõnumi (näit. päringu tema oleku kohta), siis see objekt pöördub teise objekti vastava meetodi (funktsiooni) poole.
3. **Igal objektil on oma isklik mälu, mis on reeglina nähtamatu teistele objektidele**. Seetõttu jääb objekti ehitus varjatuks *(hidden implementation*), sest objektiga saab reeglina suhelda ainult tema meetodite (objekti liidese, *interface*) kaudu.
4. **Iga objekt on konkreetset tüüpi**. Ehk teisiti – iga konkreetne objekt on mingi objektitüübi (**klassi**) ilming (*instance*). Klassi kasutaja tahab teada vastuseid küsimustele:
* mis teenuseid seda tüüpi objektid pakuvad?
* milliseid sõnumeid saab seda tüüpi objektile saata?
1. **Kõik sama tüüpi objektid on ühesuguse ehitusega**. Seetõttu kõik antud tüüpi objektid oskavad vastu võtta ühesuguseid sõnumeid.

**Üldisematest objektitüüpidest on võimalik tuletada (*derive*) alamtüüpe**.

Näiteks kui on loodud klass Kujund, siis on võimalik sellest objektitüübist tuletada objektitüüp Ring, mis **pärib** (*inherite*) kõik klassi Kujund omadused (andmed ja meetodid). Seetõttu oskab iga Ring-tüüpi objekt vastu võtta ka kõiki selliseid sõnumeid, mida oskavad käsitleda Kujund-tüüpi objektid.

Seega – kirjutades üks kord valmis klassi Kujund programmikoodi (meetodid), kandub see programmilõik automaatselt üle kõikide klassist Kujund tuletatud klasside programmikoodi!

Tuletamise-pärimise skeem ongi kõige tähtsam mehhanism, mis tagab OOP võimsuse ja efektiivsuse.

#### OOP KEELED

**Kolm omadust**:

1. **kapseldamine** (*encapsulation*) – keeles on vahendid, mis võimaldavad koondada andmed ja nendega tehtavad operatsioonid (meetodid) ühte programmilisse ühikusse (objektitüüp)

# NB! objekti andmetele saab reeglina ligi ainult meetodite

 kaudu!

1. **pärimine** (*inheritance*) – igast objektitüübist saab tuletada uusi objektitüüpe, kusjuures järglased pärivad eellase omadused (nii andmed kui meetodid)
2. **polümorfism** (*polymorphism*) – samanimelised meetodid võivad erinevatel objektidel teostuda erineval viisil

###  Objektitüüp Stack

pop

## view

push

init

OP 1

OP 2

## OP 4

OP 3

### Fields and methods

When you define a class (and all you do in Java is define classes, make objects of those classes, and send messages to those objects), you can put two types of elements in your class: ***fields*** (sometimes called data members), and ***methods*** (sometimes called *member functions*).

**A field is an object of any type that you can communicate with via its reference.**

It can also be one of the primitive types (which isn’t a reference). If it is a reference to an object, you must initialize that reference to connect it to an actual object (using **new**, as seen earlier) in a special method called a ***constructor*.**

If it is a primitive type, you can initialize it directly at the point of definition in the class. (As you’ll see later, references can also be initialized at the point of definition.)

Each object keeps its own storage for its fields; the fields are not shared among objects. Here is an example of a class with some fields: Feedback

**public** **class** Thing {

// fields (data members)

 String name;

 **int** year;

// constructor

 Thing(String n, **int** y){ // Konstruktor

 name=n;

year=y;

 }

// methods (member functions)

 **void** myMethod() { }

 **public** **static** **void** main(String[] args) {

 Thing ball = **new** Thing("Ball",2014); // call constructor

 Thing cup = **new** Thing("Cup",1999);

 }

## }

## You manipulate objects with references

You treat everything as an object, using a single consistent syntax. Although you *treat* everything as an object, the identifier you manipulate is actually a “**reference**” to an object.

You might imagine this scene as a television (the object) with your remote control (the reference). As long as you’re holding this reference, you have a connection to the television, but when someone says “change the channel” or “lower the volume,” what you’re manipulating is the reference, which in turn modifies the object. If you want to move around the room and still control the television, you take the remote/reference with you, not the television.

Also, the remote control can stand on its own, with no television. That is, just because you have a reference doesn’t mean there’s necessarily an object connected to it. So if you want to hold a word or sentence, you create a **String** reference:

String s;

**But here you’ve created *only* the reference, not an object.** If you decided to send a message to **s** at this point, you’ll get an error (at run time) because **s** isn’t actually attached to anything (there’s no television). A safer practice, then, is always to initialize a reference when you create it:

String s = "asdf";

However, this uses a special Java feature: strings can be initialized with quoted text. Normally, you must use a more general type of initialization for objects.

## You must create all the objects

When you create a reference, you want to connect it with a new object. You do so, in general, with the **new** keyword. The keyword **new** says, “Make me a new one of these objects.” So in the preceding example, you can say:

String s = new String("asdf");

Not only does this mean “Make me a new **String**,” but it also gives information about *how* to make the **String** by supplying an initial character string.

### Special case: primitive types

One group of types, which you’ll use quite often in your programming, gets special treatment. You can think of these as “primitive” types.

The reason for the special treatment is that to create an object with **new**—especially a small, simple variable—isn’t very efficient, because **new** places objects on the heap.

That is, instead of creating the variable by using **new**, an “automatic” variable is created that is *not a reference*. The variable holds the value, and it’s placed on the stack, so it’s much more efficient.

Java determines the size of each primitive type. These sizes don’t change from one machine architecture to another as they do in most languages. This size invariance is one reason Java programs are portable.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Primitive type** | **Size** | **Minimum** | **Maximum** | **Wrapper type** |
| **boolean** | — | — | — | **Boolean** |
| **char** | 16-bit | Unicode 0 | Unicode 216- 1 | **Character** |
| **byte**  | 8-bit | -128 | +127 | **Byte** |
| **short** | 16-bit | -215 | +215—1 | **Short** |
| **int** | 32-bit | -231 | +231—1 | **Integer** |
| **long** | 64-bit | -263 | +263—1 | **Long** |
| **float** | 32-bit | IEEE754 | IEEE754 | **Float** |
| **double** | 64-bit  | IEEE754 | IEEE754 | **Double** |
| **void** | — | — | — | **Void** |

## Guaranteed initialization with the constructor

You can imagine creating a method called **initialize( )** for every class you write. The name is a hint that it should be called before using the object. Unfortunately, this means the user must remember to call the method.

**In Java, the class designer can guarantee initialization of every object by providing a special method called a *constructor*. If a class has a constructor, Java automatically calls that constructor when an object is created, before users can even get their hands on it.** So initialization is guaranteed.

Method overloading

**Thus, *method overloading* is essential to allow the same method name to be used with different argument types**. And although method overloading is a must for constructors, it’s a general convenience and can be used with any method.

It’s as if when you don’t put in any constructors, the compiler says “You are bound to need *some* constructor, so let me make one for you.” But if you write a constructor, the compiler says “**You’ve written a constructor so you know what you’re doing; if you didn’t put in a default it’s because you meant to leave it out**.”

### Default constructors

As mentioned previously, a default constructor (a.k.a. a “no-arg” constructor) is one without arguments that is used to create a “basic object.” **If you create a class that has no constructors, the compiler will automatically create a default constructor for you**.

The **this** keyword

Suppose you’re inside a method and you’d like to get the reference to the current object.

Since that reference is passed *secretly* by the compiler, there’s no identifier for it. However, for this purpose there’s a keyword: **this**.

The **this** keyword—which can be used only inside a method—produces the reference to the object the method has been called for. You can treat this reference just like any other object reference.

Keep in mind that if you’re calling a method of your class from within another method of your class, you don’t need to use **this**. You simply call the method. The current **this** reference is automatically used for the other method.

### Scoping

Most procedural languages have the concept of *scope*. This determines both the visibility and lifetime of the names defined within that scope. In C, C++, and Java, scope is determined by the placement of curly braces **{}**. So for example: Feedback

{

 int x = 12;

 // Only x available

 {

 int q = 96;

 // Both x & q available

 }

 // Only x available

 // q “out of scope”

}

A variable defined within a scope is available only to the end of that scope.

### Scope of objects

Java objects do not have the same lifetimes as primitives. When you create a Java object using **new**, it hangs around past the end of the scope. Thus if you use:

{

 String s = new String("a string");

} // End of scope

the reference **s** vanishes at the end of the scope. However, the **String** object that **s** was pointing to is still occupying memory. In this bit of code, there is no way to access the object, because the only reference to it is out of scope.

### Using other components

**Whenever you want to use a predefined class in your program, the compiler must know how to locate it.**

Of course, the class might already exist in the same source code file that it’s being called from. In that case, you simply use the class—even if the class doesn’t get defined until later in the file (Java eliminates the “forward referencing” problem, so you don’t need to think about it).

**What about a class that exists in some other file?**

This is accomplished by telling the Java compiler exactly what classes you want by using the **import** keyword.

**import** tells the compiler to bring **in a package, which is a library of classes**. Most of the time you’ll be using components from the standard Java libraries that come with your compiler.

With these, you don’t need to worry about long, reversed domain names; you just say, for example:

import java.util.ArrayList;

to tell the compiler that you want to use Java’s **ArrayList** class.

However, **util** contains a number of classes and you might want to use several of them without declaring them all explicitly. This is easily accomplished by using ‘**\***’ to indicate a wild card:

import java.util.\*;

###

## The hidden implementation

It is helpful to break up the playing field into ***class creators*** (those who create new data types) and ***client programmers*** (the class consumers who use the data types in their applications).

**The goal of the client programmer** is to collect a toolbox full of classes to use for rapid application development.

**The goal of the class creator is to build a class that exposes only what’s necessary to the client programmer and keeps everything else hidden**.

**Why?** Because if it’s hidden, the client programmer can’t access it, which means that the class creator can change the hidden portion at will without worrying about the impact on anyone else. The hidden portion usually represents the tender insides of an object that could easily be corrupted by a careless or uninformed client programmer, so hiding the implementation reduces program bugs.

Kasutatava klassi liikmete NÄHTAVUS (*visibility*) kasutaja klassis

Klass User saab juba olemasolevat klassi Useful kasutada kahel erineval viisil:

1. komponendina:

**class** User {

Useful uc = **new** Useful(); // vms.

// ......

 }

1. baasklassina:

**class** User **extends** Useful { //...... }

Kasutatava klassi Useful LIIKMETE NÄHTAVUS kasutajaklassis User on esitatav järgmise tabelina:

|  |  |
| --- | --- |
| Klassi Useful liikme ligipääsu määraja (*access identifier*) | **class** User { ... } |
| **import** kuskilt.Useful; | Ei impordita - klass User on klassiga Useful samas *package*-is (kaustas) |
| **Kompositsioon:****new** Useful() | **Tuletamine:****extends** Useful |
| **private** | NO | NO | NO |
| määraja puudub (*package access*) | NO | NO | YES |
| **protected** | NO | YES | YES |
| **public** | YES | YES | YES |

**NB! Importida saab vaid avalikke (public) klasse!**

**public** **class** Thing {

// data members

 **private** String name;

 **private** **int** year;

// constructor

 **public** Thing(String n, **int** y){ // Konstruktor

 name=n; year=y;

 }

// methods (member functions)

 **public** **static** **void** main(String[] args) {

 Thing ball = **new** Thing("Ball",2014);

 Thing cup = **new** Thing("Cup",1999);

 }

}

### The static keyword

Ordinarily, when you create a class you are describing how objects of that class look and how they will behave. You don’t actually get anything until you create an object of that class with **new**, and at that point data storage is created and methods become available.

**But there are two situations in which this approach is not sufficient.**

One is if you want to have only one piece of storage for a particular piece of data, regardless of how many objects are created, or even if no objects are created.

The other is if you need a method that isn’t associated with any particular object of this class.

That is, you need a method that you can call even if no objects are created. You can achieve both of these effects with the **static** keyword.

When you say something is **static**, it means that data or method is not tied to any particular object instance of that class. So even if you’ve never created an object of that class you can call a **static** method or access a piece of **static** data.

With ordinary, non-**static** data and methods, you must create an object and use that object to access the data or method, since non-**static** data and methods must know the particular object they are working with.

Of course, since **static** methods don’t need any objects to be created before they are used, they cannot *directly* access non-**static** members or methods by simply calling those other members without referring to a named object (since non-**static** members and methods must be tied to a particular object).

To make a field or method **static**, you simply place the keyword before the definition. For example, the following produces a **static** field and initializes it.

class StaticTest {

 static int i = 47;

}

Now even if you make two **StaticTest** objects, there will still be only one piece of storage for **StaticTest.i.** Both objects will share the same **i**.Consider:

StaticTest st1 = new StaticTest();

StaticTest st2 = new StaticTest();

At this point, both **st1.i** and **st2.i** have the same value of 47 since they refer to the same piece of memory.

There are two ways to refer to a **static** variable.

As the preceeding example indicates, you can name it via an object, by saying, for example, **st2.i**.

You can also refer to it directly through its class name, something you cannot do with a non-static member. (This is the preferred way to refer to a **static** variable since it emphasizes that variable’s **static** nature.)

StaticTest.i++;

The **++** operator increments the variable. At this point, both **st1.i** and **st2.i** will have the value 48.

Similar logic applies to static methods. You can refer to a static method either through an object as you can with any method, or with the special additional syntax **ClassName.method( )**. You define a static method in a similar way: Feedback

class StaticFun {

 static void incr() { StaticTest.i++; }

}

An important use of **static** for methods is to allow you to call that method without creating an object.

This is essential, as we will see, in defining the **main( )** method that is the entry point for running an application.

Like any method, a **static** method can create or use named objects of its type, so a **static** method is often used as a “shepherd” for a flock of instances of its own type.

**ÜLESANNE**

**Klassis Thing nummerdada kõik loodavad objektid nende loomise järjekorras (alates 1).**