Object-Oriented Languages

Many of the most popular languages today are object-oriented (hereafter abbreviated as OO). Thus, it is important to understand how they work so that we can use them more effectively. This document describes the basic ideas and concepts behind OO languages, using Java as an example language in the family. These notes are, of course, also applicable to other statically (or mostly statically) typed object-oriented languages such as C++ or Modula-3. Dynamically-typed object-oriented languages, such as Smalltalk, are somewhat different and are not discussed in this chapter.

To understand OO languages we need to understand five main concepts: **classes**, **inheritance**, **overriding**, **subtyping**, and **dynamic dispatch**. We now discuss each of these in turn.

1.1. Classes and ADT

Classes are a mechanism for expressing abstract-data types (ADTs) in object-oriented languages. An ADT groups together data with the operations on the data. For example, the following class, called Point0, groups together the data needed to represent a point in two-dimensional space with the operations on it:

```
class Point0 {
  int x; int y;
  void getx() { return x; }
  void gety() { return y; }
  void setxy(int newx, int newy) { x = newx; y = newy; }
}
```

A class, such as Point0, looks a lot like a record (or struct in C/C++ terminology) except that it also has operations in it which records typically do not have. You can think Point0 as being a *blueprint*: it describes the properties of all **instances** of Point0. One can create an instance of a class by using a creation operation (e.g., **`new Point0()`** in Java/C++-like syntax). Instances of class **`Point0`** are often called objects of class **`Point0`**. Each instance has a *unique* name, called its **identity**. You can think of the identity as being analogous to a social security number: each person in the US has a unique social security number. Different languages implement object identity differently. Some languages, such as C++, use the memory address of an instance as its identity. Java does not use the memory address of an instance as its identity since the memory address of an instance might change if the garbage collector moves the instance. Instead, Java uses a more abstract notion that it calls a **reference**.

Each instance of **`Point0`** has its own private copy of the two instance variables **`x`** and **`y`**. Instance variables are analogous to fields. Object-oriented languages prefer to use the term “instance variables” instead of “fields” because “instance variables” is more explicit: an *instance variable* is a variable belonging to an *instance* of the class.

Each instance of **`Point0`** also gets its own private copy of the **methods**[1] **`getx`**, **`gety`**, and **`setxy.`** The methods are operations that one can perform on an instance of **`Point0`**. For simplicity, **`Point0`** has only three methods, but to be really useful, we would need to add more methods to it (e.g., a method to compute the distance between two points). The instance variables of a class are in scope in all the methods of the class. Thus, the methods of **`Point0`** can access **`x`** and **`y`**.

Now, we are ready to see some example code that uses **`Point0`**.

```
int main() {

/* Create a new instance of Point0 and put its identity
into p1.  */
  Point0 p1 = new Point0();
/* Create another instance of Point0 and put its identity
into p2. */
  Point0 p2 = new Point0();
/* Invoke setxy method of the instance whose identity is in
p1.  After this line p1 holds the identity of a Point0
instance whose instance variables x and y have value 11 and
12 respectively */
  p1.setxy(11, 12);
/* Invoke setxy method of the instance whose identity is in
p2.  After  this line p2 holds the identity of a Point0
instance whose instance variables x and y have value 21 and
22 respectively */
  p2.setxy(21, 22);
/* Assign the identity in p2 to p1.  After this line, p1
and p2 hold the identity of the same object whose x has
value 21 and y has value 22. */
  p1 = p2;
/* Prints 21 */
  print p1.x;
}
```

So far we have assumed that all instance variables and methods are publicly accessible. However, to support *abstract* data types, classes must also provide a mechanism for hiding their concrete internals. Languages such as Java, C++, and C# allow one to annotate instance variables and methods with **public** or **private** to indicate whether they are publicly accessible or hidden. For example, a better design of a Point class would hide **`x`** and **`y`** so that code using these points (called *clients*) such as the routine "**`main`**" above cannot access the internal representation directly.

```
class Point1 {
  private int x; private int y;
  public void getx() { return x; }
  public void gety() { return y; }
```

[1] This is not really true in any real implementation of an object-oriented language but we will assume it for now for simplicity.

```
  public void setxy(int newx, int newy) { x = newx; y =
newy; }
}
```

Since **x** and **y** are private, they are accessible only by methods of the same class. For example, if "**main**" above used **Point1** instead of **Point0**, it would not be able to execute **p1.x** since **x** is private and thus inaccessible outside the **Point1** class.

1.1.1 An Extended Example

Now that we have discussed the terminology and basic concepts behind classes, let’s consider a larger example, a stack for integer values. Stacks are a heavily used data structure, so it makes sense to write it so that it is easy to use for any client.

```
Class Stack0 {
  private int values[];
  private unsigned int size;
  private int firstFree;
  // Constructor: initializes a new instance of a stack
  public Stack0(unsigned int s) {
    if (s <= 0) error;
    values = new int[s];
    size = s;
    firstFree = 0;
  }
  // Another constructor: used when no argument is
explicitly passed
  public Stack0() {
    // Use the other constructor with the argument 100
    this(100);
  }
  public void push(int v) {
    if (firstFree < size) {
      values[firstFree] = v; firstFree = firstFree + 1;
    }
    else error
  }
  public int pop() {
    if (firstFree == 0) error
    else {
      firstFree = firstFree - 1;
      int retval = values[firstFree];
      return retval;
    }
  }
}
```

The above stack uses an array of **s** integers to store the values. A more sophisticated stack might grow the array as needed rather than throwing an error if a client pushes more than **s** elements on the stack.

The above stack class uses a new concept: a **constructor**. A constructor is just a special function that is executed for each new instance of the class. It is a great place for putting initialization code. For example, the constructor for **Stack0** creates an array to hold the contents of the stack.

Let's now consider a client of **Stack0**:

```
int main() {
  // Declare two variables that can hold the identity of a
stack
  Stack0 s1;
  Stack0 s2;
  // s1 holds the identity of a new stack with a size of
100
  s1 = new Stack0(100);
  // s2 holds the identity of a new stack with a size of 10
  s2 = new Stack0(10);
  // push 1 and 2 on stack whose identity is in s1
  s1.push(1); s1.push(2);
  // push 3 and 4 on stack whose identity is in s2
  s2.push(3); s2.push(4);
  // print 2
  print s1.pop();
  // print 4
  print s2.pop();
  s1 = s2;
  // print 3
  print s1.pop();
}
```

Note that main invokes **push** and **pop** on the instances of **Stack0** but cannot access **firstFree, values,** and **size** since they are private instance variables. To see why it is important to make these instance variables private, imagine if the instance variables were actually public. Let's suppose that a client wants to write a routine that enumerates over all the integers in the stack without actually popping all the integers off. A client may write a routine such as:

```
int printAll(Stack0 s) {
  int cur;
  for (cur = 0; cur < s.firstFree; cur++) {
    print s[cur]; cur = cur + 1;
  }
}
```

This works fine for now (assuming, of course, that the instance variables are actually public). Let's suppose the author of **Stack0** decides to change the internal representation of a stack so that instead of using an array of integers, it uses a linked list of integers. This is a reasonable change to make since using a linked list allows the stack to grow or shrink as needed. In other words, the maximum depth of a stack does not have to be fixed at the time an instance of a stack is created. If the author of **`Stack0`** makes this change, the code **`printAll`** breaks: it still expects an array as the internal representation.

The author of an ADT needs to decide for each method and instance variable whether it belongs to the **representation** of the ADT or to its **public interface**. The representation should be protected from external access (e.g., by marking it as "**`private`**"). The public interface is a promise that the author of the ADT makes to her clients: clients can assume that the public interface will not change, at least not in a way that breaks the clients' code. Commercial suppliers of library code try very hard to respect and maintain this distinction.

1.2. Inheritance

One of the major thrusts of software engineering especially over the last two decades has been code reusability. One should be able to write code once and use it many times. I remember one of the biggest frustrations while programming in C was that I had to keep writing basically the same code many times, each time with some small difference. Inheritance is one mechanism in object-oriented languages for facilitating code reuse.

For example, let's suppose that I need a stack that supports not just **`push`** and **`pop,`** but also supports **`top`**, where **`top`** returns the top integer in the stack without removing it from the stack. I *could* write a new stack package, called **`Stack1`**, that has **`push`**, **`pop`**, and **`top`** methods but that would be wasteful since the implementations of **`push`** and **`pop`** would be identical to those of the push and pop in a **`Stack0`**. I could avoid some of the effort by just using the copy and paste feature of my editor to copy the push and pop from a **`Stack0`** to **`Stack1`**. However, now I have two copies of the same code and thus, if I find a bug in my implementation of **`pop`**, I have to fix it in two places, rather than just one. A stack is such a generally useful data structure that if I have a large program, I may have many uses of a stack, each with slightly different capabilities. Thus, large programs further aggravate the situation; in a million line program (which is not uncommon nowadays) I may have to fix a hundred copies of **`pop`**!

Inheritance provides a way out of this dilemma by allowing a programmer to specify a new class which extends another class. For example, in Java I can define my **`Stack1`** as the following:

```
/* Define Stack1 to be an extension of Stack0; i.e., Stack1
has all the methods and instance variables of Stack0 plus
possibly some more */
class Stack1 extends Stack0 {
  /* Define the extension: a new top method defined in
terms of push and pop */
  public int top() {
    int retval = pop();
    push(retval);
```

```
      return retval;
    }
  }
```

Using this new stack is simple:

```
  int main() {
    Stack0 s0 = new Stack0();
    Stack1 s1 = new Stack1();
    s0.push(10);
    s1.push(20);
    // Prints out 20 20
    print s1.top(), s1.top();
    // Error since s0, of type Stack0, has no top method
    print s0.top();
  }
```

`Stack1` is called a **subclass** of `Stack0` and `Stack0` is called a **superclass** of `Stack0`. By extending `Stack0`, `Stack1` **inherits** all the behavior of `Stack0`, but is free to define additional behavior (such as `top` in our example) of its own. In our example, `Stack1` only added a new method but it could have added new instance variables.

It is worth reemphasizing that `Stack1` is able to reuse the code in `Stack0` without making a copy of it: a big software engineering victory!

1.3. Overriding

In our example above, the subclass simply added a new method. However, sometimes a subclass may want to modify a superclass' method. For example, let's suppose you need to write a stack that not only has `push` and `pop` methods but also counts the number of times something is pushed onto the stack. You could, of course, write your new stack independently of the existing stacks but that would result in a maintenance nightmare: you would end up with multiple copies of the same code that you need to maintain. Overriding allows a subclass to customize a superclass' method.

```
  class Stack2 extends Stack0 {
    private int npushes = 0;
    public void push(int v) {
      npushes = npushes+1;
      /* Use the superclass's push method */
      super.push(v);
    }
    public int getNPushes() { return npushes; }
  }
```

`Stack2` extends `Stack0` and gives its own implementation of the `push` method. `Stack2`'s `push` method increments the count of the number of pushes and then uses "super" to invoke the superclass' `push` method which performs the actual push. Clients can use instances of `Stack2` exactly the same way as instances of `Stack0` except that it also supports a new method, `getNPushes`, which returns the number of pushes performed on the stack instance.

A method that may be overridden is called a **virtual** method. A non-virtual method cannot be overridden. In Java, all methods are virtual by default. In C++ all methods are non-virtual by default; one needs to explicitly label a method as *virtual* in order to make it virtual. The decision of whether or not to make a method virtual should be based on whether or not a method makes sense to override. The issue of whether or not to make a method virtual is a deep one that software engineers argue over!

In this section we have assumed that an overridden method must have the same signature (argument and return types) as the superclass. A language that requires this is said to have an **invariant** type system because it does not allow the subclass to vary the type of any overridden method. We will later discuss the reason why some languages use an invariant type system while others do not.

It may be natural to ask at this point if a class can inherit from (i.e., extend) more than one class. It turns out that some (e.g., C++) languages allow this and some (e.g., Java) do not. To fully understand the reason for this, we first need to finish our discussion of the two remaining key concepts behind OO languages: subtyping and dynamic dispatch.

1.4. Subtyping

When creating subclasses, a client may wish to extend the superclass in many ways:

1. Add a new method
2. Change the body of a superclass method
3. Add an argument to a superclass method
4. Change argument types of the superclass method
5. Change the return type of a superclass method
6. Add a new instance variable
7. Modify the type of an existing instance variable
8. Change the access control (public, private) of an instance
9. Delete a method
10. …

For each of the above, I'm sure you can think of a motivating example. However, despite their apparent usefulness, modern languages only allow some of the above possibilities. For example, Java allows only the first two possibilities. By restricting what how programmers can extend classes using inheritance, language designers are able to provide subtyping and subclassing using the same mechanism: inheritance. In other words, language designers often design languages so that if **`S1`** is a subclass of **`S2`**, then **`S1`** is also a subtype of **`S2`**. Tying together these two distinct concepts, subtyping and subclassing, comes at a cost. We will discuss that later.

Let's look at some of the extensions described above and determine if they lead to subtypes.

- Let's suppose we create a new subclass of **`Stack0`**, called **`Stack1`** which adds a **`top`** method (as already done above). **`Stack1`** supports all the behavior of **`Stack0`**. Namely, it has a **`push`** and a **`pop`** method just like the **`push`** and **`pop`** method in **`Stack0`**. Thus, **`Stack1`** can be used whenever **`Stack0`** is expected. More concretely, one can substitute an instance of **`Stack1`** any place where an instance of **`Stack0`** is expected. In formal notation, **`Stack1`** <: **`Stack0`**. It may be natural to ask if **`Stack0`** <: **`Stack1`**. For this to be the case, it must be possible to substitute an instance of **`Stack0`** when an instance of **`Stack1`** is expected. This will not work because a **`Stack1`** has a top operation that **`Stack0`** does not support.

- Let's suppose we create a new subclass of **`Stack0`**, called **`Stack2`** which changes the body of the push method in **`Stack0`** (as already done above). As in the case above, **`Stack2`** supports all the behavior of **`Stack0`**, though for the **`push`** method it may execute different code. Thus, an instance of **`Stack2`** can be used whenever an instance of **`Stack0`** is expected. In formal notation, **`Stack2`** <: **`Stack0`**. It is important to note here that subtyping considers only the types and not the implementation of methods: thus even though **`Stack2`** will execute different code on a push than a **`Stack0`**, it is still a subtype of **`Stack0`** because the types are compatible. It may also be worth asking if **`Stack0`** <: **`Stack2`**. It turns out that using the same argument as above, **`Stack0`** should be considered a subtype of **`Stack2`**. However, for simplicity, languages such as Java, C++, etc., do not support **`Stack0`** <: **`Stack2`** because that would mean that a subclass is a supertype of its superclass. (Aside: if we use *structural subtyping*, which to my knowledge is something that only researchers talk about, then **`Stack0`** would actually be a subtype of **`Stack2`**).

- Let's suppose we create a new subclass of **`Stack0`**, called **`Stack3`**, which changes the argument type of **`push`** from **`int`** to **`boolean`**. In this case, **`Stack3`** cannot be used when a **`Stack0`** is expected since **`Stack0`** expects an integer as an argument to **`push`** while **`Stack3`** expects a **`boolean`** (assume that there are no conversions between integers and booleans). Thus **`Stack3`** is not a subtype of **`Stack0`**. It actually turns out that in type theory one can change the type of an argument (in a specific way) while still maintaining subtyping between subclass and superclass.

- Let's suppose we create a new subclass of **`Stack0`**, called **`Stack4`**, that changes the return type of **`pop`** from **`int`** to a **`subrange`** (say **`[1 TO 10]`**). In this case

Stack4 can still be used where a **Stack0** is expected. This is actually a consequence of the arrow rule. To see this, consider the following example:

```
Stack0 getS0() {
  Stack0 s0 = new Stack0();
  Stack0.push(5);
}
Stack4 getS4() {
  Stack4 s4 = new Stack4();
  s4.push(5);
}
void main() {
  int x;
  /* The code below expects that getS0 will return a
Stack0, and thus the pop will return an int.  Let's suppose
we substitute getS4 for getS0; in other words, we are
substituting a value of type Stack4 where a value of type
Stack0 is expected.  This still works because the pop will
return a subrange which is legal to assign to x */
  x = getS0().pop();
}
```

You should work through the remaining extensions yourself and make sure you understand them!

1.5. Dynamic Dispatch

Consider the following method:

```
void f(Stack0 s) {
  s.push(10);
}
```

Since **Stack2** is a subtype of **Stack0**, I can pass not only an instance of **Stack0** to this function but also an instance of **Stack2**. However, **Stack0** and **Stack2** have different implementations of the **push** virtual method; which one will be invoked?

The answer: it depends! If you pass an instance of **Stack0** to **f**, then the push will use **Stack0**'s push implementation. If you pass an instance of **Stack2** to **f**, then the **push** will use **Stack2**'s push implementation! In other words, at compile time you cannot know (in general) which implementation will be used for push: it depends on the class of the instance actually passed in to **f.** This is called dynamic dispatching because the "dispatch" is dynamic (i.e., happens at run time).

1.6. Challenges

The discussion so far applies all non-dynamically-typed object-oriented languages. Most languages include additional mechanisms besides the ones discussed above. Since these additional mechanisms are usually designed to fix some problem with the above model, it is worth looking at them in more detail.

1.1.2 Encapsulation and Inheritance

Classes along with the information hiding that they provide supports *encapsulation.* By encapsulation I mean that the internal state in a class is hidden from the outside and may be accessed only via the public interface (which is made up of **`public`** methods). Encapsulation is very desirable since it makes classes easier to understand. Moreover, one can modify the internals of a class without affecting clients since other classes cannot access the internals; they can only use the public interface. For example, as we saw above, since the internal representation of **`Stack0`** is inaccessible to clients except through the public interface, one can change the internal representation without breaking clients.

Inheritance can, however, create some problems for encapsulation.

Let's suppose a class **`C`** has a print method that you would like to reuse in two classes that you are writing. Then you could define your class as the following:

```
class MyClass1 extends C { … }
class MyClass2 extends C { … }
```

In this case you used inheritance for code reuse. However, you ended up with not just code reuse but subtyping relationships: **`C`** <: **`MyClass1`** and **`C`**<: **`MyClass2`**. This relationship is unintended and to see why this is a bad thing, consider a client that writes the following code:

```
C a[] = new C[10];
a[0] = new MyClass1();
a[1] = new MyClass2();
```

In other words, even though you used inheritance only for code reuse, the client is exploiting the subtyping relationship to store objects of type both **`MyClass1`** and **`MyClass2`** into an array of **`C`**.

Now let's suppose you decide that you discover another class, **`D`**, and want to reuse the print method in **`D`** instead of **`C`** for **`MyClass1`**. In other words, you change the implementation of **`MyClass1`** to be:

```
class MyClass1 extends D {…}
```

From your perspective you are changing just the implementation of your class. This *should not* affect any clients because you are using encapsulation which should allow you to change the implementation without affecting clients. However, in this case, it *does* affect the clients: the client cannot put an instance of MyClass1 into the array anymore.

The above problem arises because most modern object-oriented languages use inheritance for *both* code reuse and subtyping rather than using a separate mechanism for each. Java's interfaces helps with this issue. C++'s private inheritance also helps

1.1.3 Multiple inheritance

So far we have assumed that a class can *directly* inherit only from one superclass. However, some languages, notably C++, allow a class to inherit from multiple superclasses. Bjarne Stroustrup, the designer of C++, motivates multiple inheritance with the following example:

> "A fairly standard example of the use of multiple inheritance would be to provide two library classes displayed and task for representing objects under the control of a display manager and co-routines under the control of the scheduler, respectively. A programmer could then create classes such as:
> **`class my_displayed_task : public displayed, public task {…}"`**

[*The design and evolution of C++*, Page 258]

In other words, my_displayed_task is a subclass of both displayed and task (C++ does not use the "extends" keyword of Java; instead it uses ":").

One can imagine many other uses of multiple inheritance. Even then, most languages today do not support multiple inheritance. The reason for this is the complexity that multiple inheritance introduces in both the language and the implementation. It is a classic tradeoff between expressivity and simplicity.

To get a feel for some of the complexity of multiple inheritance, consider the following code (assume that all methods are virtual). Note that I'm using Java's "extends" notation rather than C++'s ":" since I find the Java notation to be more readable.

```
class A {
  void f() {…};
}
class B extends A {
  void g() {…}
}
class C extends A {
  void g() {…}
}
class D extends B, C {
  …
}
```

The problem is that class D is getting the f and g methods through both its immediate superclasses (B and C). So what happens if you invoke either of these methods on an instance of class D? Which implementation will be used? C++'s solution to this problem is to force programmers to explicitly indicate which implementation they want to use if there is any ambiguity. This adds complexity to the language and for programmers; some language designers feel it is worth it and some do not.