CSE305 Week 11: From ADT to OOP

First, to finish some other points of coverage of chapters 9 and 10:

Example on Passed Subprograms (Sebesta, section 9.6)

Here's a rewrite of Sebesta's code-sketch example in section. It reorders the functions, renames them according to my football pass analogy, and uses how Javascript (now) allows declaring and initializing in the same line of code:

```
function sub1() {
   let x = 1;
   function football() {   //code that gets passed
      alert(x);
   }
   function receiver(theBall) {
      let x = 4;
      theBall();
   }
   function launchPass() {
      let x = 3;
      receiver(football);
   }
   launchPass();
}
```

Of Sebesta's "three choices":

- Using the environment of the call statement that enacts the passed subprogram (called **shallow binding**) would take x = 4 in the body of `receiver`.
- Using the environment of the definition of the passed subprogram (called **deep binding**, which I would also call **statically-scoped binding** here) would go with where "football" appears in the textual code, so this would print x = 1.
- Using the environment of the call statement that passed the subprogram (called **ad-hoc binding**) would use the body of `launchPass` and thus give x = 3.

Only statically-scoped binding enables the compiler to determine the referent of `x` before any code is run, and thus generate the activation frame. Sebesta points out that if `function football()` were defined within `launchPass` just before the statement passing it to `receiver`, then both deep binding and ad-hoc binding would give x = 3 (which could be what the code writer intended). Sebesta then says that (otherwise) "ad-hoc binding has never been used---because, one might surmise, the environment in whcih the procedure appears as a parameter has no natural connection to the passed subprogram." Maybe it helps to understand this by picturing where it would make sense to **inline** the code body "alert(x)". It arguably doesn't make sense to execute it in `launchPass` because the code is supposed to be executed when the receiver figuratively catches the ball. (Yet, by "eager evaluation", if what we passed was `receiver(football())` with a function call, or in a language with an internal `eval` function, the call `receiver(eval("alert(x)"))`, then that is where the `x` *would* be read as x=3.)

Implementing Dynamic Scoping (Sebesta 10.6)

Implementing dynamic scoping is easy in concept---to look up a non-local variable, you follow the chain of dynamic links (DL) rather than the chain of static links (SL). Last week's notes included an example. The text observes that unlike with static scoping, the search through stack frames is an open-ended search, because the currently-running frame could be called from different places after different sequences of subprogram calls. This "deep access" policy is more time-consuming than static scoping. The text goes on to mention a shortcut called "shallow access" (I'm not bolding these names because they're silly and IMPHO actually misleading, so forgetaboutem) which is just the idea of keeping a separate record of changes in value for each named variable---which in turn is unnecessary with static scoping.

One remark looking forward is that the action of following links to do lookup at runtime is reprised in OOP to look up the applicable version of an overridable method in a class hierarchy, based on the actual type of the object a base-class variable happens to be holding. This, too, is an open-ended search---and the extra time matters a *lot* more because virtual method calls happen more often than well-motivated uses of non-local variables anyway.

The one program element that has to be dynamically scoped is exceptions. This and event handling are the subjects of Chapter 14, but we've been covering the workings of exceptions on-the-fly. Here's another example---to the effect that the dynamic lookup of an exception handler in a caller completely pops off and wipes out the calling frames along the way. (Event handling does not have the call-sequence element---an event does not invalidate the stuff in-between---so IMPHO it is best regarded as going hand-in-hand with OOP, as in fact was its historical genesis in Smalltalk and the other work at Xerox PARC.)

Example on Exceptions

Here is a tweak to some Python code used for the Assignment 5 *TopHat* part---changes in red:

```
def decodeRating(rtg):
   return rtg if (rtg/0 > 0 and rtg != 1000) else "Unrated"

#rating = 3
```

```
try:
   rating = decodeRating(1000)
except ZeroDivisionError:
   print("Nan's rating is ", rating)

if rating > 0:
   print(rating)
```

This handles the divide-by-zero exception, but raises the error that the name `rating` is not defined. The reason is that the stack frame for `decodeRating` got wiped by the divide-by-zero error, and likewise the `try:` block which called it. Not only was the assignment of a value never completed, the storage object for `rating` was wiped too. Because Python is interpreted, it keeps the association between names and storage objects, not just the binding address as with statically compiled languages. Thus Python reports the error as the name being absent.

Blocks As Quasi-Frames (Sebesta 10.5)

In C/C++/Java/Javascript, one can create a nested scope by enclosing any sequence of statements inside curly braces **`{ ... }`**. This is what happens naturally for the "then" and "else" branches of an `if-else` statement---declarations inside those branches do not apply after the `if-else` statement ends for the simple reason that the branch might not be executed. The same is true of the **`{...}`** body of a loop (and this is irrespective of whether the compiler permissively allows "`int i`" (etc.) declared in the loop *header*, which is outside the braces, to be visible after the loop). (What's even more important to appreciate here is that the alternatives in the body of a `match` in OCaml---and in Scala too---are separate scopes nested in the whole construct.)

Much the same thing happens with the `let-in` construct in OCaml (which works the same as `let-in-end` in standard ML, which Sebesta could have mentioned here). The declarations in the part before `in` are visible only in the body after it. The whole thing can see variables and functions in the enclosing scope. The idea is to use convenient names in a temporary manner in the body without worrying about their obscuring other entities with the same name after the body exits. We've seen this used `via let open CE in ...` with our example modules so that their shorter names do not clash. Another common use is to define a recursive helper function that does the "heavy lifting" between `let` and `in`, while putting the ultimastely-desired simple call to it in the body after `in`.

The text makes a point of noting that since simple blocks (here, maybe as opposed to blocks in loops) are one-shot deals that enter and exit in sequence, the compiler can know to re-use the memory space allocated to the vvariables declared in the first one for the variables of the second one, and so on. This is IMPHO a minor point---maybe it was more major in the 1970s---sitting under what for me is the helpful point about *all* blocks, including those of if-then-else and loops: they can be pictured as being implemented with the same mechanics and nonlocal-lookup rules as stack frames.

Pointers to Functions (Sebesta 9.7)

In C the native syntax for declaring a pointer `p` to a function `A foo(B x)` is

```
A (*p)(B).
```

The pointer name goes inside the type and the parameter name(s) are omitted. Sebesta gives an example with two parameters: `float (*pfun) (float, int)`. Because the type syntax is clunky, it is standard practice to use a typedef to name and declare function pointer types:

```
typedef float (*FI2F) (float, int);
```

Then a pointer `p` to any function of type `float * int -> float` can be declared simply as

```
FI2F p;
```

Either way, the function can be executed through the pointer by, e.g., `(*p)(3.5, 2)`; Because the name of a function is actually synonymous with the binding address of its object code, the syntax was liberalized to allow omitting the `*` from the type declaration and then the applications. C# cleaned up the syntax a little more by having you write `delegate` up front, e.g.

```
public delegate float pfun(float, int);
```

The presence of "`public`"---as well as my use of capital letters `FI2F` for my float-and-int-to-float type name---hints at the general drift: higher-order functions in these languages are best pictured as manipulating **function objects**, which are objects of a class that defines a method to call the function. The method can have a conventional name such as **apply** and not provide any "syntactic magic" beyond that. C++ allows overloading the "parenthesis operator" so that function-object applications look just like function calls. Scala does something halfway similar. In OCaml we can have class-based function objects, but functions in relevant senses already *are* objects.

Pointers to functions remain important in embedded-systems and related applications (especially C/C++ based ones), but for the motives of this course, they too are best folded into OOP. Likewise, overloading (sections 9.9 and 9.11) and generics (9.10) can be "delegated" to OOP coverage. To which we now turn...

From ADTs to OOP (Sebesta chapter 11 and parts of 12, but with more history),

Recall that two of the main advances in programming language design in the 1960s were Procedure Abstraction and Data Abstraction. Let us speak of “level”—not of a “computation,” but of a program—

as the main units the programmer thinks about while writing it. In assembly code, the “level” was individual assembly instructions; in the old FORTRAN, it was individual statements. COBOL and ALGOL-60 were the first languages to offer both user-defined (record) types and user-defined procedures.

People wrote “real software” at the procedure level well into the 1980s. Some still do. Classic Pascal has no other levels—everything is in one main program.

Some major limitations of the procedure level:

1. They expose the source code of the implementation.
2. They may have “side effects,” not part of their prescribed behavior, that other parts of the program may come to rely on. This makes it difficult to change the implementation.
3. (Taken together, these say that procedures do not provide information hiding.)
4. Procedures cannot “of themselves” be compiled separately. Pascal leaves you with one big program to recompile on every little change.
5. As programs get large, either you get zillions of little procedures—and then it’s hard to keep tabs on how control flows among them—or you get individual procedures with zillions of lines, defeating their purpose to begin with. Nesting of procedures (as in Pascal) helps some, but not much (and is impossible in C anyway).
6. Procedures do not relate directly to data—rather, data is subservient to procedures in the form of parameters.

This last may be the most important point. The root cause of programming difficulties in these languages is that they did not provide any good means of associating operations and (user-defined) data types.

The Answer to all of these problems was first the concept of a **Module**. This answer was realized already in the 1960s, but Niklaus Wirth (in particular) passed it over to keep ISO Standard Pascal simple. C header (`.h`) files do not really “encapsulate” data and operations—because they don’t provide a common (prefix) name or a common access pattern for their members.   It was left to the design of Ada and Modula-2 in the late 1970’s (and nonstandard extensions to Pascal...) to bring modules to a wide audience.

Classes in OOP languages perform the essential functions of modules---and others besides, Whereas, modules in languages like Ada and Modula-2 do not support inheritance or dynamic binding, so these languages are (well, were) called **object-based**, not object-*oriented*.

A module mainly implements an abstract data type in the manner of a *mathematical structure*. A famous reaction to the proliferation of mathematical structures in the late 1800s was Leopold Kronecker's statement, "God created the integers; everything else is *Menschenwerk*." An example of such "handiwork" is the definition of a **field** as a structure $\mathfrak{F} = (S, 0, 1, +, *, /)$ where S is a set called the *domain* of the field, S has at least two members abstractly called 0 and 1, and the field has operations called $+$, $*$, and $/$ that satisfy certain laws. The revelation was that mathematics is not just about numerical *"stuff"* but also the *operations* and *laws* that go with the stuff. The concepts of **members** and **methods** capture two-thirds of that triad, but IMPHO the representation of logical laws has lagged. (We will at the end view the Prolog language as an attempt to remedy this lack that fell short of its 1980s "5th Generation" hype.)

Modules

A module is a collection of declarations, typically of procedures and one or more types, all wrapped inside a "namespace" [this is vital, too!]. The main purposes are:

1. To associate operations to the type(s) explicitly.
2. To partition the program text into high-level "chunks."
3. To control which features of this chunk are visible outside the module—and which are not, so that they may be changed.
4. To provide a controlled way of initializing data types.
5. To provide a localized way of enforcing and verifying that vital data invariants hold during the execution of the program.
6. To provide separate compilation.
7. To hide the source code of the implementation as a "trade secret" while giving enough information for compilers to link into the module—this is done by having separate "specification/definition/header" and "implementation" files. [Implementation hiding refers to both visibility and this.] [However, the recent Open-Source movement is downplaying the importance of this.]
8. To wrap related code inside a namespace, so that it cannot cause name conflicts with other code. (This becomes a real issue as projects pass 100,000 lines. ANSI C++ introduced a feature called namespace for organizing variables that would have been "global" otherwise.)

We can once again view Ada in the rear-view mirror as the most prominently developed evocation of the procedural-language approach to modules and some germinations of classes. Here is an Ada `package`:

```
--File "customers.ads"---Ada spec file.
with [other packages]; use [stuff];

package Customers is    --public part
  type Customer is private;
  type Clientele is limited private;
  --so ":=" and "=" unusable for Clienteles.

  procedure Create(c: out Customer;
                         name, address: in MyString;
    [other information]);
  procedure addCustomer (cbase: in out Clientele;
                                    newC: Customer);
  procedure billCustomer(c: in out Customer);
  procedure deleteCustomer (cbase: in out Clientele;
                                       oldC: Customer);
  ...
private --Type info here, in the spec file, is exposed to readers.
          -- In Ada95 this area works like "protected" in C++.
  type Customer is record
    [fields for customers]
  end record;
  type Clientele is array(integer range <>) of Customer;
end Customers;
```

Then one must have a separate “package body” file named “`customers.adb`” that defines all the members, much as a .c or .cc file does in C or C++.

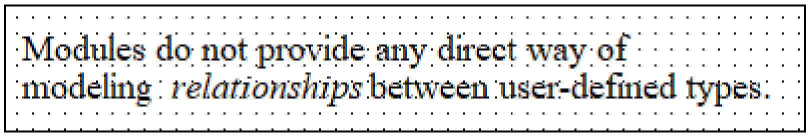
Modula-2 uses files whose first lines are `Definition Module` and `Implementation Module`. The language was quite popular for courses through the early 1990s and was adopted for real software projects by DEC, but even though modules fixed so many problems, applications programmers began to realize:

What Modules Don’t Do

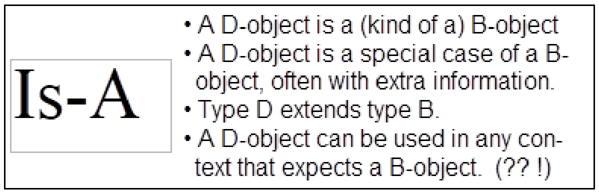
First, lets note some minor “clunks” of modules:

- A minor irritant is that one person’s `create` is another’s `make` is another’s `initialize` is another’s `init`... and so on. But this points to a potentially more troubling fact:
- Objects of an exported type have to be declared in statements separate from the procedure call that init-ializes them. In the interim, the objects have to be considered invalid. Yes, of course, one should call the proper initializer right after the declaration, but Pascal, Modula-2, Ada, and C all require declarations to be set apart from “body” code, often far enough away that it’s easy to forget to initialize one and go astray. Even aside from this, what about declaring an array of such objects? Stopping to initialize each one in separate calls for each one might be scads of unwanted time overhead.
- And why should one need a separate procedure call to initialize a data object anyway? !
-

But the most important lack is the following:


Modules do not provide any direct way of modeling *relationships* between user-defined types.

That's not quite true: modules can model the Has-A and Uses relationships—one can import types from another module, and for Has-A, one can define record fields of imported types in your own module's record types. But what's missing is the Big One:


Is-A

- A D-object is a (kind of a) B-object
- A D-object is a special case of a B-object, often with extra information.
- Type D extends type B.
- A D-object can be used in any context that expects a B-object. (?? !)

This relationship comes often in the "real-world" environment that we are trying to model. For instance, a Customer database will typically have many different categories of customers. The basic features common to all customers (name, balance, etc.) can be written into a base type Customer, and variations built on that. For instance, a `CorporateCustomer` may have extra fields such as `invoiceAccountNo` and `bondRating`. Let's look at several ways to model this relationship.

[Ada variant records used to be covered in earlier editions of Sebesta---even may hardcopy 9th edition---but were replaced by F# in the 12th edition's section 6.10. And through the 9th edition at least, Sebesta had a section on Ada95 in the OOP chapter 12. I am keeping the following material because variant records resemble OCaml datatypes and also relate to the implementation of OOP via object-type tags in all the major languages. Note how the main running example gets repeated in C++ and then Java.]

Variant records (Pascal, Modula-2, C, **Ada–shown**):

```
package body Customers is
  type CustTags is (Base,Corporate,...);    -- an Ada enumeration type
  type Customer(kind: CustTags) is record
    name: MyString;         -- user-def. strings
    balance: DollarAmt;     --also user-defined
    case kind is
      when Base => null;
      when Corporate =>
        invoiceAccountNo: integer;   --(?)
        bondRating: MoodyBlues;    --(!)
        ...
      when [etc.]
    end case;
  end record;

  procedure printAccountInfo(c: Customer) is
  begin
    case c.kind is
      when Base => [print out base info]
      when Corporate => [generate a corporate report]
      when [etc.]
    end case
  end printAccount;

  procedure billCustomer(c: in out Customer;
                         x: DollarAmt) is
  begin
    c.balance := c.balance + x;    --needs no tag check
  end billCustomer;
  ...
end Customers;
```

Type Extension (Modula-3, Oberon, **Ada95–shown**)

```
package body Customers is
  type CustTags is (Base,Corporate,...);
  type Customer is tagged record
    name: MyString; --user-def. strings
    balance: DollarAmt;
    ...
  end record;

  type CorporateCustomer is new Customer
  with record
    invoiceAccountNo: integer;   --(?)
    bondRating: MoodyBlues;    --(!)
  end record;

  procedure printAccountInfo(c: Customer) is
  begin
    [print out base info]
  end printAccount;
  procedure printAccountInfo   -- overrides
    (cc: CorporateCustomer) is
  begin
    [generate a corporate report]
  end printAccount;

  procedure billCustomer(c: in out Customer;
                         x: DollarAmt) is
  begin  -- simply inherited by CorporateCustomer, no override
    c.balance := c.balance + x;
  end billCustomer;
  ...
end Customers;
```

In the Ada95 code, the "tag" becomes implicit and is directly supported by the language. Notice that the code "looks" less clunky. Most important, it is possible to add more kinds of customers without having to go back and edit "case blocks" in all the pre-existing data declarations and procedures.
The rules of Overloading and Type Derivation (via "`new`"—recall from Ch. 6 notes on "Types") that were already present in Ada83 now take over and ensure that things work the way we're accustomed to in OOP:

- The type extension adds new fields, just like in C++. One can also do type extension without adding new fields by writing "`...with null record;`"
- An object `cc` of the derived type CC can be used in a call such as `billCustomer(cc,x);` without an explicit conversion. In OOP-speak, CC inherits this operation from C.
- However, the rules of `new` prevent a base object c from being used in a procedure such as `getBondRating(cc: CC)`—a call with argument c would be caught as a compile-time error.

- Most relevant for us, however: All OOP languages basically work by the same kind of "tag"-lookup and case-analysis that we saw with Ada variant records! The difference is not so much in the underlying mechanism but in the convenience to the programmer of extending existing types and putting the new methods in separate files. The Ada95 scheme's main drawback is that it still separates types and methods.

Type-With-Module Extension: (Simula67, Smalltalk, **C++ (shown)**, Java, etc.)

```
class Customer {
   MyString _name;           //default area is private
   DollarAmt _balance;
public:
   Customer(MyString name, DollarAmt balance)
      : _name(name), _balance(balance) {}
   Customer(MyString name)    //since one parameter, could say "explicit"
      : _name(name), _balance(0) {}
   void bill(DollarAmt x) {   // "mutator"
      _balance += x;
   }
   virtual void printAccountInfo() const {
      [print base info]
   }
};  //end Customer.  Semicolon mandatory

class CorporateCustomer: public Customer {
   int invoiceAccountNo;     //(int ?)
   MoodyBlues bondRating;
public:
   CorporateCustomer(MyString name, Etc.)
      : Customer(name), invoiceAccountNo(Etc.) { }
   virtual void printAccountInfo() const {    //overrides
      [print a corporate report]
   }
};  //end CorporateCustomer
```

Note that the "explicit first parameter" c or cc of the Ada methods becomes the "implicit first parameter before the dot" in C++. Ada `printAccountInfo(c)` becomes `c.printAccountInfo()` in C++, and Ada `billCustomer(c,x)` becomes `c.bill(x)`.
Here the methods are inside the record structure Note also how constructors in C++ answer the minor complaints about initialization in Modula-2 and Ada under "What Modules Don't Do" above.

The calls `c.printAccountInfo()` and `cc.printAccountInfo()` are statically bound. To get "virtual" behavior, one can use the pointer type of the base class to stand for "the class-wide type:"

```
   Customer* c3 = &c;       //points to a Customer
   c3->printAccountInfo();  //prints base info
   c3 = &cc;  //now points to a CC object
   c3->printAccountInfo();  //corporate report.
```

Now the method binding is done at activation time, i.e., "dynamically."

In Java, every object variable behaves like a C++ pointer, and every member function is "virtual" by default. Here is the same example sketched in Java:

```
public class Customer {
   String name;         //default area is protected
   DollarAmt balance;   // using _ is legal but frowned on; you can
                        // write this.balance to emphasize a field name
   public Customer(String gname, DollarAmt gbalance) {
      name = gname;
      balance = gbalance;
   }
   public Customer(String gname) {
      name = gname; balance = 0;
   }
   public void bill(DollarAmt x) {  //"mutator"
      balance := balance + x;
   }
   public void printAccountInfo(){  //"accessor"
      [print base info]
   }
}  //end Customer.  No semicolon!

public class CorporateCustomer extends Customer {
   int invoiceAccountNo;  //(int ?)
   MoodyBlues bondRating;

   public CorporateCustomer(String name, Etc.) {
      super(name);
      invoiceAccountNo = Etc.;
   }
   public void printAccountInfo(){ //overrides: every method is "virtual"
     [print a corporate report]
   }
}  //end CorporateCustomer
```

Note the absence of `const`, and `super(name)` in place of the named constructor call in the corresponding C++ code. Also note that `public` (and `private` and `protected`) is applied to individual members, not to a "region" of the class. Now let's look at some calls in Java:

```
   Customer c3 = c;       //does not copy c !
   c3.printAccountInfo();  //prints base info
   c3 = cc;             //does not copy cc
   c3.printAccountInfo();  //corporate report.
```

These calls are dynamically bound, at activation time. If you want to use a method like `getBondRating()` that is defined only in the derived class, then you need a "promotion cast" such as

```
((CorporateCustomer)c3).getBondRating();
```

For this to work, `c3` had better be holding a `CC` object at the time this line is executed—otherwise Java will throw a `RuntimeException`. One can pre-test for this via the instanceof keyword, as in

```
if (c3 instanceof CorporateCustomer) {
   ((CorporateCustomer)c3).getBondRating();
} else {
   [do something else...];
}
```

Cast notation appears much more frequently in Java than in C++. (I originally wrote this in 1997--KWR.) It may look ugly, but...it took C++ a long time to standardize an equivalent mechanism that in this case would involve a call to test a `Customer*` pointer by

```
CorporateCustomer* q = dynamic_cast<CorporateCustomer>(&c3));
if (q) {
   q->getBondRating();
} else {
   [do something else]
}
```

If the cast fails, then `q` is set equal to `NULL` (now called `null` but still just `0` under the hood), which fails the "`if`" test. If the cast succeeds, then the call---which the compiler already greenlights by `q` having the derived class type---will not cause a segmentation fault. This is all more complicated and "heavy" looking than in Java.

To declare and initialize a new `Customer` object, Java uses the syntax

```
Customer c4 = new Customer("Jones");
```

The repetition of "`Customer`" may sound quaint, but it actually comes straightforwardly from the conjunction of pointer and constructor syntax in the analogous C++ line

```
Customer* c4 = new Customer("Jones");
```

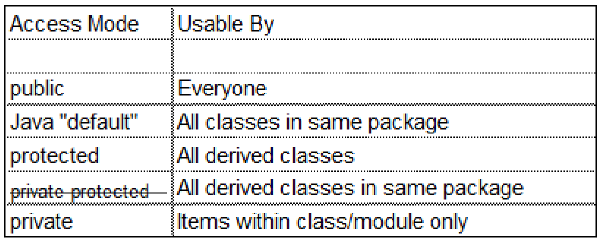
The upshot is that although Java does away with the `&`, `*`, and `->` operators of C++, and thus prevents you from manipulating pointer addresses explicitly in the code, implicitly "under the hood," everything (except primitive types) behaves like pointers in C++. Thus the sequence `c4 = c3;`

`c4.bill(99.50);` also bills `c3`! Some other comparisons to C++:

Every “library-level” (a.k.a. “file-scope-level”) item in Java must be a class (or an interface, which is a special kind of class with no data fields or constructors). In particular:

1. There are no “global variables” or “global functions,”
2. Even `main` resides inside a class—actually, you can have a `main` inside every class! (This is useful for supplying a “test driver” for classes.)
3. there is (still currently) no separation of method header and body as in Ada and C++.

While on that score, Java does not use “`::`” to connect the classname to items in the class; the Ada “`.`” notation is used throughout.   Java “packages” remove most of the need for “`friend`” in C++, and refine C++’s access/visibility rules (one refinement got axed in 1995).



Access Mode	Usable By
public	Everyone
Java "default"	All classes in same package
protected	All derived classes
~~private protected~~	All derived classes in same package
private	Items within class/module only

Sebesta's Takes (12.1--12.3)

The above examples have illustrated the evolution of what Sebesta calls three "essential characteristics" of OOP:

1. Implementing ADTs
2. Inheritance
3. Dynamic binding---in particular, **dynamic dispatch** of virtual method calls.

A convenient feature of the last is that the same mechanism enables both **static typing** and **dynamic binding** of method calls. Sebesta gives the example of an object hierarchy based on a class `Building` that can have many derived classes ranging to way-out-there ones like `FrenchGothicCathedral`. Each can have a `drawBlueprints()` method. We can't give any meaningful code for the blueprints of a vanilla `Building`. But by putting an **abstract** method of that name in that class, we:

- ensure that every derived class either defines or inherits a concrete implementation of the method, and

- force every call from an actual concrete building object to follow a **super** chain until it finds an implementation of the method---the call cannot be statically bound to the base class version because it has no code body.

(Don't confuse the superclass chain with either the static or dynamic links in a subprogram call chain.) Sebesta also helpfully distinguishes the name of the invoked method as the **message**, reserving **method** to mean the actual code that gets called. Thus he says that *messages are bound dynamically to methods.*

Sebesta's own list of **Design Issues For OOP Languages**:

1. Should the language exclusively use objects?
   (a) C++ and Python and OCaml: no.
   (b) Scala yes---even the basic types are objects.
   (c) Java and Javascript: everything except the bedrock basic types is an object, and we even provide Object-based workalikes of those.
2. Are Subclasses Subtypes? My "Square-Is-A-Rectangle?" example from the first lecture is an instance of this, but Sebesta gives an even narrower formulation. The text's mantra is:
   - A **subtype** is supposed to inherit behavior as well as interface;
   - A **subclass** inherits *implementations*, primarily for code re-use.
3. Should multiple inheritance---*of implementation as well as interface*---be allowed?
   - C++ allows unrestricted multiple inheritance of classes. To resolve the "**Diamond Problem**", the programmer must designate which base class is "virtual".
   - Java allows multiple inheritance only of **interfaces**. Each class must have only one **super**.
   - Scala allows multiple inheritance only of **traits**; as in Java, a class can have only one **super**. But Scala's **traits** are richer than Java's interfaces and can hold data and concrete implementations, leading to the Diamond Problem.
   - OCaml allows **mixin inheritance**, meaning (basically) that if A is your **super**, you can also inherit from a class B what B bases only on its own inheritance from A.
4. Are objects allocated only on the heap, or can they be stack-dynamic? Morphing the question a little: does the language provide **value objects**?
   - Java: no.
   - C++: yes---indeed, keeping full support for value objects is a main motivation.
   - Scala: yes ("case objects" and "case classes"), but they can't inherit from each other.
   - OCaml: Pure-value objects by default unless one declares a field `val mutable`.
5. Must all bindings of method code to messages use the same dynamic lookup mechanism, or can some of the bindings be static? Here **static** means that a message `x.foo(...)` is bound to the same code whether `x` currently holds a base class or derived class object.
   - C++: to get dynamic dispatch, you must *both* declare the `foo` method to be `virtual` and must invoke it either as `px->foo` through a pointer or must create the variable `x` as an **express reference** to an existing object.
   - Java: dynamic is only option---though the ability to access an ancestor-class implementation of a method via **super** prvides a little effect of static.

- Scala (esp. 3.0): We have syntax for everything now...
- OCaml: customizable like Scala.

6. Can classes be nested---and is there any privileged point to doing so? If so, how full-fledged and/or independent are the objects of those classes?
7. How is initialization done? (This is a huger topic than appears from Sebesta.)
8. Can new classes be defined and their objects created during the run? This is part of what **section 10.6** on **reflection** is getting at.

The Diamond Problem

To model personnel at UB, we can start with a base class `Person`. It can have fields not only for name and address etc. but also the basic UBID is the same for everyone. From there we can define the common idea of an `account`. To make this more concrete than typical textbook examples are, let's also imagine that the system adopts an architecture in which events affecting the account are logged rather than passed as explicit parameters. When requested, a parameterless method `updateAccount()` consults the log, and the account is updated accordingly. Then it is natural for the system to branch this way:

- `class Student extends Person` { ... `updateAccount()` for tuition coming in ... }
- `class Employee extends Person` { ... `updateAccount()` for salary going out ... }

But increasingly nowadays there is a category that blends them:

- `class UTA extends Student,Employee` { ... which `updateAccount()` to re-use? ... }

There are various levels of issues and workarounds:

- At lowest level, the `account` field can be inherited by both `Student` and `Employee`. Then the rule of combining visible fields from inherited classes will induce the compiler to allocate two copies of that field in the record structure for class `UTA`.
  - Workaround 1: declare the `account` field `private` in `Person`. Still leaves other problems. (Available in C++ and Java but has other issues.)
  - Workaround 2: treat class extension as field inclusion. Whichever of `Student` and `Employee` is included last gets to define the `account` field in the class `UTA`. (Used by Scala with `traits` and OCaml with anything.)
- Next issue: the `UTA` constructor will want to invoke constructors in both `Student` and `Employee`. Each of those in turn wants to invoke a `Person` constructor. So it looks like the `Person` part will get constructed twice.
  - Workaround 1: In C++ one can declare one or both Student and Employee to derive from `: public` **`virtual`** `Person`. One of several effects of this is that the `UTA` constructor can call the "grandparent constructor" `Person` directly---once.
- General issue: How do we consistently re-use both the `Student.updateAccount()` and

`Employee.updateAccount()` methods?

Even the last issue is largely helped if we have a situation where it makes sense for (say) `Employee` to inherit from `Student`. This would largely resolve issues of `UTA` inheriting from both. Or, perhaps there is a way to inherit code but not data from `Employee`. In various fashions, Employee would then be a **mixin**, defining **mixin inheritance**. Mixin inheritance is generally considered software-engineering sound.

Other Design Issues

1. Is there a notion of **object** that is *separate* from the notion of **class**? Can one have objects without classes?
2. Is there support for **singleton objects**, like `Clientele` in the "`Customer`" example?
3. Are constructor arguments distinguished as class arguments from other fields? (Scala and OCaml: yes)
4. Are **accessor** features of classes distinguished in other ways? (get/set in Scala, C#)
5. Is class extension treated as *specialization* or as *generalization*? (Continuing discussion from before spring break, we will use thuis as one "angle" on OCaml...)

OCaml Classes and Objects

Main source: https://ocaml.org/docs/objects
Next recommended source: https://dev.realworldocaml.org/classes.html