Object Orientation

• Progression of data abstraction/encapsulation so far:
  ◊ Global variables — always visible, all the time
  ◊ Local variables — limited lifetime and visibility
  ◊ Nested scopes — local subroutines and variables
  ◊ Modules — first as manager, then as type

• Take module-as-type, then add:
  ◊ Inheritance — incremental refinement or extension of existing abstractions
  ◊ Dynamic method binding — new behavior in a context that expects an earlier version
  ◊ Classes and objects — families of related abstractions, and the ability to create instances belonging to that family

• ...and you get an object-oriented programming paradigm

Brief History

• The three key object-oriented programming concepts of encapsulation, inheritance, and dynamic method binding first appeared in Simula, mid-1960s

• Simula’s data hiding was improved upon with Clu, Modula, Euclid, etc. in the 1970s

• Inheritance and dynamic method binding were refined into a message-based model with Smalltalk, also 1970s

• Object-orientation stayed relatively dormant until the 1990s, with C++, Eiffel, Objective-C, Ada 95, and ultimately Java, C#, Python, Ruby
Programming Elements

• Three key benefits to data abstraction:
  ◊ Reduce conceptual load — less things to think about at any given time
  ◊ Provides fault containment — limits the context in which code may run or be applicable
  ◊ Independence among components — ability to modify internals without affecting externals

• The “as-is” rule of software reuse — if you can’t use code as-is, then you can’t [won’t] reuse it

• In many ways, object-oriented programming seeks to facilitate code reuse by allowing extension and/or refinement of existing abstractions, without having to dig into those abstractions

Terminology Check

• Object-oriented languages uniformly use classes for their abstractions, with objects representing individual instances of a class

• Within a class, terminology sometimes diverges:

<table>
<thead>
<tr>
<th>data member</th>
<th>field, instance variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>subroutine member</td>
<td>method, member function</td>
</tr>
<tr>
<td>current object</td>
<td>self, this, current</td>
</tr>
<tr>
<td>class derived from</td>
<td>derived class, child class, subclass</td>
</tr>
<tr>
<td>class from which</td>
<td>base class, parent class, superclass</td>
</tr>
<tr>
<td>“top-level” class</td>
<td>Object, ANY</td>
</tr>
</tbody>
</table>
More Terms

- **Constructor** — programmer-specified initialization code
- **Destructor** — programmer-specified clean-up, deallocation code; not available in all languages
- **Public** and **private** designations — object-oriented equivalent of module export
- **Overriding or redefining** — new code in a subclass for a method that also exists in the superclass; languages provide a mechanism for accessing elements of the superclass (*super, base, superclass_name::*)

Encapsulation

- Originated with module-based languages in the 1970s
- Euclid introduced *closed scopes* — explicitly stating the names that other code may see — and *opaque types* — types that are known in external code by name only
- The only thing you could do with instances of opaque types was pass them into the subroutines defined by the modules: `push(stack, value)` or `value = pop(stack)`
- Opaque types then evolved into today’s “objects” — `stack.push(value), value = stack.pop()`, where `stack` is implicitly passed by reference as a *this* variable
Encapsulation When Inheritance is Involved

Additional rules needed with inheritance/subclasses

• What can a subclass see from its superclass?
• Can a subclass modify the visibility of members to something different from the superclass? (C++, yes; Java & C#, no)
• New level of visibility: protected — visibility only to some well-defined subset, such as derived classes (or packages, in Java’s case)
• What is the default visibility when no keyword is given?

Initialization

• As mentioned previously, object-oriented languages introduce a special type of subroutine, called a constructor, that takes care of initializing new objects
• Code within a constructor is responsible for populating a new object’s members, not for allocating space for the object itself
• More than one constructor is typically supported, to accommodate different valid initialization arguments
• Some languages also provide a destructor, for finalizing an object at the end of its lifetime
Choosing Among Constructors

- C++, Java, C#: constructors have the same name as their classes, and are distinguished by their signatures (i.e., number and types of formal parameters)

- Different constructor names Smalltalk and Eiffel: in Smalltalk, a constructor looks like a static Java method

<table>
<thead>
<tr>
<th></th>
<th>Java Constructor</th>
<th>Java Static Method</th>
<th>Smalltalk Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Date for today”</td>
<td>public Date()</td>
<td>public static Date getToday()</td>
<td>Date today</td>
</tr>
<tr>
<td>“Date with a given year, month, and day”</td>
<td>public Date(int y, int m, int d)</td>
<td>public static Date getDate(int y, int m, int d)</td>
<td>Date with Year: month: day</td>
</tr>
<tr>
<td>“Date for yesterday”</td>
<td>public Date(int delta) (hmm…)</td>
<td>public static Date getYesterday()</td>
<td>Date yesterday</td>
</tr>
</tbody>
</table>

Reference vs. Value Variables

- Reference model enforces explicit creation of objects, and therefore explicit calling of constructors

- Value model raises the issue of implicit constructors:

  ValueModelClass vmc; // With value model, vmc should already be an object.

- C++ approach: direct declaration calls the zero-argument constructor implicitly; additional syntax provided for calling other constructors

- General rule of thumb: reference model typically works better for objects, but require heap allocation and additional indirection with every access
Constructor Execution Order

- Issue arises with derived classes: typically, a derived class constructor implicitly calls its base class constructor(s) first, from the generic to specific

- But, since each class can have multiple constructors, which superclass constructor is invoked?

  - C++: can specify base class constructor call and/or member variable initial values
  - Java: super(args) call as first line of constructor invokes the given superclass constructor

Garbage Collection

- Object orientation doesn’t immediately imply automatic garbage collection — case-in-point, C++

- Corresponds to need for explicit destructor — when an object’s lifetime ends, the destructor allows it to deallocate storage that it was using

- Automatic garbage collection eliminates (or at least severely reduces) the need for destructors, since a reclaimed object’s children will eventually be hit by the automated garbage collector anyway
Dynamic Method Binding

• If a class $D$ is derived from a class $C$, then $D$ has all of the members of $C$, and therefore $D$ should be usable anywhere that $C$ is usable:

```java
C object1, object2;
object1 = new C();
object2 = new D();
object2.anyMethodThatCHas();
object1.methodThatExpectsArgumentOfTypeC(object2);
// ...etc.
```

• The ability to use a subclass in a context that expects its superclass is called **subtype polymorphism**

• But what if $D$ overrides or redefines some method $m$ in $C$ — when $m$ is called, which version is used?
  
  ◦ If we use the version corresponding to the variable’s declared type, then we have **static method binding**
  
  ◦ If we use the version corresponding to the variable’s actual object, then we have **dynamic method binding**

• Dynamic method binding is central to object-oriented programming — otherwise, why bother with inheritance and method overrides?
  
  ◦ Particularly important if a subclass’s version of a method has code that specifically maintains internals
  
  ◦ Imposes run-time overhead; one early reason for complaints that object-orientation was slow
Accommodating Both Binding Methods (or not)

- Smalltalk, Objective-C, Modula-3, Python, Ruby: dynamic method binding all the time
- Java, Eiffel: dynamic method binding by default, can be optimized by disallowing overrides (\texttt{final} or \texttt{frozen}, respectively) — net result is elimination or reduction of runtime overhead
- Simula, C++, C#: static method binding by default; can change to dynamic by designating a method as \texttt{virtual}

Abstract Methods & Classes

- An \textit{abstract method} is one for which no body is given
  - Java, C#: use \texttt{abstract} keyword in method declaration
  - C++: “assign” a subroutine to zero; C++-specific terminology is \textit{pure virtual} method
- An \textit{abstract class} if it has at least one abstract method
- A \textit{concrete class} is a subclass of an abstract class that provides a body for every abstract method it inherits
- Abstract methods and classes help define generic behaviors that do not have a specific implementation at the level of the superclass (e.g., \texttt{shape.draw()})
Dynamic Method Binding Implementation

- Static method binding means that the specific subroutine code to be invoked is known at compile time, and so can be referenced directly; not so for dynamic method binding.

- Common implementation approach: accompany objects with a virtual method table that lists the addresses of its methods; objects of a derived class overwrite the addresses of methods that are overridden by that class.

- Dynamic method call thus uses an additional lookup.

Dynamic Method Binding and Variable Types

- Typed variables allow some static verification of code: a given variable $v$ will be enforced to “at least” have class with which it was declared.

  ◦ Class casting is still typically allowed, but requires dynamic checks.
  ◦ For backward compatibility, C++ accommodates unchecked casts…caveat coder.

- Untyped variables (Smalltalk, CLOS, Objective-C) make all checks dynamic: a method call requires a runtime lookup on the object currently assigned to a variable.

  ◦ Smalltalk uses a messaging model for subroutine invocations: method calls “send a message” to the object, and non-existed methods result in a “message not understood” error.
The Fragile Base Class Problem

- Runtime subroutine/method lookup avoids the fragile base class problem — what if you are running code that expects a different version of some class than is available on the system?

  ◊ It can happen, particularly in Java: multiple versions of Java, portability of class code

- Runtime lookups trigger better error reporting, akin to “method not understood” in messaging models

- Static references may access invalid memory locations

Generics and Closures in Object-Oriented Languages

- Note that dynamic method binding does not eliminate the usefulness of generics in a language

- Generics implement parametric polymorphism — it defines code that can be used in common among unrelated types

- Dynamic method binding offers an alternative to closures — define a class with the desired subroutine, and call that subroutine off its instances

  ◊ Class approach can embed information beyond a subroutine’s formal parameters

  ◊ But, tends to be more verbose (e.g., Java’s Runnable)
Multiple Inheritance

• Conceptually simple: allow a derived class to inherit from more than one base class; the derived class acquires the union of the members of the base classes

• But the devil is in the details…
  ◦ What if two superclasses have a method with the same name?
  ◦ What if two superclasses in turn have a common superclass — does the “grandchild” class have two copies of the shared members?

• Multiple inheritance involving a common “grandparent” class is called repeated inheritance
  ◦ Repeated inheritance resulting in multiple copies of grandparent members in the “grandchild” is called replicated inheritance — default in C++
  ◦ Repeated inheritance with one copy of grandparent members is called shared inheritance — default in Eiffel

• Simula, Smalltalk, Objective-C, Modula-3, Ada 95, Oberon: single inheritance only

• Java, C#, Ruby: define an interface class that declares methods only; “multiple inheritance” occurs by allowing only one class parent but any number of interfaces (a “best-of-both-worlds” solution)

• C++, CLOS, Python, Eiffel: multiple inheritance, with specific variations
“Object-Orientation” in Perl

Perl gained object-oriented characteristics in version 5

• “Classes” are Perl packages
• “Methods” are subroutines within those packages
• Three new constructs:
  1. A “method operator” (\rightarrow) implicitly passes the package as an argument
  2. A reserved @ISA array lists the “superclasses” of a package (i.e., other packages)
  3. A bless operator “binds” a package (class) to a variable, giving that variable semantics similar to an instance in terms of the package’s (class’s) subroutines (methods)

• Is this a case of “forcing a feature?” Discuss…

Objects in JavaScript

• JavaScript objects are \textit{prototype-based}, and not \textit{class-based} — there is one Object type, and as you know, Objects may have any number/name/type of properties
• Preceding a function invocation with \texttt{new} creates a new object and passes it into the function as this; the function can then assign this’s properties and return it
• Methods use the constructor’s special \texttt{prototype} property — the constructor passes \texttt{prototype} to \texttt{this}
• JavaScript’s approach makes inheritance as we know it somewhat unwieldy — it’s actually a different paradigm