SIENAcollege
Computer Science

Computer Science 340
Programming Languages
Siena College
Fall 2019

Topic Notes: Object-Oriented Programming Support

Many programming languages support the *object-oriented programming (OOP)* paradigm. We will review/consider some of the important issues both from a programmer's perspective and that of programming language design.

The text chapter describes OOP support in several languages. Again, we will focus on a subset of the languages that cover the important OOP features.

Inheritance

The first major feature needed for OOP is support for abstract data types (ADTs), which we just studied.

A second is *inheritance* – the ability to define new classes in terms of existing ones.

A major motivation for ADTs is to support code reuse. In some circumstances, this seems relatively easy. Java's `ArrayList` class serves many needs as-is, but it is often that case that an ADT provides only some of the needed functionality for a given task. We would like to reuse what we can but allow this extensibility. Inheritance provides the mechanism to do the latter.

First, some terminology, much of which you already know:

- *classes* – the definitions for ADTs
- *objects* – instances of classes
- *derived class* or *subclass* – a class that inherits from another, which is its *parent class* or *superclass* – and we say the subclass *extends* the parent class
- data is in *instance variables* (those which are created uniquely for each object instance) and *class variables* (those which exist once per class, regardless of whether or how many instances exist)
- *methods* of a class provide the operations on objects of that class (*instance methods*) or on the class itself (*class methods*)
- method calls are sometimes referred to as sending *messages* to an object and require both method name and object on which to call the method while those to a class require method name and class name
- entities of a class (*i.e.*, methods, instance variables and class variables) might be available to or hidden from subclasses, which is separate from being visible or hidden from users of the class

- a subclass may *override* a method from its parent
- a subclass can add new entities to those provided by its parent
- with *single inheritance*, a subclass inherits directly from a single parent class
- with *multiple inheritance*, a subclass can inherit directly from multiple parent classes – much more complex!

Polymorphism

The use of inheritance also usually requires support for *polymorphism*.

Suppose a parent class `X` is extended by subclass `Y`, and a method `a` is provided by `X` and overridden by a method `a` provided by `Y`. Then a variable `b` of type `X` could refer to an instance of `X` or an instance of `Y`. Polymorphism requires that a call to `a` through `b` (*e.g.*, `b.a()`) would call the method defined in `X` if `b` refers to an object of type `X`, and to the method defined in `Y` if `b` refers to an object of type `Y`. This is also called *dynamic binding*.

Confused by the above paragraph?

See Example:
`/home/cs340/examples/dynamicbinding`

A related concept is that of an *abstract method* – one that a class defines only by protocol (*i.e.*, the method header) but without providing a complete definition. A class which includes an abstract method is called an `abstract class`, and cannot be instantiated. Only subclasses that define an actual implementation for its abstract methods can be instantiated. Note: C++ uses the term *virtual methods* here.

OOP Design Issues

Several design issues arise in language support for OOP.

- Is everything an object?

  A purely OOP language would require *exclusivity of objects* – that everything is an object. Many languages (such as Java) include support for primitive types separate from objects, and provide wrapper classes for situations where primitive types need to be treated as objects.

  Allowing objects exclusively leads to a more elegant language at the likely expense of efficiency.

- Is a subclass a subtype?

  That is, if we construct an instance of a subclass, is it also an instance of its parent?

  Usually, the answer here is yes. The text discusses some fairly subtle issues about this.

- Is multiple inheritance supported?

  Java: no. C++: yes. The main disadvantage is in the complexity of implementation. How does the language deal with potential name collisions? That is, if a subclass inherits from two parents, each of which define an instance variable named `x` that is visible to the subclass.

  C++, for example, checks at compile time and issues an error when a conflict occurs.

  There is also the issue of *diamond inheritance*, as shown in Figure 12.3, where the parent class `X` defines a method `f`. `A` extends `X` and overrides `f`. `B` extends `X` and also overrides `f`. Then `X` extends both `A` and `B`. Which version of `f` would/should be called on an instance of `C`?

  Java chose to provide *interfaces* to provide some of the functionality that can be provided through multiple inheritance, while avoiding the language complexity that multiple inheritance would introduce.

  In Java 8, the idea of *default methods* was added to interfaces:

  On the web: Default Methods at `tutorialspoint.com` at `https://www.tutorialspoint.com/java8/java8_default_methods.htm`

  This example demonstrates how it handles possible collisions similar to the diamond inheritance problem described above:

  See Example:
  `/home/cs340/examples/defaultmethods`

  Aside: what problems in terms of naming arise when new features like this are added to a language and how did Java avoid name troubles?

- Allocation and deallocation of objects

  Some languages require objects to allocated from the heap, with only references on the stack.

  If stack dynamic allocation of objects is permitted, the problem of *object slicing* can arise:

  Consider a class `B` which extends a class `A` and adds data fields. It would be legal to declare instances on the stack:

  ```
  A a;
  B b;
  ```

  and later assign:

  ```
  a = b;
  ```

  Which should be legal, as a `B` is an `A`. But `b` requires more space than `a`, so at best only the part of `b` that uses `A`'s variables could be copied without exceeding the space needed.

- Static vs. Dynamic binding of methods

  We saw that polymorphism requires a dynamic binding, but that had some expense (space and time efficiency).

  So some languages allow a user to specify when binding should be dynamic or not.

  Dynamic binding sometimes used a mechanism called a *virtual method table* to locate the appropriate version of a method to be called by an object. This introduces a cost to method invocation.

- Initialization

  On construction of a new object of a subclass, how are parent members initialized? Default values? Call a default constructor of the parent implicitly? Call a parent constructor explicitly?

  Given this example, how does Java handle calling of constructors?

  See Example:
  `/home/cs340/examples/javaconstructors`

Language Support

The text describes support for OOP in several languages. The C++ section is well worth a read, and we will discuss several points from it in class. The other sections are also interesting.

1 A Java Case Study

We will consider Java's OOP support using a large package of data structures as a case study:

On the web: Java Structures at
`http://www.cs.williams.edu/~bailey/JavaStructures/Welcome.html`

Also, the source code is available in `https://github.com/SienaCSISDataStructuresJDT/structure5`.

Note in particular the extensive use of Java interfaces to define common functionality for various classes of structures. Then, implementations of methods common to more than one other structure that are "factored out" into abstract classes when possible.