**OBJECT-ORIENTED PROGRAMMING**

Object-oriented programming is a style of programming

Imperative Programming:

* Program = Data + Algorithms

Logic/Declerative Programming:

* Program = Axioms + Queries

Functional Programming:

* Program = Functions Functions

OO Programming:

* Program = Object.message(object)

Outline

Central concepts in object-oriented languages

* Dynamic lookup, encapsulation, subtyping, inheritance

Objects as activation records

* Simula: implementation as activation records with static scope

Pure dynamically-typed object-oriented languages

* Object implementation and run-time lookup
* Class-based languages (Smalltalk)
* Prototype-based languages (Self, JavaScript)

Statically-typed object-oriented languages

* C++ – using static typing to eliminate search
* C++ – problems with C++ multiple inheritance
* Java – using Interfaces to avoid multiple inheritance

Primary object-oriented language concepts

* dynamic lookup
  + When I send a message to an object, how do I know that message belongs to that object, how do I know it is gonna process that message?
* encapsulation
* inheritance
* subtyping

Program organization

* Work queue, geometry program, design patterns

Comparison

* Objects as closures?

Objects

Table

Description automatically generatedAn object consists of

* hidden data
  + instance variables, also called fields, data members, …
  + hidden functions also possible
* public operations
  + methods or member functions
  + can also have public variables in some languages

Object-oriented program:

* Send messages to objects

*What’s interesting about this?*

Universal encapsulation construct

* Data structure
* File system
* Database
* Window
* Integer

Metaphor usefully ambiguous

* sequential or concurrent computation
* distributed, synchronous or asynchronous communication

Object Orientation

Programming methodology

* organize concepts into objects and classes
* build extensible systems

Language concepts

* dynamic lookup
* encapsulation
* subtyping allows extensions of concepts
* inheritance allows reuse of implementation

Dynamic Lookup

In object-oriented programming

object 🡪 message (arguments)

code depends on object and message

In conventional programming,

operation(operands)

meaning of operation is always the same

Fundamental difference between abstract data types (alone) and objects

**Example**:

Text

Description automatically generated with medium confidence

IMPORTANT DISTINCTION: Overloading is resolved at compile time. Dynamic lookup is a run-time operation.

My dynamic lookup has to be able to resolve which add, which algorithm this is gonna run.

We are creating objects at runtime.

Some languages resolve “add(x, y)” in run-time. They do type inference.

Language Concepts

“dynamic lookup”

* different code for different objects
* integer “+” different from string “+”

encapsulation

subtyping

inheritance

Deciding things at runtime will require extra processing.

Some languages do mapping, lookup at compile time.

Encapsulation

Builder of a concept has detailed view

User of a concept has “abstract” view

Encapsulation separates these two views

* Implementation code: operate on representation
* Client code: operate by applying fixed set of operations provided by implementer of abstraction

Shape

Description automatically generated

Language Concepts (Again)

“Dynamic lookup”

* different code for different object
* integer “+” different from real “+”

Encapsulation

* Implementer of a concept has detailed view
* User has “abstract” view
* Encapsulation separates these two views

Subtyping

Inheritance

Subtyping and Inheritance

Interface

* The external view of an object, from user’s perspective

Subtyping

* Relation between interfaces
* Abstracted view of how concepts (objects) are related to each other
  + Can I use concept A instead of concept B as a messaging mechanism or as an object or vice versa

Implementation

* The internal representation of an object

Inheritance

* Relation between implementations
* Implementer view of how concepts (objects) are related to each other
  + Heavily interested in code reuse

Object Interfaces

Interface

* The messages understood by an object

Example: point

* x-coord : returns x-coordinate of a point
* y-coord : returns y-coordinate of a point
* move : method for changing location

The interface of an object is its type (values & operations on those values)

Subtyping

If interface A contains all of interface B, then A objects can also be used as B objects

Graphical user interface, text, application, chat or text message

Description automatically generated

Point is more generic.

Inheritance

Implementation mechanism

New objects may be defined by reusing implementations of other objects

**Example**:

A picture containing timeline

Description automatically generated

OO Program Structure

Group data and functions to achieve encapsulation

Class

* Defines behavior of all objects that are instances of the class

Subtyping

* Place similar data in related classes

Inheritance

* Avoid re-implementing functions that are already defined

**Example**: Geometry Library

Define general concept: shape

Implement two shapes: circle, rectangle

Functions on implemented shapes

* center, move, rotate, print

Anticipate additions to library

Shapes

Interface of every shape must include

* center, move, rotate, print

Different kinds of shapes are implemented differently

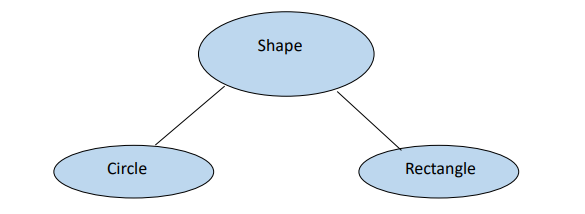
* Rectangle: four points, representing corners
* Circle: center point and radius

Subtype Hierarchy

General interface defined in the shape class

Implementations defined in circle, rectangle

Extend hierarchy with additional shapes



Code Placed in Classes

Table

Description automatically generated

Dynamic lookup

* circle 🡪 move(x,y) calls function c\_move

Conventional organization

* Place c\_move, r\_move in move function

***Example use: Processing Loop***

Graphical user interface, text, application

Description automatically generated

I can build this queue as queue of shapes.

When I perform action on each element, move function knows how to behave.

Outline

Central concepts in object-oriented languages

* Dynamic lookup, encapsulation, subtyping, inheritance

Objects as activation records

* Simula – implementation as activation records with static scope

Pure dynamically-typed object-oriented languages

* Object implementation and run-time lookup
* Class-based languages (Smalltalk)
* Prototype-based languages (Self, JavaScript)

Statically-typed object-oriented languages

* C++ – using static typing to eliminate search
* C++ – problems with C++ multiple inheritance
* Java – using Interfaces to avoid multiple inheritance

Simula: Objects as Activation Records

Simula 67: First object-oriented language

Designed for simulation

* Later recognized as general-purpose programming language

Extension of Algol 60

Standardized as Simula (no “67”) in 1977

Inspiration to many later designers

* Smalltalk
* C++
* ...

Brief history

Graphical user interface, text, application

Description automatically generated

Sample Simula 67 Code



Objects in Simula

Class

* A procedure that returns a pointer to its activation record
  + Activation record is like information about the stack, how you come to that function and what type of local information on top of the stack.
* Implemented as simple function that just returns its pointer to activation function
* When you return a pointer to your activation function and if it remains there, then you already defined your local variables, local methods… that you can now use as an object

Object

* Activation record produced by call to a class
  + just a pointer

Object access

* Access any local variable or procedures using dot notation: object.var
  + Address information is there, you are gonna decide where that function, variable is gonna be

Memory management

* Objects are garbage collected
  + user destructors considered undesirable

Diagram

Description automatically generated***Example: Circles and lines***

Problem

* Find the center and radius of the circle passing through three distinct points, p, q, and r

Solution

* Draw intersecting circles Cp, Cq around p,q and circles Cq’, Cr around q, r (Picture assumes Cq = Cq’)
* Draw lines through circle intersections
* The intersection of the lines is the center of the desired circle
* Error if the points are colinear

Approach in Simula:

Methodology

* Represent points, lines, and circles as objects
* Equip objects with necessary operations

Operations

* Point
  + equality(anotherPoint) : boolean
  + distance(anotherPoint) : real (needed to construct circles)
* Line
  + parallelto(anotherLine) : boolean (to see if lines intersect)
  + meets(anotherLine) : REF(Point)
* Circle
  + intersects(anotherCircle) : REF(Line)

Simula Point Class:

A picture containing text

Description automatically generated

Activation Records

Modern imperative programming languages typically have local variables

* Created upon entry to function
* Destroyed when function returns

Each invocation of a function has its own instantiation of local variables

* Recursive calls to a function require several instantiations to exist simultaneously
* Functions return only after all functions it calls have returned 🡪 last-in-first-out (LIFO) behavior.
* A LIFO structure called a stack is used to hold each instantiation

The portion of the stack used for an invocation of a function is called the function’s activation record

Representation of objects

Diagram

Description automatically generated

p is pointer to activation record.

You start as class is a function that never returns. You create local variables so to say, build the code for subfunctions and put them somewhere and put the code in the activation record.

proc equals is pointer to where the code is.

From functions, you know how to go back to your activation record to look at the global variables (x and y).

Simula Line Class:

A picture containing graphical user interface

Description automatically generated

Messages are resolved by “activationrecord.something”.

Derived Classes in Simula

Diagram

Description automatically generated with medium confidence

A class B:

* B class is derived from A
* B is gonna create an activation record and you are gonna add at the top of A.

Class in Simula is simply a function call which creates certain things as part of the activation record and then just returns a pointer to activation record.

Main Object-Oriented Features

Classes

Objects

* Instances of classes as activation records

Inheritance (“class prefixing”)

Subtyping

Virtual methods

* No implementation in base class
* A function can be redefined in subclass
* Which things will be implemented and which things are gonna be reused
* It is not simple to implement if you are just using activation records

Inner

* Combines code of superclass with code of subclass

Inspect/Qua

* run-time class/type tests
* run-time capabilities on operation on objects
* user’s perspective of an object
* what can I do with that object, on run-time
* 2 things
  + at run-time, I wanna check if object is there
  + what type of methods I can send to this object
* Qua is giving us a way to where to get that specific method for the object of a derived class

Inspect and Qua

Following the two are the same (XA is object of some class that has Show):



Or,

if XA is an activation record, an object, do the show method

A picture containing text

Description automatically generated

Or,

Text

Description automatically generated

Text, letter

Description automatically generated

Features Absent from Simula 67

Encapsulation

* All data and functions accessible; no private, protected

Self/Super mechanism of Smalltalk

* But has an expression this<class> to refer to object itself, regarded as object of type <class>. Not clear how powerful this is…

Class variables

* But can have global variables
* No class specific things, no public and private

Exceptions

* Not fundamentally an OO feature ...

Simula Summary

Class

* ”procedure" that returns ptr to activation record
* initialization code always run as procedure body

Objects: closure (or activation record) created by a class

Encapsulation

* protected and private not recognized in 1967
* added later and used as basis for C++

Subtyping: determined by class hierarchy

Inheritance: provided by class prefixing in front of derived class

A closure is a function or reference to a function together with a referencing environment. A closure allows a function to access non-local variables even when invoked outside its immediate lexical scope.

Outline

Central concepts in object-oriented languages

* Dynamic lookup, encapsulation, subtyping, inheritance

Objects as activation records

* Simula – implementation as activation records with static scope

Pure dynamically-typed object-oriented languages

* Object implementation and run-time lookup
* Class-based languages (Smalltalk)
* Prototype-based languages (Self, JavaScript)

Statically-typed object-oriented languages

* C++ – using static typing to eliminate search
* C++ – problems with C++ multiple inheritance
* Java – using Interfaces to avoid multiple inheritance

Smalltalk

A person sitting at a piano

Description automatically generated with medium confidence

You don’t want your programmer to be able to mess up with internal state of what you programmed.

In functional programming, I don’t want to do any type of state change.

Here we are talking about encapsulation and abstraction to avoid users to mess up with the internal state of what we have programmed.

Major language that popularized objects

Developed at Xerox PARC

* Smalltalk-76, Smalltalk-80 were important versions

Object metaphor extended and refined

* Used some ideas from Simula, but it is a very different language
* Everything is an object, even a class
  + If a class is an object, I can ask that class to create another object for me (an instance of that class). I can talk to that class to create a new class and perhaps a derived class then I can expand that class as an object and use that object to send a message to that one, to create an instance…
* All operations are “messages to objects”
* Very flexible and powerful language
  + Similar to “everything is a list” in Lisp, but more so
  + Example: object can detect that it has received a message it does not understand, can try to figure out how to respond by trying to look out perhaps

Idea of object in Smalltalk is generic idea.

Motivating Application: Dynabook

Diagram

Description automatically generatedConcept developed by Alan Kay

Small portable computer

* Revolutionary idea in early 1970’s
  + At the time, a minicomputer was shared by 10 people, stored in a machine room
* What would you compute on an airplane?

Influence on Smalltalk

* Language intended to be programming language and operating system interface
* Intended for “non-programmer”
* Syntax presented by language- specific editor

Smalltalk Language Terminology

* Object Instance of some class
  + Class is object as well, which one comes first? It is like chicken egg problem
  + If you define an object that you can start everything with, then problem is solved
* Class Defines behavior of its objects
  + Class is an object but it can create new objects
  + Basic idea in a class object is to define the behaviour of its objects
* Selector Name of a message
* Message Selector together with parameter values
* Method Code used by a class to respond to message
  + If a class understands a message, if it is able to respond to that message, it will be responding to it using the method that matches that
* Instance variable Data stored in object
  + Class, since it is an object, it can have its data stored inside as well. Therefore you can think of a classes instance variables to be the class variable that you would think of in other languages.
* Subclass Class defined by giving incremental modifications to some superclass

***Example: Point Class***

Class definition written in tabular form:

Table

Description automatically generated

“super class : Object”

* Base thing for every object to be based on

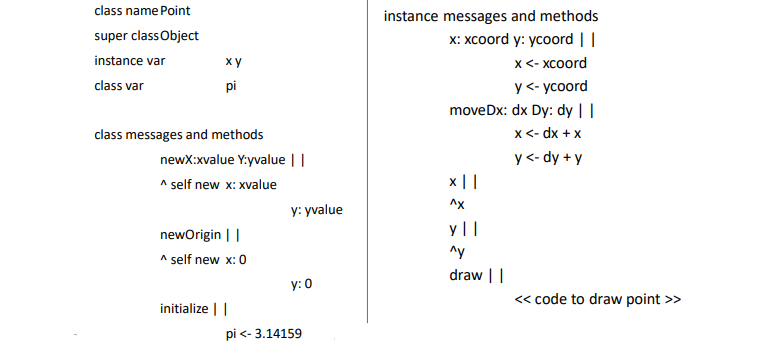
class var is available to anyone that has access to Point class

instance variables are the object level things that objects that are created from this class will have their instance variables.

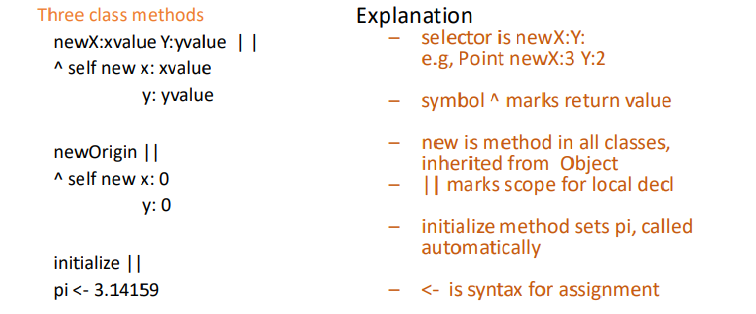
class messages are understood by this class and methods respond to those messages. These are related to class object itself.

instance messages are understood by the instances of this class as object.

Code:



Class messages and methods:



Instance messages and methods:

Graphical user interface

Description automatically generated with medium confidence

Run-time Representation of Point:

Diagram

Description automatically generated

Template will have the name and signatures of its instance variables and so forth.

Method dictionary will have messages that it can understand and the code that defines the behaviour how to respond to those messages.

When you create an object of this point, you will have this type of data structure:

* one part of the data is pointer to class itself
* some memory. This memory will be values of these instance variables. In the template I know that I have 2 variables x and y. 3 and 2 are the x and y values that are used.

*How lookup is gonna happen?*

I am currently with my object. I want to send a message to it: “I want you to move to another location.”.

Its behavior is defined in the class that it is being generated off. I have pointer to go and look at to get a access to the class.

In the class, I have 2 things:

* Templates (class instance variables)
* Method dictionary that defines what messages that I can get and where the code for those messages are

If you send me a move message, I take it, look at a point class, go to method dictionary, search the method dictionary for my move. If it is there, get the code. If the message is not found, it needs to be at super class. I go to super class object and look at the method dictionary. If there is no message there neither, perhaps there is another superclass. At the end, if there is nothing, then I can say this is not gonna take the message that I have sent it to.

This code can access to templates as well as the method dictionary and also the instance values.

*Inheritance:*



create (constructor)  
not overridden, newly defined

draw was already exist in Point class.

Run-time Representation:

Diagram

Description automatically generated

Method dictionary of derived ColorPoint class has only new ones or overridden ones.

move function is going to be found in super class’s method dictionary.

Dynamic lookup comes with a cost. If I start with my ColorPoint object, to find the move, I have to search in my method dictionary and also in super class’s method dictionary. 4 indirections and 2 list search.

In C++ we solve this like this:

* Maybe you don’t have ColorPoint’s method dictionary here. You need to put it somewhere closer to ColorPoint object and expand that dictionary (v table) and your search become just an indirection instead of real search.

Encapsulation in Smalltalk

Methods are public

* If there is private, protected concept, you have to have private method dictionary, public method dictionary seperately

Instance variables are hidden

* Not visible to other objects
  + pt x is not allowed unless x is a method
* But may be manipulated by subclass methods
  + This limits ability to establish invariants
  + Example:
    - Superclass maintains sorted list of messages with some selector, say insert
    - Subclass may access this list directly, rearrange order

Smalltalk Summary

Class

* creates objects that share methods
* pointers to template, dictionary, parent class

Objects: created by a class, contains instance variables

Encapsulation

* methods public, instance variables hidden

Subtyping: implicit, no static type system

Inheritance: subclasses, self, super

* Single inheritance in Smalltalk-76, Smalltalk-80
* There is no multiple inheritance

SELF PROGRAMMING LANGUAGE

Prototype-based pure object-oriented language

Purely dynamic again

Designed by Randall Smith (Xerox PARC) and David Ungar (Stanford University)

* Successor to Smalltalk-80
* “Self: The power of simplicity” appeared at OOPSLA ‘87
* Initial implementation done at Stanford; then project shifted to Sun Microsystems Labs
* Vehicle for implementation research

Self 4.4 available from <http://selflanguage.org/>

A dialect of Smalltalk

Dynamically typed & just-in-time compilation & prototype-based approach to objects

Prototype-based Programming

No classes

Objects (that are built) inherit directly from other objects through a prototype property (Self, JavaScript, lo)

Pros:

* Simpler language rules, no need to answer questions e.g., is class an object?, if so what class is it an instance of?, …
* Any object can be given specialized behavior, e.g., debugging and object – override a method of the object…
* Better control, e.g., singleton – do not give it a copy method…

Singleton Pattern

Only one instance of a class needed throughout the system

* Objects needed for logging, communication, database

How to ensure a class has only one instance and can be accessed globally?

* You can use global variables

Pros

* More control over instance – can add additional logic to access, e.g., dynamic creation (cannot be done with globals), synchronizing access (per thread vs per process)
* Polymorphism, e.g., single graphics card interface, many machines

Cons

* Still context dependent as global variables

Design Goals

Conceptual economy

* Everything is an object
* Everything done using messages
* No classes
* No variables

Concreteness

* Objects should seem “real”
* GUI to manipulate objects directly

How Successful?

Self is a carefully designed language

Few users: not a popular success

* No compelling application, until JavaScript
* Influenced development of object calculi w/o classes

However, many research innovations

* Very simple computational model
* Enormous advances in compilation techniques
* Influenced the design of Java compilers

Language Overview

Dynamically typed

* Type definition and type checks etc. the typing system is maintained dynamically.
* When I look at the code, I cannot trace things from one place to another.

Everything is an object

All computation via message passing

Creation and initialization done through messaging: clone object

Operations on objects:

* send messages
* add new slots
  + adding new behaviours to that object
* replace old slots
  + replace old behaviours
* remove slots
  + remove some behaviours

Objects and Slots

Object consists of named slots

* Data
  + Such slots return contents upon evaluation; so act like instance variables
* Assignment
  + Set the value of associated slot
* Method
  + Slot contains Self code
* Parent
  + Point to existing object to inherit slots

Diagram

Description automatically generated

objects

Diagram, engineering drawing

Description automatically generated

Diagram, engineering drawing

Description automatically generated

Chart, box and whisker chart

Description automatically generatedMessages and Methods

When message is sent, object searched for slot with name

If none found, all parents are searched

* Runtime error if more than one parent has a slot with the same name

If slot is found, its contents evaluated and returned

* Runtime error if no slot found

Table

Description automatically generated

Mixing State and Behaviour

Diagram, table

Description automatically generated

Object Creation

To create an object, we copy an old one

We can add new methods, override existing ones, or even remove methods

These operations also apply to parent slots

Changing Parent Pointers

Table

Description automatically generated

Table

Description automatically generated

Disadvantages of classes?

Classes require programmers to understand a more complex model

* To make a new kind of object, we have to create a new class first
* To change an object, we have to change the class
* Infinite meta-class regression

But: Does Self require programmer to reinvent structure?

* Common to structure Self programs with traits: objects that simply collect behavior for sharing

JavaScript Prototype

Graphical user interface, text, application

Description automatically generated

With “p1.color”, I am adding a property, a slot to that object and setting its value to “green”.

After adding distancetoorigin method, we have 3 instance variables and additional function.

Now we have another class.

First class has instance in p1. That class only has two variables xc and yc.

At the end of code, I have another class which is a new concept but I don’t have the class somewhere in there as an entity. It is just a concept. That concept came around by just me modifying the original concept and adding 2 new things, 2 new slots to that particular thing.

This is how prototype-base languages work.

Prototype here is actually adding new things to my existing object on the fly (runtime), a prototype.

Every JavaScript object has a prototype

* Object literals linked to Object.prototype
* Otherwise, prototype based on constructor

function Foo() {

this.x = 1;

}

obj = new Foo;

Changing the JavaScript prototype

* The prototype property is immutable
* Changes to prototype property inherited immediately

Concept implementation and instantiation of that concept is not separated here. Concept is in your mind which is a class. What you have is only the objects. Objects are keeping the specific concept in implementation as it is, it is real.

Problem:

* At the end of the code. If I ask you what your object is doing, what is the semantic behing that object, you have to look at the code.

Concept is not anymore written in a class, but it is kept somewhere in your mind.

You are operating on objects, not with classes. You are modifying objects to create new concepts.

I can maintain method dictionary. To get to a few levels up, in the hierarchy of these concepts, you’ll have indirections. Not very efficient in terms of implementation.

Runtime efficiency of these things will be quite bad compared to more efficiency oriented languages like C++.

Outline

Central concepts in object-oriented languages

* Dynamic lookup, encapsulation, subtyping, inheritance

Objects as activation records

* Simula – implementation as activation records with static scope

Pure dynamically-typed object-oriented languages

* Object implementation and run-time lookup
* Class-based languages (Smalltalk)
* Prototype-based languages (Self, JavaScript)

Statically-typed object-oriented languages

* C++ – using static typing to eliminate search
* C++ – problems with C++ multiple inheritance
* Java – using Interfaces to avoid multiple inheritance

**C++ BACKGROUND**

C++ is an object-oriented extension of C

C was designed by Dennis Ritchie at Bell Labs

* used to write Unix, based on BCPL

C++ designed by Bjarne Stroustrup at Bell Labs

* His original interest at Bell was research on simulation
* Early extensions to C are based primarily on Simula
* Called “C with classes” in early 1980’s
* Popularity increased in late 1980’s and early 1990’s
* Features were added incrementally
  + Classes, templates, exceptions, multiple inheritance, type tests...

C++ Design Goals

Provide object-oriented features in C-based language, without compromising efficiency

* Backwards compatibility with C
* Better static type checking
* Data abstraction
* Objects and classes
* Prefer efficiency of compiled code where possible
  + Things need to be static

Important principle

* If you do not use a feature, your compiled code should be as efficient as if the language did not include the feature. (compare to Smalltalk)

How successful?

Given the design goals and constraints,

* this is a very well-designed language

Many users -- tremendous popular success

However, very complicated design

* Many features with complex interactions
* Difficult to predict from basic principles
* Most users chose a subset of language
  + Full language is complex and unpredictable
* Many implementation-dependent properties

Significant Constraints

C has specific machine model

* Access to underlying architecture

No garbage collection

* Consistent with goal of efficiency
* Need to manage object memory explicitly

Local variables stored in activation records

* Objects treated as generalization of structs
  + Objects may be allocated on stack and treated as L-values
  + Stack/heap difference is visible to programmer

C++ Object System

Object-oriented features

* Classes
* Objects, with dynamic lookup of virtual functions
* Inheritance
  + Single and multiple inheritance
  + Public and private base classes
* Subtyping
  + Tied to inheritance mechanism
* Encapsulation
  + Public, private, protected visibility

Some Good Decisions

Public, private, protected levels of visibility

* Public: visible everywhere
* Protected: within class and subclass declarations
* Private: visible only in class where declared

Friend functions and classes

* Careful attention to visibility and data abstraction

Allow inheritance without subtyping

* Better control of subtyping than without private base classes

Some Problem Areas

Casts

* Sometimes no-op, sometimes not (e.g., multiple inheritance)

Lack of garbage collection

* Memory management is error prone
  + Constructors, destructors are helpful // smart pointers?

Objects allocated on stack

* Better efficiency, interaction with exceptions
* But assignment works badly, possible dangling ptrs

Overloading

* Too many code selection mechanisms?

Multiple inheritance

* Emphasis on efficiency leads to complicated behavior

Sample Class: 1D Points

Graphical user interface, text, application

Description automatically generated

Virtual Functions

Member functions are either

* Virtual, if explicitly declared or inherited as virtual
* Non-virtual otherwise

Virtual functions

* Accessed by indirection through ptr in object
* May be redefined in derived (sub) classes

Non-virtual functions

* Are called in the usual way through stack. Just ordinary functions.
* Cannot redefine in derived classes (except overloading)
* Cannot be inherited
* Better performance

Pay overhead only if you use virtual functions

* For indirection of finding the code through some pointer mechanism

Sample Derived Class

Graphical user interface, text, application

Description automatically generated

*Run-time Representation:*

Diagram

Description automatically generated

Point object is created on stack.

vtable is similar to method dictionary or template we have in Smalltalk. vtable is more efficient.

I don’t know the method names, I know their offsets. I reach to darken function with just 1 indirection and just an offset.

In Smalltalk, we were checking the name by searching the method dictionary.

*Compare to Smalltalk/JavaScript:*

Diagram

Description automatically generated

*Run-time Representation:*

Diagram

Description automatically generated

Single vtable for each class.

Base class methods are at the top of the vtable.

Why is C++ lookup simpler?

Smalltalk/JavaScript have no static type system

* Code obj.operation(pars) could refer to any object
  + This is done at runtime
* Need to find method using pointer from object
* Different classes will put methods at different place in method dictionary

C++ type gives compiler some superclass

* Offset of data, fctn ptr same in subclass and superclass
  + This is coming from the static typing
* Offset of data and function ptr known at compile time
* Code p->move(x) compiles to equivalent of

(\*(p -> vptr[0]))(p,x) if move is first function in vtable

(p,x) are arguments

Looking Up Methods

Diagram

Description automatically generated

Looking Up Methods 2

Diagram

Description automatically generated

move is always the first entry.

Calls to Virtual Functions

Text, letter

Description automatically generated

“This” Pointer (self in Smalltalk)

Code is compiled so that member function takes “object itself” as first argument

Code int A::f(int x) { … g(i) …;}

compiled as int A::f(A \*this, int x) { … this->g(i) …;}

“this” pointer may be used in member function

* Can be used to return pointer to object itself, pass pointer to object itself to another function, ...

Non-virtual Functions

How is code for non-virtual function found?

Same way as ordinary “non-member” functions:

* Compiler generates function code and assigns address
* Address of code is placed in symbol table next to particular function name
* At call site, address is taken from symbol table and placed in compiled code

Overloading

* Remember: overloading is resolved at compile time
* This is different from run-time lookup of virtual function
* Rename them in the symbol table accordingly

Overloading helps you to achieve some result that requires different algorithms. So code changes.

In polymorphism, your code stays the same but arguments change. A lot to do with typing system.

Virtual vs Overloaded Functions

Graphical user interface, text, application

Description automatically generated

Outline

Central concepts in object-oriented languages

* Dynamic lookup, encapsulation, subtyping, inheritance

Objects as activation records

* Simula – implementation as activation records with static scope

Pure dynamically-typed object-oriented languages

* Object implementation and run-time lookup
* Class-based languages (Smalltalk)
* Prototype-based languages (Self, JavaScript)

Statically-typed object-oriented languages

* C++ – using static typing to eliminate search
* C++ – problems with C++ multiple inheritance
* Java – using Interfaces to avoid multiple inheritance

Multiple Inheritance

Diagram

Description automatically generated

*Problem: Name Clashes:*

Diagram

Description automatically generated with medium confidence

*Possible Solutions to Name Clash:*

Three general approaches

* Implicit resolution
  + Language resolves name conflicts with arbitrary rule
  + User has to know this rule, following the behaviour is not good
* Explicit resolution
  + Programmer must explicitly resolve name conflicts
  + If there is a name clash, I will let you know. You solve it 🡪 we say this to user
* Disallow name clashes
  + Programs are not allowed to contain name clashes
  + Restriction

No solution is always best

C++ uses explicit resolution

*Repair to Previous Example:*

Rewrite class C to call A::f explicitly:

Text

Description automatically generated

Reasonable solution

* This eliminates ambiguity
* Preserves dependence on A
  + Changes to A::f will change C::f

vtable for Multiple Inheritance

Text

Description automatically generated

Table

Description automatically generated with low confidence

Multiple Inheritance “Diamond”

Diagram

Description automatically generated

Graphical user interface

Description automatically generated with medium confidence

Make D the virtual class. Inherited code will not be allowed. There is no code for the behaviour in D. Therefore A and B must implement their behaviour.

You don’t see this in Smalltalk because there is no multiple inheritance there.

Outline

Central concepts in object-oriented languages

* Dynamic lookup, encapsulation, subtyping, inheritance

Objects as activation records

* Simula – implementation as activation records with static scope

Pure dynamically-typed object-oriented languages

* Object implementation and run-time lookup
* Class-based languages (Smalltalk)
* Prototype-based languages (Self, JavaScript)

Statically-typed object-oriented languages

* C++ – using static typing to eliminate search
* C++ – problems with C++ multiple inheritance
* Java – using Interfaces to avoid multiple inheritance

Java Language Background

James Gosling and others at Sun, 1990 - 95

Oak language for “set-top box”

* small networked device with television display
  + graphics
  + execution of simple programs
  + communication between local program and remote site
  + no “expert programmer” to deal with crash, etc.

Internet applications

* simple language for writing programs that can be transmitted over network
* not an integrated web scripting language like JavaScript

Design Goals

Portability

* Internet-wide distribution: PC, Unix, Mac

Reliability

* Avoid program crashes and error messages

Safety

* Programmer may be malicious

Simplicity and familiarity

* Appeal to average programmer; less complex than C++

Efficiency

* Important but secondary

General Design Decisions

Simplicity

* Almost everything is an object (Similar to SELF or JAVASCRIPT)
* All objects on heap, accessed through pointers
  + there is no stack type of treatment
* No functions, no multiple inheritance, no go to, no operator overloading, few automatic coercions

Portability and network transfer

* Bytecode interpreter on many platforms

Reliability and Safety

* Typed source and typed bytecode language
* Run-time type and bounds checks
* Garbage collection

Language Terminology

Class, object – as in other languages

Field – data member

Method – member function

Static members – class fields and methods

this – self

Package – set of classes in shared namespace

Native method – method compiled from in another language, often C

Java Classes and Objects

Syntax similar to C++

Object

* has fields and methods
* is allocated on heap, not run-time stack
* accessible through reference (only ptr assignment)
* garbage collected

Dynamic lookup

* Similar in behavior to other languages
* Static typing => more efficient than Smalltalk
* Dynamic linking, interfaces => slower than C++

Point Class

Text

Description automatically generated

Visibility similar to C++, but not exactly

Object Initialization

Java guarantees constructor call for each object

* Memory allocated
* Constructor called to initialize memory
* Some interesting issues related to inheritance

Cannot do this (would be bad C++ style anyway):

Obj\* obj = (Obj\*)malloc(sizeof(Obj));

Static fields of class initialized at class load time

Garbage Collection and Finalize

Objects are garbage collected

* No explicit free
* Avoids dangling pointers and resulting type errors

Problem

* What if object has opened file or holds lock?

Solution

* finalize method, called by the garbage collector
  + Before space is reclaimed, or when virtual machine exits
  + Space overflow is not really the right condition to trigger finalization when an object holds a lock...)
* Important convention: call super.finalize

Encapsulation and Packages

Graphical user interface, application

Description automatically generated

Visibility and Access

Four visibility distinctions

* public, private, protected, package

Method can refer to

* private members of class it belongs to
* non-private members of all classes in same package
* protected members of superclasses (in diff package)
* public members of classes in visible packages

Visibility determined by files system, etc. (outside language)

Qualified names (or use import)

Diagram

Description automatically generated

Inheritance

Similar to Smalltalk, C++

Subclass inherits from superclass

* Single inheritance only (but Java has interfaces)

Some additional features

* Conventions regarding super in constructor and finalize methods
* Final classes and methods

Example Subclass

Text, application

Description automatically generated

Class Object

Every class extends another class

* Superclass is Object if no other class named

Methods of class Object

* GetClass – return the Class object representing class of the object
* ToString – returns string representation of object
* equals – default object equality (not ptr equality)
* hashCode
* Clone – makes a duplicate of an object
* wait, notify, notifyAll – used with concurrency
* finalize

Constructors and Super

Java guarantees constructor call for each object

This must be preserved by inheritance

* Subclass constructor must call super constructor
  + If first statement is not call to super, then call super() inserted automatically by compiler
  + If superclass does not have a constructor with no args, then this causes compiler error
  + Exception to rule: if one constructor invokes another, then it is responsibility of second constructor to call super, e.g.,
    - ColorPoint() { ColorPoint(0,blue);}

is compiled without inserting call to super

Different conventions for finalize and super

* Compiler does not force call to super finalize

Final Classes and Methods

Restrict inheritance

* Diagram

  Description automatically generatedFinal classes and methods cannot be redefined

Example

* java.lang.String

Reasons for this feature

* Important for security
  + Programmer controls behavior of all subclasses
  + Critical because subclasses produce subtypes
* Compare to C++ virtual/non-virtual
  + Method is “virtual” until it becomes final

Java Interfaces

Graphical user interface, application

Description automatically generated

Interfaces vs Multiple Inheritance

C++ multiple inheritance

* A single class may inherit from two base classes
* Constraints of C++ require derived class representation to resemble all base classes

Java interfaces

* A single class may implement two interfaces
* No inheritance (of implementation) involved
* Java implementation does not require similarity between class representations
  + For now, think of Java implementation as Smalltalk/JavaScript implementation, although the Java type system supports some optimizations

Subtyping and Inheritance

Interface

* The external view of an object

Subtyping

* Relation between interfaces

Implementation

* The internal representation of an object

Inheritance

* Relation between implementations

Example: Smalltalk Point class

Table

Description automatically generated

*Subclass: ColorPoint*

Table

Description automatically generated

*Object Interfaces*

Interface

* The messages understood by an object

Example: point

* x:y: set x,y coordinates of point
* moveDx:Dy: method for changing location
* x returns x-coordinate of a point
* y returns y-coordinate of a point
* draw display point in x,y location on screen

The interface of an object is its type

*Subtyping*

Graphical user interface, text, application

Description automatically generated

*Implicit Object Types – Smalltalk/JS*

Each object has an interface

* Smalltalk: set of instance methods declared in class
* Example:
  + Point { x:y:, moveDx:Dy:, x, y, draw}
  + ColorPoint { x:y:, moveDx:Dy:, x, y, color, draw}
* This is a form of type
* Names of methods, does not include type/protocol of arguments

Object expression and type

* Send message to object
  + p draw p x:3 y:4
  + q color q moveDx: 5 Dy: 2
* Expression OK if message is in interface

Subtyping

Relation between interfaces

* Suppose expression makes sense
  + p msg:pars -- OK if msg is in interface of p
* Replace p by q if interface of q at least contains entire interface of p

Subtyping

* If interface is superset, then a subtype
* Example: ColorPoint subtype of Point
* Sometimes called “conformance”

Can extend to more detailed interfaces that include types of parameters

Subtyping and Inheritance

Smalltalk/JavaScript subtyping is implicit

* Not a part of the programming language
* Important aspect of how systems are built

Inheritance is explicit in Java

* Used to implement systems
* No forced relationship to subtyping

Smalltalk Collection Hierarchy

Diagram

Description automatically generated

C++ Subtyping

Text

Description automatically generated

Implementation of Subtyping

Graphical user interface, text, application

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Table

Description automatically generated with medium confidence

Text

Description automatically generated with low confidence

2 independent classes cannot be subtypes.

Why C++ design?

Client code depends only on public interface

* In principle, if ColorPoint interface contains Point interface, then any client could use ColorPoint in place of point
* However -- offset in virtual function table may differ
* Lose implementation efficiency (like Smalltalk)

Without link to inheritance

* Subtyping leads to loss of implementation efficiency

Also encapsulation issue:

* Subtyping based on inheritance is preserved under modifications to base class …

Recurring Subtype Issue: Downcast

The Simula type of an object is its class

Simula downcasts are checked at run-time

Text

Description automatically generated

We have to be sure a has all the messages that b can have to downcast.

Function Subtyping

Subtyping principle

* A <: B if an A expression can be safely used in any context where a Bexpression is required

Subtyping for function results

* If A <: B, then C🡪A <: C🡪B

Subtyping for function arguments

* If A <: B, then B🡪C <: A🡪C

Terminology

* Covariance: A <: B implies F(A) <: F(B)
* Contravariance: A <: B implies F(B) <: F(A)

*Examples:*

Chart, radar chart

Description automatically generated

circle🡪circle and shape🡪shape ------> covariance

circle🡪shape and shape🡪circle ------> contravariance

Subtyping with Functions

Graphical user interface, text, application

Description automatically generated

Java Types

Two general kinds of types

* Primitive types – not objects
  + Integers, Booleans, etc
* Reference types
  + Classes, interfaces, arrays
  + No syntax distinguishing Object \* from Object

Static type checking

* Every expression has type, determined from its parts
* Some auto conversions, many casts are checked at run time
* Example, assuming A <: B
  + If A x, then can use x as argument to method that requires B
  + If B x, then can try to cast x to A
  + Downcast checked at run-time, may raise exception

Classification of Java Types

Diagram

Description automatically generated

Subtyping

Primitive types

* Conversions: int -> long, double -> long, …

Class subtyping similar to C++

* Subclass produces subtype
* Single inheritance => subclasses form tree

Interfaces

* Completely abstract classes
  + no implementation
* Multiple subtyping
  + Interface can have multiple subtypes (implements, extends)

Arrays

* Covariant subtyping – not consistent with semantic principles

Java Class Subtyping

Signature Conformance

* Subclass method signatures must conform to superclass

Three ways signature could vary

* Argument types
* Return type
* Exceptions

Java rule

* Java 1.1: Arguments and returns must have identical types, may remove exceptions
* Java 1.5: covariant return type specialization

Interface Subtyping: Example

Text, letter

Description automatically generated

Properties of Interfaces

Flexibility

* Allows subtype graph instead of tree
* Avoids problems with multiple inheritance of implementations (remember C++ “diamond”)

Cost

* Offset in method lookup table not known at compile, this is efficiency issue
* Different bytecodes for method lookup
  + one when class is known
    - don’t need for search
  + one when only interface is known
    - search for location of method
    - cache for use next time this call is made (from this line)

Array Types

Automatically defined

* Array type T[ ] exists for each class, interface type T
* Cannot extend array types (array types are final)
* Multi-dimensional arrays are arrays of arrays: T[ ] [ ]

Treated as reference type

* An array variable is a pointer to an array, can be null
* Example: Circle[] x = new Circle[array\_size]
* Anonymous array expression: new int[] {1,2,3, ... 10}

Every array type is a subtype of Object[ ], Object

* Length of array is not part of its static type

Array Subtyping

Covariance

* if S <: T then S[ ] <: T[ ]
  + Text, whiteboard

    Description automatically generated

Standard type error

* class A {…}
* class B extends A {…}
* B[ ] bArray = new B[10]
* A[ ] aArray = bArray // considered OK since B[] <: A[]
* aArray[0] = new A() // compiles, but run-time error

// raises ArrayStoreException

*Covariance problem again …*

Simula problem

* If A <: B, then A ref <: B ref
* Needed run-time test to prevent bad assignment
* Covariance for assignable cells is not right in principle

Explanation

* interface of “T reference cell” is
  + put : T 🡪 T ref
  + get : T ref 🡪 T
* Remember covariance/contravariance of functions

Java Exceptions

Similar basic functionality to other languages

* Constructs to throw and catch exceptions
* Dynamic scoping of handler

Some differences

* An exception is an object from an exception class
* Subtyping between exception classes
  + Use subtyping to match type of exception or pass it on …
  + Similar functionality to ML pattern matching in handler
* Type of method includes exceptions it can throw
  + Actually, only subclasses of Exception

Diagram

Description automatically generated

*Why define new exception types?*

Exception may contain data

* Class Throwable includes a string field so that cause of exception can be described
* Pass other data by declaring additional fields or methods

Subtype hierarchy used to catch exceptions

* catch <exception-type> <identifier> { … }
* will catch any exception from any subtype of exception-type and bind object to identifier

Subtyping concepts

Type of an object represents its interface

Subtyping has associated substitution principle

* If A <: B, then A objects can be used in place of B objects

Implicit subtyping in dynamically typed lang

* Relation between interfaces determines substitutivity

Explicit subtyping in statically typed languages

* Type checker may recognize some subtyping
* Issues: programming style, implementation efficiency

Covariance and contravariance

* Function argument types reverse order
* Problems with Java array covariance

Principles

* Object “width” subtyping
* Function covariance, contravariance
* Object type “depth” subtyping
* Subtyping recursive types

Applications of Principles

Dynamically typed languages

* If A <: B in principle, then can use A objects in place of B objects

C++

* Class subtyping only when public base class
* Compiler allows width subtyping, covariant depth subtyping. (Think about why…)

Java

* Class subtyping only when declared using “extends”
* Class and interface subtyping when declared
* Compiler allows width subtyping, covariant depth subtyping
* Additional typing issues related to generics

Java Language Implementation: Outline

Java virtual machine overview

* Loader and initialization
* Linker and verifier
* Bytecode interpreter

JVM Method lookup

* four different bytecodes

Verifier analysis

Method lookup optimizations (beyond Java)

Java security

* Buffer overflow
* Java “sandbox”
* Stack inspection

Java Implementation

Compiler and Virtual Machine

* Compiler produces bytecode
* Virtual machine loads classes on demand, verifies bytecode properties, interprets bytecode

Why this design?

* Bytecode interpreter/compilers used before
  + Pascal “pcode”; Smalltalk compilers use bytecode
* Minimize machine-dependent part of implementation
  + Do optimization on bytecode when possible
  + Keep bytecode interpreter simple
* For Java, this gives portability
  + Transmit bytecode across network

Diagram

Description automatically generated

A picture containing diagram

Description automatically generated

Class Loader

Runtime system loads classes as needed

* When class is referenced, loader searches for file of compiled bytecodeinstructions

Default loading mechanism can be replaced

* Define alternate ClassLoader object
  + Extend the abstract ClassLoader class and implementation
  + ClassLoader does not implement abstract method loadClass, but has methods that can be used to implement loadClass
* Can obtain bytecodes from alternate source
  + VM restricts applet communication to site that supplied applet

Static Members and Initialization (example issue in class loading and linking)

Graphical user interface, text, application

Description automatically generated

JVM Linker and Verifier

Linker

* Adds compiled class or interface to runtime system
* Creates static fields and initializes them
* Resolves names
  + Checks symbolic names and replaces with direct references

Verifier

* Check bytecode of a class or interface before loaded
* Throw VerifyError exception if error occurs

Verifier

Bytecode may not come from standard compiler

* Evil hacker may write dangerous bytecode

Verifier checks correctness of bytecode

* Every instruction must have a valid operation code
* Every branch instruction must branch to the start of some other instruction, not middle of instruction
* Every method must have a structurally correct signature
* Every instruction obeys the Java type discipline

Last condition is complicated.

Bytecode Interpreter

Standard virtual machine interprets instructions

* Perform run-time checks such as array bounds
* Possible to compile bytecode class file to native code

Java programs can call native methods

* Typically functions written in C

Multiple bytecodes for method lookup

* invokevirtual - when class of object known
* invokeinterface - when interface of object known
* invokestatic - static methods
* invokespecial - some special cases

Type Safety of JVM

Run-time type checking

* All casts are checked to make sure type safe
* All array references are checked to make sure the array index is within the array bounds
* References are tested to make sure they are not null before they are dereferenced

Additional features

* Automatic garbage collection
* No pointer arithmetic

A picture containing text, orange, dark

Description automatically generated

Chart

Description automatically generated

Field and Method Access

Instruction includes index into constant pool

* Constant pool stores symbolic names
* Store once, instead of each instruction, to save space

First execution

* Use symbolic name to find field or method

Second execution

* Use modified “quick” instruction to simplify search (caching)

Outline

Java virtual machine overview

* Loader and initialization
* Linker and verifier
* Bytecode interpreter

JVM Method lookup

* four different bytecodes

Verifier analysis

Method lookup optimizations (beyond Java)

Java security

* Buffer overflow
* Java “sandbox”
* Stack inspection

Java Security

Security

* Prevent unauthorized use of computational resources

Java security

* Java code can read input from careless user or malicious attacker
* Java code can be transmitted over network – code may be written by careless friend or malicious attacker

Java is designed to reduce many security risks

Java Security Mechanisms

Sandboxing

* Run program in restricted environment
  + Analogy: child’s sandbox with only safe toys
* This term refers to
  + Features of loader, verifier, interpreter that restrict program
  + Java Security Manager, a special object that acts as access control “gatekeeper”

Code signing

* Use cryptography to establish origin of class file
  + This info can be used by security manager

Buffer Overflow Attack

Most prevalent general security problem today

* Large number of CERT advisories are related to buffer overflow vulnerabilities in OS, other code

General network-based attack

* Attacker sends carefully designed network msgs
* Input causes privileged program (e.g., Sendmail) to do something it was not designed to do

Does not work in Java

* Illustrates what Java was designed to prevent

Text

Description automatically generated with low confidence

Graphical user interface

Description automatically generated

Graphical user interface

Description automatically generated

Graphical user interface

Description automatically generated

Graphical user interface, application

Description automatically generated