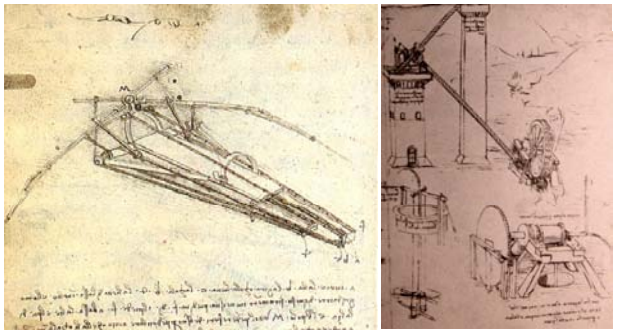
HY540 – Advanced Topics in Programming Language Development



Lecture 1

Introduction (1/7)

- Languages are tools
- Languages define programming ages
- Languages progress incrementally
- Languages progress mainly by practical demands
- Different languages may look very similar
- Old languages may look prehistoric at present
- But still a lot of prehistoric code around
- Newer languages will soon become obsolete
- No language will rule them all
- Language dominance depends on support and tools

Introduction (2/7)

- Development
  - Language design
    - Novel programming constructs
    - Enhanced programming practices
    - Coherence of elements
  - Language implementation
    - Algorithms and data structures
    - Software design and organization
    - Architecture solutions and patterns

Introduction (3/7)

This original technical report by Tony Hoare in 1973 makes the earliest known separation among the role of feature designers and entire language developers.

HINTS ON PROGRAMMING LANGUAGE DESIGN

BY

C. A. R. HOARE

SUPPORTED BY

ADVANCED RESEARCH PROJECTS AGENCY
ARPA ORDER NO. 2494
PROJECT CODE 3D30

DECEMBER 1973

Introduction (4/7)

(1) The designer of a new feature should concentrate on one feature at a time. If necessary, he should design it in the context of some well known programming language which he likes. He should make sure that his feature mitigates some disadvantage or remedies some incompleteness of the language, without compromising any of its existing merits. He should show how the feature can be simply and efficiently implemented. He should write a section of a user manual, explaining clearly with examples how the feature is intended to be used. He should check carefully that there are no traps lurking for the unwary user, which cannot be checked at compile time. He should write a number of example programs, evaluating all the consequences of using the feature, in comparison with its many alternatives. And finally if a simple proof rule can be given for the feature, this would be the final accolade.

Introduction (5/7)

(2) The language designer should be familiar with many alternative features designed by others, and should have excellent judgment in choosing the best, and rejecting any that are mutually inconsistent. He must be capable of reconciling, by good engineering design, any remaining minor inconsistencies or overlaps between separately designed features. He must have a clear idea of the scope and purpose and range of application of his new language, and how far it should go in size and complexity. He should have the resources to implement the language on one or more machines, to write user manuals, introductory texts, advanced texts; he should construct auxiliary programming aids and library programs and procedures; and finally, he should have the political will and resources to sell and distribute the language to its intended range of customers. One thing he should not do is to include untried ideas of his own. His task is consolidation, not innovation.

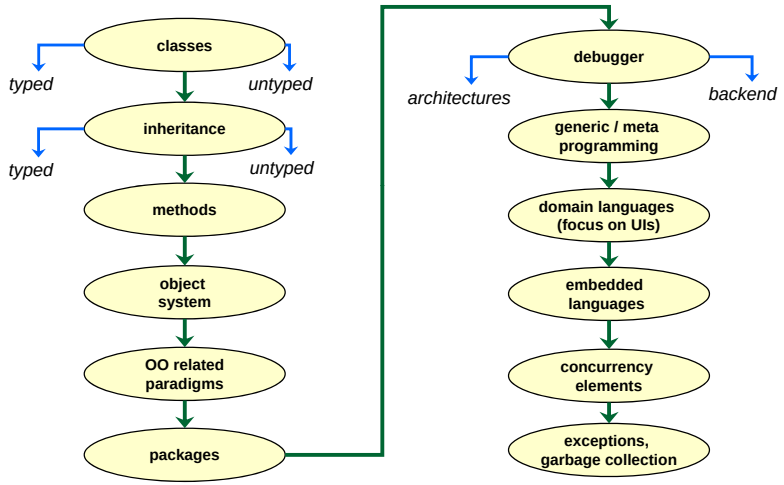
Introduction (6/7)

- Holly Grails (yes, there are many in programming languages)
  - Ultimate *concurrency*
  - Ultimate *safety*
  - Ultimate *reusability*
  - Ultimate *performance*
  - Ultimate *compositionality*
  - Ultimate *portability*
  - Ultimate *simplicity*
- Unfortunately, in practice they may expose conflicting requirements or incompatible solutions
- We do not know if all of them can be accomplished in a single language so we are in a constant quest

Introduction (7/7)

- Our view: that of a language developer
- Our targets: *reusability* and *compositionality*
- Our vehicles: *polymorphism* and *genericity*
- Our focus: dominant modern techniques
- Our bias: imperative programming paradigm
- Our philosophy: no language can have it all
- Our hobby: dig into debugger development
- Our lectures: based on selected papers and practice
- Your benefits: upgrade of programming philosophy
- Our advice: avoid becoming language developers

Roadmap



Classes (1/2)

- What is a class?
  - Classes are models of objects
  - They define **state** (data) and **behavior** (methods)
  - The structure of objects (runtime model or object structure model) is a matter of implementation
  - Objects are usually forbidden to access directly (from within their methods) the state of other objects
  - There is a relationship ***instance(C, O)*** among ***classes C*** and ***objects O*** linking objects to their class
  - Classes are a key ingredient of the OOP paradigm

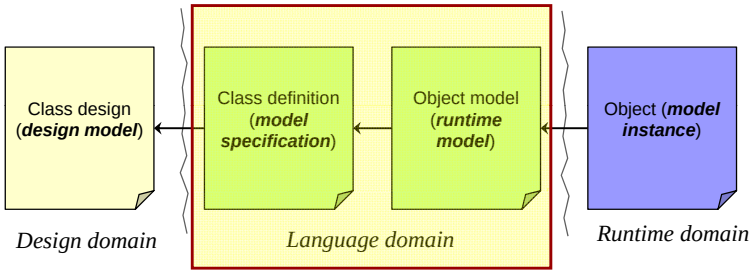
Classes (2/2)

- Where did classes came from?
  - An analogy of real world modeling - where objects exist - applied to software systems
    - Conceptual design perspective (**chicken-egg issue**)
  - Natural evolution of languages to support and model system components directly by the language
    - System design perspective (***object comes first***)
  - The explicit support of modules by the language reflecting the principles of modular design
    - Software engineering perspective (***class comes first***)

Typed classes – introduction (1/3)

- Concern languages offering ways (elements) to define new classes
- Every class definition implies the definition of a corresponding distinct data type
- Object creation is made possible using classes as factories
- The *instance* relationship is an integral part of the language semantics
- Classes may be values (first-class classes)
- ⇨ The languages of our focus are *class-based*

Typed classes – introduction (2/3)



Typed classes – introduction (3/3)

- Next we study
  - Object models
  - Class details
  - Typing issues
  - Dynamic classes

Object models

- Object models define the runtime image of instances encapsulating both state and behavior
- *There is an object meta-model defining how concrete models are derived*
- *The meta-model defines how field access is handled*
- Class-based languages compute object models by parsing class definitions and mapping them on the meta models
- The meta-model may be visible to the programmer and may be part of the language semantics

Object models
Memory region (1/5)

- Meta-model of any $class\ A\ \{\ T_1\ a_1\ \ldots\ T_n\ a_n\ \}$ is
  - Continuous program memory region
  - Region size is $sizeof(A) \geq sizeof(a_1)+\ldots+sizeof(a_n)$
  - $sizeof(T)$ for non-class types (built-in or user defined if supported) is well-defined
  - Alignment bytes may be added by the compiler
  - Memory layout is usually ... $low|a_1|a_2|\ldots|a_n|\ high$ ...
  - Field offset $offs(a_i) = if\ i=1\ 0\ else\ offs(a_{i-1})+sizeof(a_{i-1})$ Field or any other policy giving distinct offsets to fields
  - Methods are kept separately by the compiler
  - *Field binding is static, meaning field access at runtime is merely memory indirection, thus very fast*

Object models
Memory region (2/5)

- Assume ***class Point { double x, y; }***
- For the compiler classes are kept as objects encompassing hash tables for field and method access, trivially:

```
Class* point  = new Class("Point");
point->AddField("x", sizeof(double), "double");
point->AddField("y", sizeof(double), "double");
```


Object models
Memory region (3/5)

- Depending on the target instruction set, the class memory size may appear in the target code when claiming object memory (static, stack or heap)
  - ***Object(x): sizeof(x) =*** `GetObject("x")->GetClass()->SizeOf()`
  - ***Class(A): sizeof(A)=*** `Classes::Get("A")->SizeOf()`
- Let (C++) ***Point pt1;***
  - Local: ***sizeof(pt1)*** added to current block's computed size
  - Global: ***sizeof(pt1)*** added to current files' computed data size
- Let(C++) ***Point* pt2 = new Point;***
  - ***sizeof(Point)*** appears as a constant value in the target code instructions claiming the required dynamic object storage

Object models
Memory region (4/5)

- Special treatment for non-trivial field access (C++):
- Field pointer types: ***double Point::*x = &Point::x; pt.*x = 10;***

```
FieldPtr* x = new FieldPtr(Classes::Get("Point"), "x");
Offset offs = GetObject("pt")->GetOffset() + x->GetOffset();
```

  - Upon initialization (decl) they gain the field offset (constant value in the target code)
  - The `.*` special dereference operator is evaluated statically as ***offs(pt) + offs(Point:x)***
- Getting field address: ***double* px = &pt.x;***
  - The `&` operator is evaluated statically as ***offs(pt) + offs(Point:x)***

Object models
Memory region (5/5)

- Languages adopting the compiled memory region model (partially or entirely)
  - C++: visible to programmers, not part of the language documentation (implementation dependent), memory is addressable (and corruptible) by the program
  - Java, C#, Action Script (static classes): invisible to programmers, memory part of the virtual machine or runtime, not addressable by the program (untouchable)
  - *You cache memory loves the memory region model*

Object models
Field dictionary (1/5)

- Meta-model of any ***class A { T_1 a_1 … T_n a_n }*** is
  - String key (id) hash table with elements of a generic content type (can hold any value allowed in the language and method addresses), associated explicitly to its class via a reflection mechanism
  - Comes with reference semantics for objects (else a generic content value cannot be defined)
  - Common operations required by the dictionary are:
    - *New New<special_types> GetClass*
    - *GetField SetField GetStatic SetStatic*
    - *IsInstanceOf IsConvertibleTo*
    - *GetMethod InvokeMethod (redirected automatically to the class)*
  - *Field binding is dynamic, meaning field access at runtime is hashing (or search anyway), thus with a performance penalty*

Object models
Field dictionary (2/5)

- They are supported by class-based languages either compiling to byte code (run by virtual machines) or being interpreted
- Classes are usually represented at runtime as immutable referable objects (*reflection*)
- The dictionary model compromises speed for increased flexibility, which, for class-based language, means:
  - introduce and deploy new classes on-the-fly
  - extend objects beyond class boundaries
- Non-virtual methods are statically bound in class-based languages (even for dictionary object models)
- However, the performance penalty on field access requires field lookup caching

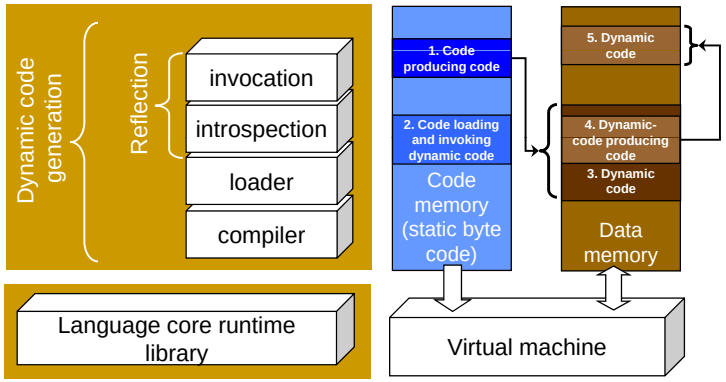
Object models
Field dictionary (3/5)

- Languages adopting the field dictionary model (partially or entirely)
  - Compiled:
    - C#, Java *(… only as a programmer model in some cases)*
      - Support memory regions, but allow more dynamic features via: compiler invocation, code / class loading and object creation and deployment via reflection, making the programmer think that field dictionary is on control (but is not)
    - Action Script
      - Mainly memory region (AS3), though allows to add fields and methods on objects of dynamic classes (true field dictionary)
  - Interpreted (class-based)
    - Python, Ruby, Cecil, Diesel
  - *You cache memory hates the field dictionary model*

Object models
Field dictionary (4/5)

- The big challenge is when models are mixed, that is we support both static and dynamic compilation, as supported in Java and C#
- At least the following language components should be part of the language library:
  - Compiler
  - Loader
  - Type information (Introspection / reflection / RTTI)
  - Execution (dynamic invocation)
- The dynamic objects cannot be statically typed, meaning they can only be generic object references.
- Thus, type checking should be done by client code using runtime type information (introspection).

Object models
Field dictionary (5/5)



Object models
Examples (1/12)

- *Java: dynamic class loading*
  - Provides `Class` that can be retrieved from every program object or be *loaded dynamically* like:

```
Class foo = Class.forNamestatic("foo");
Foo obj = (Foo) foo.newInstance();
```

  - The compiled class binary should reside in a place that the currently used class loader can get (e.g., in the class path, imported package, Jar file, etc)
    - For custom loading policies, one should build a custom made class loader
  - The Class API provides the required introspection and dynamic invocation functions
    - Access qualifiers do not apply, meaning encapsulation (information hiding) evaporates

Object models
Examples (2/12)

- *Java: instantiation and invocation*
  - The `Class` is a built-in type that offers introspection enabling to dynamically access methods and fields

```
Method[] getDeclaredMethods()

Method  getDeclaredMethod(String name, Class[] argTypes)
```

  - As observed, `argTypes` is necessary to be able retrieve versions of overloaded methods.
  - Normally, when unknown classes are loaded one cannot have type casting in the source code, meaning all access is bound to reflection.

Object models
Examples (3/12)

- *Java: Sun JVM memory layout (1/2)*
  - The JVM specification does not define a memory layout (!!!), but leaves it open to JVM developers choose one.
  - For example, to save memory, the Sun JVM doesn't lay out object's attributes in the order they are declared.
  - Instead, fields are organized in memory with the following descending type priority:
    - **double long int float short char boolean byte Object**
  - But then, how is field access compiled to support memory layouts, though not forcing one, and still be able to run by different JVMs using memory regions? We address this latter.

Object models
Examples (4/12)

- *Java: Sun JVM memory layout (2/2)*

```
class MyClass {
    byte a;
    int c;
    boolean d;
    long e;
    Object f;
}
```

Without reordering

```
[HEADER:   8 bytes]  8
[a:        1 byte ]  9
[padding:  3 bytes] 12
[c:        4 bytes] 16
[d:        1 byte ] 17
[padding:  7 bytes] 24
[e:        8 bytes] 32
[f:        4 bytes] 36
[padding:  4 bytes] 40
```

After reordering

```
[HEADER:   8 bytes]  8
[e:        8 bytes] 16
[c:        4 bytes] 20
[a:        1 byte ] 21
[d:        1 byte ] 22
[padding:  2 bytes] 24
[f:        4 bytes] 28
[padding:  4 bytes] 32
```

An instance of the java.lang.Boolean class takes 16 bytes of memory!

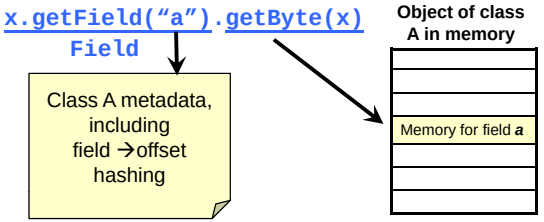
```
[HEADER:   8 bytes]  8
[value:    1 byte ]  9
[padding:  7 bytes] 16
```


Object models
Examples (5/12)

- *Java: code generation trick*
  - The Java compiler does not compute field offsets, but produces symbolic references for fields and methods
  - A symbolic reference is a numeric index (like #3) to the constant program pool where an entry exists with all information about the field (name, type, order)
  - The JVM will firstly compute the precise memory layout per class upon loading
  - Then, when loading the program code, for every field access it will produce a table (normal, not hashing) converting symbolic references to memory offsets
    - e.g., offsets[4] is the offset of field with symbolic reference #4

Object models
Examples (6/12)

- *Java: hybrid field access*
  - Static field access is similar to memory indirection, though an extra table access as explained before
  - Dynamic field access for objects created via reflection, although accessing objects of the same model (memory region), is technically similar to field dictionary



Object models
Examples (7/12)

- *C# : on-the-fly compilation*

```
using System;
using System.IO;
using Microsoft.CSharp;
using System.CodeDom.Compiler;
using System.Reflection;
        // Create C# compiler instance
CSharpCodeProvider csc = new CSharpCodeProvider();
ICodeCompiler icc = csc.CreateCompiler();
        // Set input params for the compiler
CompilerParameters co = new CompilerParameters();
co.OutputAssembly = "foo.dll";
co.ReferencedAssemblies.Add("system.dll");
        // Force to generate a DLL
co.GenerateExecutable = false;
        // Run the compiler for the source file
icc.CompileAssemblyFromFile(co, "foo.cs");
```

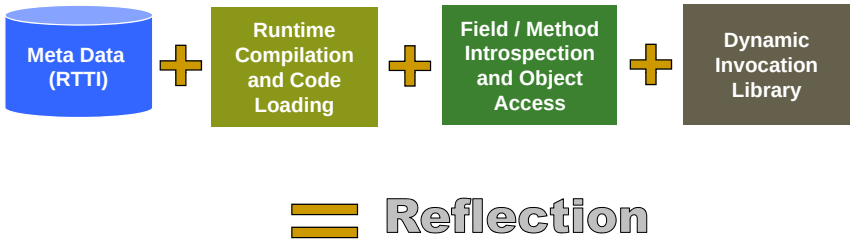
- This feature allows write programs that produce code which can extend a running system.
- Such programs can be seen as meta-programs.
- This technique can be used as a limited form of meta-programming.
- Interestingly, debugging of dynamically compiled code should not be a problem.

Object models
Examples (8/12)

- C# : *instance creation and method invocation*

```
        // Load newly generated assembly into ApplicationDomain namespace
Assembly asm = Assembly.LoadFrom("foo.dll");
Type t = asm.GetType("Foo");
BindingFlags bf =BindingFlags.DeclaredOnly |
                        BindingFlags.Public       |
                        BindingFlags.NonPublic    |
                        BindingFlags.Instance;
        // Construct an instance of the desirable class
Object obj = t.InvokeMember(
                        "FooConstructor", // or "" if none
                        bf | BindingFlags.CreateInstance, null, null, null
                        );
        // Invoke a method
t.InvokeMember(
        "Bar",
        bf | BindingFlags.InvokeMethod, null, obj, null
);
```


Object models
Examples (9/12)



Reflection in Delta (1/2)

```
const source    = "NullClass.dsc";
const asm       = "NullClass.dbc";
const nl        = "\n";

// Produce source on-the-fly and write it to a file.
result = strsavetofile(
           "function NullClass(){ return [ {.class : \"Null\"} ]; }",
           source
         );
assert result;

// Invoke compiler dynamically to compile the dynamic source.
result = vmcomp(source, asm, (function(error){ print(s, nl); }), false);
assert result;

// Load and initialise (run) the dynamic byte code.
vmrun(vm = vmload(asm, "NullClass"));

// Extract global methods (as object) through introspection.
funcs = vmfuncs(vm);

// Invoke a method (class constructor here).
a = funcs.NullClass();
print(a, nl); // Test purposes only.

// Add at shared space the class ctor so that other VMs can use it.
shwrite("NullClass", funcs.NullClass);

// Test deployment of dynamic class ctor via shared space.
print(shread("NullClass")(), nl);
```

Code manipulation is trivial (string handling)

Runtime compilation

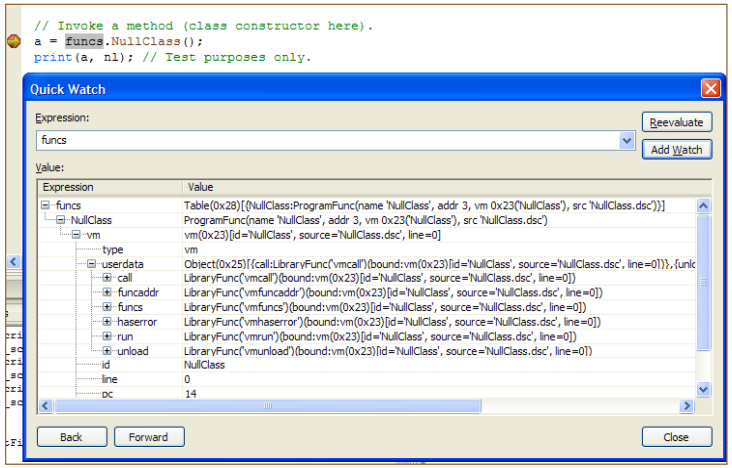
Dynamic code loading

Method introspection

Dynamic invocation, field and method access

Dynamic objects and methods are all normal first-class values

Reflection in Delta (2/2)



Zen debugger, within the ***Sparrow*** IDE of Delta: Full runtime debugging and inspection of dynamically loaded constructor methods and objects produced by these constructors

Object models
Examples (10/12)

- Action Script 3: *sealed and dynamic classes*
  - *Sealed* is a (normal) class whose objects can only have the fields and methods defined within the class definition
    - The best object model of a sealed class is memory region (memory efficient and fast), but this information is not available
      - *Although typed, it is surprising why some errors are runtime*
  - *Dynamic* is a class whose objects can introduce new *public* fields and methods not appearing in the respective class definition (note: term dynamic class is misused in AS3)
    - The best object model of a dynamic class is a memory region with one extra reference field as a field dictionary, but again this information is hidden

Object models
Examples (11/12)

- Action Script 3: *sealed and dynamic classes*
  - There are also some rules among sealed and dynamic classes that break polymorphism, although they preserve typing
    - sealed classes derived from dynamic are sealed (entirely)

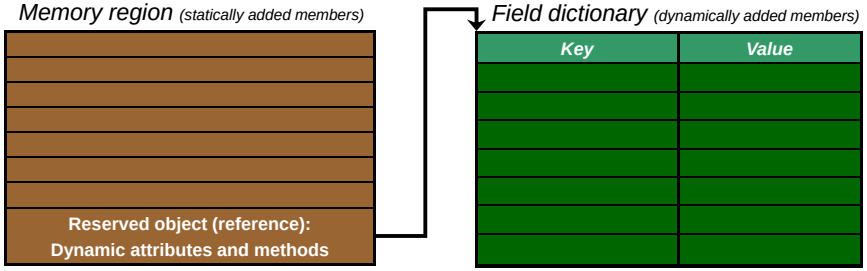
```
dynamic class Person
   { var name:String; }
Person p = new Person();
p.Name = "Joe";
p.Age = 25;
p.printMe = function () {
   trace (p.name, p.age);
}
p.printMe(); // Joe 25
```

```
public class Student extends Person
     {}
var ts:Person = new Student();
st.Age = 25;

>>>ReferenceError: Error #1056:
Cannot create property Age on
Student.
```


Object models
Examples (12/12)

- Action Script 3: *sealed and dynamic classes*
  - The hybrid object model is inevitable in languages allowing statements that introduce (or remove) extra slots (attributes or methods) beyond the class definition boundaries



Class details (1/10)

- *Static members*: shared by all objects of a class
  - Their support is easy by simply applying extra scoping rules, treating statics as global elements in the closed namespace of the class
  - May support special initialization blocks for statics in the scope of a class called during global initialization
    - Sometimes called static constructors (C#)
    - Handling requires support for code execution during initialization (C++ allows that via static objects, java not)

Class details(2/10)

- *Member access qualification*: usually handled by trivial compile-time checking to force access rules
  - *Unbiased* (i.e. apply equally to all other classes)
    - Common private / public qualifiers
  - *Biased* (i.e. some classes are treated differently)
    - *Unselective* (i.e. apply to all members, as friend qualifier in C++)
    - *Selective* (i.e. can have member-specific privileges per class)
  - Care: when objects can extend beyond their class, dynamically added methods may be restricted only to public access (as in Action Script 3)
    - Such violations can be detected only at runtime

Class details(3/10)

- *Constructors*: an automation of a common object initialization pattern
  - Single with no arguments – very uncommon
  - Multiple overloaded versions - C++, Java, C#
  - Explicitly callable
    - Inside class - C++, Java, C#
    - Outside classes as a normal method - C++ yes, not in Java, C#
  - First-class value – not in C++, Java, C#

Class details(4/10)

- Supporting constructors is no different than supporting class methods, with some simple extra work
  - Same code generation as with methods (member functions) when treating `this` pointer
  - Constructor overloading is handled exactly like function or method overloading
  - Automatic base constructor invocation is addressed by emitting an extra instruction to explicitly invoke it
  - First-class constructors could be easy; not clear why languages with all the required machinery don't support them
    - e.g., in C++ once we accept that the return type of constructors is implicitly `void`, we may assign their address to pointers to member functions

Class details(5/10)

- *Destructors*: an automation of a common object disposal pattern
  - Reserved method that is called automatically upon object collection (manual or automatic)
    - In most cases invocation is statically bound (easy implementation) with the exception of virtual destructors
  - Automatic base (super) destructor invocation requires emitting an extra instruction to invoke base destructor
    - As we do for constructors
  - In garbage collected languages their presence is really degenerate
    - *Should never introduce destructors (finalizers) in garbage collected languages (do you agree?)*

Class details(6/10)

- *Properties / attributes*: an automation of a common field access pattern

```
public class A {
        private T y;
        public T x { get { return y; } set { y = value; } }
        public T x { get; set; }
};
A a = new A(); a.x = <expression of T type>;          // C#
```

```
Class A {
        private var y:T;
        public function get x():T { return y; }
        public function set x(v:T): void { y=v; }
}
var a:A = new A(); a.x = <expression of T type>;      // AS3
```


Class details(7/10)

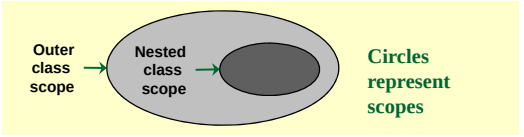
- *Properties / attributes*: implementation
  - Language syntax is extended with the forms of the pattern (optional `set` / `get`, auto complete)
  - Code generation is managed by emitting anonymous private `set` and `get` class methods (or use hidden names, like `$set_x` and `$get_x`)
  - Auto complete case requires code generation as if the user has explicitly supplied the methods
    - Can be handled by handing code over the AST or appending on the intermediate byte code
    - Since no source code is present no step-in tracing is applied during debugging

Class details(8/10)

- *Properties / attributes*: `set` overloading
  - Concerns cases where an attribute of type *T* is assigned from alternative data types
  - Requires support for multiple signatures, handled statically via overloading (easy)
  - Supporting multiple arguments means the language defines if priority is given on multi-value assignments over expression lists and that no ambiguities arise
    - Assume *x* where *set* $(T_1\ a, T_2\ b)\ \{\ x = priv\ \ f(a,b)\ \}$
    - $x = expr_1, expr_2$ ≡ $x.set(expr_1, expr_2)$ *priority*
    - $f\ (x = expr_1, expr_2)$ ≡? $f(x.set(expr_1, expr_2))$ *ambiguity*
    - $f\ (x = expr_1, expr_2)$ ≡? $f(x.set(expr_1), expr_2)$ *ambiguity*

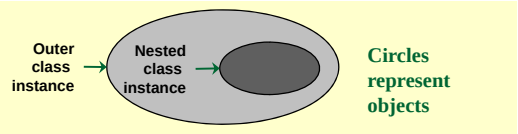
Class details(9/10)

- *Nested classes*: classes inside other classes
  - *Syntactic* nesting
    - The nested class just gains the namespace of the owner (outer) class, being semantically as any other class - C++, Java (static nested classes), C#
    - No extra privilege of the nested class in accessing the owner is involved
    - The implementation requires some extra naming and scoping



Class details(10/10)

- *Nested classes*: classes inside other classes
  - *Semantic* nesting
    - Nested class hidden outside the owner class, while an instance of nested class can exist only in an instance of an owner class and has access to the members of its enclosing instance - Java (inner classes), I-*GET* language (child agent classes)
    - When an instance of the owner class is destroyed all instances of the nested class it created are auto collected
      - Java (by garbage collection), I-*GET* language (by agent destruction semantics)



Typing issues (1/4)

- The type system of a language provides the rules for assigning types to expressions and for ensuring that illegal type uses are all tractable
- When it comes to classes, since we did not yet introduce inheritance, the question is the following:
  - Let ***A a*** and ***B b*** for distinct classes ***A*** and ***B***
  - Provide the function ***Equivalent(A,B)***
    - *Type checker*
  - Provide the compilation for ***a=b***
    - *Code generator*

Typing issues (2/4)

- There are two dominant methods for type equivalence:
  - Nominal equivalence (nominative type system)
    - $A \equiv B \Leftrightarrow A=B$ *or* $A \equiv alias(B)$ *or* $B \equiv alias(A)$
    - *Rigid type system (C, C++, Java, Eifel, C#)*
  - Structural equivalence (structural type system)
    - *Let* $A \subseteq B \Leftrightarrow \forall a \in A \; \exists b \in B : T(a) \equiv T(b)$
    - *Then* $A \equiv B \Leftrightarrow A \subseteq B \wedge A \supseteq A \wedge A.total = B.total$
    - That is, for each member of a class there must be a corresponding and identical member in the other class, and the two classes should have the same number of members

Typing issues (3/4)

- Questions about structural equivalence
  - Should the equivalent members have same names?
    - If names do not match, to guarantee correct use in substitution *copy by value* semantics maybe forced for arguments
    - Or else, temporary member aliases compatible to the original type parameters need to be introduced
      - This may affect macroscopically your language implementation since with every call we need:
        - to establish member aliases for use in the invoked method
        - as an object may be further used as an argument, stacks of such alias lists are needed
        - upon call return we should pop the alias list
      - *Performance issues may well reduce your appetite for such fancy features*

Typing issues (4/4)

- Questions about structural equivalence
  - Should we involve both fields and methods in the equivalence test or just the methods?
    - Method-based equivalence will work only when the fields are exclusively used inside methods by invocations to accessor methods (attributes)
  - Should the equivalent members be ordered identically?
    - i.e., class treated as a *tuple* (ordered list), not as a *set*
    - If classes are treated as sets, then the code generation should ensure that field access at runtime is independent to their order of appearance (like Java symbolic names)

Dynamic classes (1/5)

- Also called first-class classes, i.e. classes that can be the used as normal values
  - Consequently, a dynamic class can be the outcome of a computation (or expression evaluation)
  - We have seen one way of implementing dynamic classes, but without typing support by the language
    - Reflection mechanisms, essentially libraries
  - But now we will study them in the perspective of genuine language support (i.e. no library)

Dynamic classes (2/5)

```
; Normal class definition in Lisp, foo←bar.
; Syntax is (defclass id ( [bases] ) ( [slots] ))
  (defclass bar () ())
  (defclass foo (bar) ())
  (defclass baz () ())

; Using LISP MOP (Meta Object Protocol) to change  classes dynamically.
; The > is interpreter prompt, upper case messages are its responses.
  > (class-direct-superclasses (find-class 'foo))
  (#<STANDARD-CLASS BAR>)
  > (ensure-class 'foo :direct-superclasses '(bar
  baz))
  #<STANDARD-CLASS FOO>
  > (class-direct-superclasses (find-class 'foo))
  (#<STANDARD-CLASS BAR> #<STANDARD-CLASS BAZ>)
```

- We will observe that most cases where dynamic classes are allowed are in interpreted languages.
- There are compiled versions of Lisp too.
- MOP is a sort of an object-system protocol description through which the original program can be modified (like write-mode self reflection).

Dynamic classes (3/5)

```
> (defclass ball ()
    ((%radius :initform 10 :accessor radius)))
#<STANDARD-CLASS BALL>
> (defclass tennis-ball (ball) ())
#<STANDARD-CLASS TENNIS-BALL>
---
> (defvar *my-ball* (make-instance 'tennis-ball))
*MY-BALL*
> (radius *my-ball*)
10
---
> (defclass ball ()
    ((%radius :initform 10 :accessor radius)
     (%volume :initform (* 4/3 pi 1e3)
              :accessor volume)))
#<STANDARD-CLASS BALL>
---
> (volume *my-ball*)
4188.790204786391d0
```

- Redefining a class at runtime will cause modifications to propagate immediately among instances and subclasses of the class.
- For example in the example on the left we redefine **ball** to have an extra slot (attribute) with an initial value.
- If we check we realized **my-ball** has also the extra attribute.

Dynamic classes (4/5)

- In general the problem with dynamic classes is that to have absolute power we should:
  - Sacrifice static type safety and make everything apply at runtime
    - Thus type faults may appear as runtime errors, which is not always desirable
    - Or type-checking client code is needed, which is not always performance-wise acceptable
    - *Overall, such dynamic languages will come up with programming directives about type-specific unit testing*
  - Support structural equivalence, since expressions may return anonymous classes
    - Or else we will be bound to nominal typing of objects from the names of the methods that return dynamic classes and the types classes supplied as arguments (a dynamic analogy to C++ templates being static class generators)

Dynamic classes (5/5)

- Let `Class` be the type of a class value. Then a `Class` value is an actual class. Assume structural equivalence requiring matching of member names.
- Assume we have the necessary language operators to freely edit (add / remove / query) the fields and methods of a class value.
- Assume a `Class` value `A`. Then `new A()` will create an instance of the respective class.
- Let `Class` *F*`(Class` C_1`,Class` C_2`)` be a class generator function accepting two class arguments while returning a class value.
- Then if $A_1 \equiv A_2$ and $B_1 \equiv B_2$ we define $F(A_1,B_1) \equiv F(A_2,B_2)$.
- This is analogous to the case $F<A_1,B_1> \equiv F<A_2,B_2>$ applied for generators.

Possible semantics to support statically-typed dynamic classes with generator functions resembling templates but enabling class generation algorithmically.

The typing consistency is based on the hypothesis that generators return equivalent values for equivalent arguments.