Classes

- Components
  - fields/instance variables
    - values may differ from object to object
    - usually mutable
  - methods
    - values shared by all objects of a class
    - usually immutable
  - component visibility: public/private/protected

Inheritance

- Three traditional components of object-oriented languages
  - abstraction/encapsulation
  - subtypes/interface inheritance: interfaces inherit method signatures from super-interfaces
  - subclassing/implement inheritance: a class inherits signatures and code from a superclass (possibly "abstract")

Code Generation for Objects

- Methods
  - Generating method code
  - Generating method calls (dispatching)
  - Constructors and destructors

- Fields
  - Memory layout
  - Generating code to access fields
  - Field alignment

Compiling Methods

- Methods look like functions, are type-checked like functions...what is different?

- Argument list: implicit receiver argument

- Calling sequence: use dispatch vector instead of jumping to absolute address

The Need for Dispatching

- Example:
  ```java
  interface Point {
    int x(); int y(); float norm(); }
  class ColoredPoint implements Point {
    float norm() { return sqrt(x*x+y*y); }
  }
  class 3DPoint implements Point {
    float norm() { return sqrt(x*x+y*y+z*z); }
  }
  Point p;
  if (true) p = new ColoredPoint();
  else       p = new 3DPoint();
  int n = p.norm();
  ```

- Compiler can't tell what code to run when method is called!
**Dynamic Dispatch**

- Solution: dispatch vector (dispatch table, selector table...)
  - Entries in the table are pointers to method code
  - Pointers are computed dynamically!
  - If \( T < S \), then vector for objects of type \( S \) is a prefix of vector for objects of type \( T \)

<table>
<thead>
<tr>
<th>object reference</th>
<th>object layout</th>
<th>dispatch vector</th>
<th>method code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td></td>
<td>getx</td>
<td>gety</td>
</tr>
<tr>
<td></td>
<td></td>
<td>norm =</td>
<td>norm =</td>
</tr>
</tbody>
</table>

**Why It Works**

- If \( S < T \) and \( f \) is a method of an object of type \( T \), then
  - Objects of type \( S \) inherit \( f \);
  - Pointer to \( f \) is same index in the DV for type \( T \) and \( S \)
- Statically generate code to look up pointer to method \( f \)
- Pointer values determined dynamically

<table>
<thead>
<tr>
<th>Point reference</th>
<th>3DPoint layout</th>
<th>3DPoint vector</th>
<th>3DPoint code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td></td>
<td>getx</td>
<td>gety</td>
</tr>
<tr>
<td>( y )</td>
<td></td>
<td>norm =</td>
<td>norm =</td>
</tr>
</tbody>
</table>

**Dispatch Vector Lookup**

- Every method has its own small integer index
- Index is used to look up method in dispatch vector

\[
\begin{align*}
C &< B < A \\
A & f \\
B & f,g,h \\
C & f,g,h,e
\end{align*}
\]

**Dispatch Vector Layouts**

- Index of \( f \) is the same in any object of type \( T < A \)
- Methods may have multiple implementations
  - For subclasses with unrelated types
  - If subclass overides method
- To execute a method:
  - Lookup entry in vector
  - Execute code pointed to by entry value

\[
\begin{align*}
C &< B < A \\
A & f \quad 0 \\
B & g \quad 1 \\
C & h \quad 2
\end{align*}
\]

**Code Generation: Dispatch Vectors**

- Allocate one dispatch vector per class
  - Objects of same class execute same method code
- Statically allocate dispatch vectors

```
data LenListDV LongLenListFirst LongLenListRest LongLenListLength
```

**Interfaces, Abstract Classes**

- Classes define a type and some values (methods)
- Interfaces are pure object types: no implementation
  - no dispatch vector; only a DV layout
- Abstract classes are halfway:
  - define some methods
  - leave others unimplemented
  - no objects (instances) of abstract class
- DV needed only for concrete classes
**Static Methods**
- In Java, can declare methods static — they have no receiver object
- Called exactly like normal functions — don’t need to enter into dispatch vector — don’t need implicit extra argument for receiver
- Treated as methods as way of getting functions inside the class scope (access to module internals or semantic analysis)
- Not really methods

**Method Arguments**
- Methods have a special variable (Java, C++: this) called the receiver object
- Historically (Smalltalk): method calls thought of as messages sent to receivers
- Receiver object is (implicit) argument to method

```java
class A {
  int(int x, int y)
  {...}
}
```

**Code Generation: Method Calls**
- Code for function calls: pre-call + post-call code
- Pre-function-call code:
  - Save registers
  - Push parameters
- Pre-method call:
  - Save registers
  - Push registers
  - Push receiver object reference
  - Lookup method in dispatch vector

**Example**
0.foo(2,3);

```
push $3
push $2
push %eax
mov (%eax), %ebx
mov 4(%ebx), %ecx
call %ecx
add $12, %esp
```

**Object Layout**
- Object consists of:
  - Methods
  - Fields
- Object layout consists of:
  - Pointer to DV, which contains pointers to methods
  - Fields

```
layout (static data) (code)
DV x y
getx gety
```

**Allocation of Objects**
- Objects can be stack- or heap-allocated
- Stack allocation:
  - (C++) Point p;
  - (Java)

```
Point p = new Point();
```
- Heap:
  - (C++)
  - Point p = new Point;
  - (Java)

```
p = new Point();
```
Inheritance and Object Layout

- Method code copied down from superclass if not overridden by subclass
- Fields also inherited (needed by inherited code in general)
- Inheritance: add fields, methods
  - Extend layout
  - Extend dispatch vector
  - A supertype object can be used whenever a subtype object can be used

Field Offsets

- Offsets of fields from beginning of object known statically, same for all subclasses
- Example:
  ```java
class Shape {
    Point LL, UR;
    void setCorner(int which, Point p);
}
class ColoredRect extends Shape {
  int color;
  void setColor(int col);
}
```
- Offsets known for stack and heap allocated objects

Accessing Fields

- Access fields of current object
  - Access x equivalent to this.x
  - Current method has "this" as argument
- Access fields of other objects
  - Access of the form o.x
- In both cases:
  - Use pointer to object
  - Add offset to the field
- Access o.x depends on the kind of allocation of o
  - Stack allocation: stack access ("memcpy + stack offset")
  - Heap allocation: stack access + dereference

Field Alignment

- In many processors, a 32-bit load must be to an address divisible by 4
- In x86 (e.g., Pentium), loads are 16-bit if aligned — avoids extra load
  ⇒ Fields should be aligned

```c
struct {
  int x; char c; int y; char d;
  int z; double e;
}
```
Code Generation: Allocation

- Heap allocation: o = new C;
  - Allocate heap space for object
  - Store pointer to dispatch vector

- Stack allocation:
  - Push object on stack
  - Pointer to DV on stack

Compiling Constructors

- Compiled similarly with methods:
  - pseudo-variable "this" passed to constructor
  - return value is "this"

Destructors

- In some languages (e.g. C++), objects can also declare code to execute when objects are destructed

Analysis and Optimizations

- Dataflow analysis reasons about variables and values
- Records (objects) consist of a collection of variables (fields) - analysis must separately keep track of individual fields

- Difficult analysis for heap-allocated objects
  - Object lifetime outlives procedure lifetime
  - Need to perform inter-procedural analysis

- Constructors/destructors: must take into account the effects

Class Hierarchy Analysis

- Method calls = dynamic, via dispatch vectors
  - Overhead or going through DV
  - Prohibits function inlining
  - Makes other inter-procedural analyses less precise

- Static analysis of dynamic method calls
  - Determine possible methods invoked at each call site
  - Need to determine principal types or objects at each program point (Class Hierarchy Analysis)

  - If analysis determines object o is always of type T (not subtype), then it precisely knows the code for o.foo()
Summary

- Method dispatch accomplished using dispatch vector, implicit method receiver argument
- No dispatch of static methods needed
- Inheritance achieves extension of fields as well as methods; code can be shared
- Field alignment: declaration order matters!
- Each real class has a single dispatch vector in data segment: installed at object creation or constructor
- Analysis more difficult in the presence of objects
- Class hierarchy analysis = precisely determine object class