Class Components

- fields/instance variables
  - values may differ from object to object
  - usually mutable
- methods
  - values shared by all objects of a class
  - usually immutable
  - usually functions with implicit argument
    - object itself (this/self)
- all components have visibility
  - e.g. public, private, protected

Code generation for objects

- Methods
  - Generating method body
  - Generating method calls (dispatching)
- Fields
  - Memory layout
    - Packing and alignment
  - Generating accessor code

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

- Problem: compiler can’t tell what code to run when method is called

abstract class Point { int getx(); float norm(); }
class CartesianPoint implements Point { ...
  float norm() { return sqrt(x*x+y*y); }
}
class RadialPoint implements Point { ...
  float norm() { return r; }
  float dist(Point pt) { return pt.norm(); }
}

- Solution: dispatch table (dispatch vector, selector table...)

Method dispatch

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

abstract class A { void foo(); }
abstract class B extends A { void bar(); void baz(); }
class C implements B { void foo() [...]
  void bar() [...]
  void baz() [...]
  void quux() [...] }
Method arguments

- Methods have a special variable (in Java, “this”) called the receiver object or context object
- Historically (Smalltalk): method calls thought of as messages sent to receivers
- Receiver object is (implicit) argument to method

```java
class Shape {
    int setCorner(int which, Point p) { ... }
}
```

```java
int setCorner(Shape this, int which, Point p) { ... }
```

Calling sequence

Function

```
f(...) 
```

```java
CALL NAME(f) ...
```

Name of function

```
e . baz(...) 
```

E SEQ

```
MOVE t0 
```

E[ e ]

```
MEM t0, ... 
```

(baz's index) * 4

```
MEM t0 
```

Example

```
A foo
B bar, baz
C quux
```

```
b . bar(3) ;
```

```
push 3
push eax
mov ebx, [eax]
mov ecx, [ebx + 4] (bar's index = 1)
call ecx
```

Inheritance

Three traditional components of object-oriented languages

- abstraction/encapsulation/information hiding
- subtyping/interface inheritance -- interfaces inherit method signatures from supertypes
- inheritance/implementation inheritance -- a class inherits signatures and code from a superclass (possibly “abstract”)
Object Layout

class Shape {
    Point LL, UR;
    void setCorner(int which, Point p);
}
class ColoredRect extends Shape {
    Color c;
    void setColor(Color c_);
}

Code Sharing

Machine code for Shape.setCorner

• Don’t actually have to copy code!
• Works with separate compilation: can inherit without superclass source

Interaces, abstract classes

• Classes define a type **and some values** (methods)
• Interfaces are pure object types: no implementation
  – no V-Table: only an IM-Table layout
• Abstract classes are halfway:
  – define some methods
  – leave others unimplemented
  – no objects (instances) of abstract class
• V-Table only for (abstract) 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 for semantic analysis)
• Not really methods

Constructors

• Java, C++: classes can declare **object constructors** that create new objects:
class C { public C(x, y, z) { initialize C } ...}
• Scala, CubeX: one constructor
class C(x,y,z) { initialize C in body }

Compiling constructors

• Compiled just like static methods except:
  – pseudo-variable “this” is in scope as in methods
  – this is initialized with newly allocated memory
  – first word in memory initialized to point to v-table
  – value of this is return value of code
• For CubeX
  – Where “new C” is called
    • allocate memory for C instance
    • set first word of instance to point to C’s v-table
    • call C’s constructor passing the pointer
  – Inside C’s constructor
    • initialize fields of C using initialization statements
    • use super’s constructor to initialize super’s fields