Method Resolution Approaches

- Static - procedural languages (w/o fcn ptrs)
- Dynamically determined by data values
  - C with function pointers
  - Compile-time analysis can estimate possible callees
- Dynamically determined by receiver type
  - Some polymorphic OOPLs use run-time type of first parameter to specialize behaviors
  - Some OOPLs also use run-time types of other arguments
  - Challenge: how to have an efficient implementation of this kind of dynamic dispatch?

Dynamic Dispatch

- Choices PLs have to make:
  - When resolve function targets?
  - What to look at to do the method resolution? (e.g., receiver run-time type? Argument run-time types?)
  - How to divide the work between runtime and compile time, so as to minimize overhead for virtual calls?
  - Emphasize flexibility or performance?
Method Redefinition

- **Overriding** - replacing a superclass’s implementation of a method, by one with identical signature (except receiver type)
  - Method must be accessible, non-static
- **Overloading** - providing more than one method with same name, but different signatures to distinguish them
- Simple cases of both are intuitive

Inheritance

- Overriding can widen method visibility
  - `void f() {super f();}` - where `f` protected in superclass
- Can override instance variables, but can still get to superclass variable using `super`
- Preferred inheritance uses all private data and provides observer and mutator methods
  - Using `geta()`, `seta()` methods means that changing superclass structure will not affect subclasses
- Access to
  - Methods is by run-time type of object referenced
  - Instance variables is by compile-time type of reference
Possible Cases

- **Inheritance**, but all method names are unique
- **Inheritance with overriding**
  - Lookup happens at runtime based only on receiver’s class
  - Next slide, eg1: A, B, C with respect to f(); A, B wrt h()
- **Inheritance with overloading** (different method signatures)
  - Java: Lookup establishes best match type signature at compile-time based on arguments’ and receiver’s declared classes; actual binding done by run-time lookup to match selection
  - Next slide, eg2, A, B wrt s()

Overriding Example

```java
A a = new B();
```

How to resolve `a.toString()`?
1. **At runtime**, determine class of the object (e.g., `a` refers to an B object).
2. Start lookup for method with same signature in class B.
3. Proceed up inheritance hierarchy until find closest superclass with same signature method (i.e., method `toString()`); this may be class B itself
Java Overriding Example 1

//overriding - fcns have
//same signature
A a1 = new B();
A a2 = new C();
B b = new B();
A a = new A();
a.f(); // A's f()
a1.f(); // B's f()
a2.f(); // C's f()
b.h(0); // B's h()
a1.h(2); // B's h()
a2.h(1); // A's h()
a.h(3); // A's h()

C++ Approach to Overriding

- If return type and signature of 2 functions match exactly,
the 2nd is a re-declaration of the first and is an override
- If signatures of 2 functions match exactly, but return types
differ, then 2nd declaration is in error
- If signatures differ in number or type of arguments, the 2
function instances are OVERLOADED. (return type not
considered as part of signature here)

S. Lippman,
C++ Primer
Overloading

- Java chooses to optimize dynamic dispatch by partially resolving references through preprocessing at compile-time
  - Need to use declared type and number of arguments + receiver type to help select an unique method
- Results in a not-just-dynamic lookup procedure because pre-selection is done
- Different from multi-methods (e.g., in Cecil) where dynamic lookup is based totally on run-time types of receiver and the arguments!

Example - unexpected results

```java
//overloading -when signatures //not same, must look at type //matching between arg and //param
a.s(3.); //A's s()
a1.s(3.); //A's s() because //arg is a double and B's //s() expects an int
b.s(0); //B's s()
b.s(1.0); //A's s() //casting is not type //conversion in Java
((A) a1).h(4); //uses B's h() //matching rules are not //always straight-forward
a1.s(0); //A's s()
```

```
A a1 = new B();
A a2 = new C();
B b = new B();
A a = new A();
```
Overloading Resolution in Java

• At compile-time, assemble a set of methods whose parameters are type compatible with the arguments and receiver

• For each invocation
  • Look at compile-time class of receiver and arguments
  • Move up class hierarchy from declared receiver type class trying for a match (possibly widening argument or receiver types)
  • Collect all possible matching methods into a set and then find the most specific match (defined on next slide)

Most Specific Match

• If find unique method with exact match in type and number of arguments and compatible receiver type, choose it.

• Otherwise,
  • If any method f has arguments + receiver that can be assigned to any other method g in the set, discard g; Repeat as much as possible.
  • If only 1 method remains, use it as template.
  • If more than 1 method remains, the invocation is ambiguous, so the invoking code is invalid. Compile-time error!!
Overloading Resolution in Java
Run-time Overriding

• At run-time, use run-time type of receiver to start search up class hierarchy for function exactly matching previously defined template. (Note: ignore run-time types of arguments)
• Stop going up the hierarchy when find first match to template type. Overloading guarantees there will be at least one match.

Java Example (cf Don Smith, Rutgers)

• Class hierarchy as shown contains 4 variants of method f()
• Signatures[...] include compile-time types of receiver and argument.
• Objects named for their compile-time type
Java Example

• a.f(x)
  - signature \( f: [A, X] \)
  - check classes A
  - matches \( f: [A, X] \)

• s.f(a)
  - signature \( f: [S, A] \)
  - check classes S, R, A
  - matches \( f: [R, A] \)

A: f(X) \( f: [A, X] \)

X

R: f(A) \( f: [R, A] \)

Y

S: f(X) \( f: [S, X] \)

T: f(Y) \( f: [T, Y] \)

Java Example

• s.f(y)
  - signature \( f: [S, Y] \)
  - checks S, R, A
  - matches \([S, X], [R, A], [A, X]\).

• check pairwise for most specific
  \([S, X]\) with \([R, A]\)
  \([S, X]\) with \([A, X]\)
  \([R, A]\) with \([A, X]\)

\([S, X]\) is choice-most specific

If any method \( f \) has arguments + receiver that can be assigned to any other method \( g \) in the set, discard \( g \):
Java Example

- `r.f(x)`
  - signature `f:[R, X]`
  - checks R, A
  - matches [R, A], [A, X]

- check pair
  - R << A but A >> X
  - incomparable
  - no match
  - compile-time ERROR!

Java Example

- `t.f(y)`
  - signature `f:[T, Y]`
  - checks T, S, R, A
  - matches [T, Y], [S, X], [R, A], [A, X]

- pairwise check and get [T, Y]
Java Example

```
T t = new T();
X x = new Y();
...
t.f(x);
signature is [T, X]
checks T, S, R, A
matches [S, X], [R, A], [A, X]
specificity eliminates all but [S, X]
at run-time receiver is class T and
argument is class Y.
However, call will be resolved to f(X)
in class S and not to f(Y) in class T,
even though t.f(Y) is a perfect
match to these types at runtime!
```

Java Example - 2

```
X x = new X();
Y y = new Y();
X xy = (X) y;
x.f(x) -- invokes X.f(X)
x.f(y) -- invokes X.f(Y) since more specific than X.f(X)
x.f(xy) -- invokes X.f(X)
y.f(x) -- invokes Y.f(X), since more specific than X.f(X)
xy.f(x) -- invokes Y.f(X) which overrides X.f(X) for Y
receivers.
Key point is only receiver's run-time type is looked at for
method resolution to the pre-computed template.
```