Programming Language Concepts: Lecture 3

Madhavan Mukund

Chennai Mathematical Institute

madhavan@cmi.ac.in

http://www.cmi.ac.in/~madhavan/courses/pl2009

PLC 2009, Lecture 3, 21 January 2009

Subclasses

- A class `Employee` for employee data

```
class Employee{
  private String name;
  private double salary;

  // Some Constructors ...

  // "mutator" methods
  public boolean setName(String s){ ... }
  public boolean setSalary(double x){ ... }

  // "accessor" methods
  public String getName(){ ... }
  public double getSalary(){ ... }

  // other methods
  double bonus(float percent){
     return (percent/100.0)*salary;
  }
```

Subclasses

- Managers are special types of employees with extra features

```
class Manager extends Employee{
      private String secretary;
      public boolean setSecretary(name s){ ... }
      public String getSecretary(){ ... }
}
```

- `Manager` objects inherit other fields and methods from `Employee`
  - Every `Manager` has a `name`, `salary` and methods to access and manipulate these.
- `Manager` is a subclass of `Employee`
  - Think of subset

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
- Common to extend a parent class written by someone else

Subclasses

- Can use parent class's constructor using `super`

```
class Employee{
  ...
  public Employee(String n, double s){
     name = n; salary = s;
  }
  public Employee(String n){
     this(n,500.00);
  }
}
```

- In `Manager`

```
public Manager(String n, double s, String sn){
   super(n,s);   /* super calls
                    Employee constructor */
   secretary = sn;
}
```

Subclasses

- Subclass can override methods of super class

```
double bonus(float percent){
   return 1.5*super.bonus(percent);
}
```

- In general, subclass has more features than parent class
- Can use a subclass in place of a superclass

```
Employee e = new Manager(...)
```

- Every `Manager` is an `Employee`, but not vice versa!
- Recall
  - `int[] a = new int[100];`
  - Aside: Why the seemingly redundant reference to `int` in `new`?
- One can now presumably write

```
Employee[] e = new Manager(...)[100]
```

Subclasses

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary`?

Subclasses

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`

Subclasses

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`
- What about `e.bonus(p)`? Which `bonus` do we use?
  - Static: Use `Employee.bonus`
  - Dynamic: Use `Manager.bonus`

Subclasses

```
Employee e = new Manager(...)
```

- Can we invoke `e.setSecretary`?
  - `e` is declared to be an `Employee`
  - Static typechecking — `e` can only refer to methods in `Employee`
- What about `e.bonus(p)`? Which `bonus` do we use?
  - Static: Use `Employee.bonus`
  - Dynamic: Use `Manager.bonus`
- Dynamic dispatch (dynamic binding, late method binding, ...) turns out to be more useful
  - Default in Java, optional in C++ (use `virtual`)

Dynamic dispatch

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

Dynamic dispatch

```
Employee[] emparray = new Employee[2];
Employee e = new Employee(...);
Manager e = new Manager(...);

emparray[0] = e;
emparray[1] = m;

for (i = 0; i < emparray.length; i++){
  System.out.println(emparray[i].bonus(5.0);
}
```

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Also referred to as runtime polymorphism or inheritance polymorphism

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types
- Can have different functions with the same name and different signatures
  - For example, multiple constructors

Functions, signatures and overloading . . .

- Java class `Arrays`: method `sort` to sort arbitrary scalar arrays

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);  // sorts contents of darr
Arrays.sort(iarr);  // sorts contents of iarr
```

Functions, signatures and overloading . . .

- Java class `Arrays`: method `sort` to sort arbitrary scalar arrays

```
double[] darr = new double[100];
int[] iarr = new int[500];
...
Arrays.sort(darr);  // sorts contents of darr
Arrays.sort(iarr);  // sorts contents of iarr
```

- Methods defined in class `Arrays`

```
class Arrays{
  ...
  public static void sort(double[] a){..}
     // sorts arrays of double[]
  public static void sort(int[] a){..}
     // sorts arrays of int[]
  ...
}
```

Functions, signatures and overloading . . .

- Overloading: multiple methods, different signatures, choice is static
- Overriding: multiple methods, same signature, choice is static
  - `Employee.bonus`
  - `Manager.bonus`
- Dynamic dispatch: multiple methods, same signature, choice made at run-time

Inheritance

```
Employee e = new Manager(...)
```

- Can we force `e.setSecretary` to work?

Inheritance

```
Employee e = new Manager(...)
```

- Can we force `e.setSecretary` to work?
- Type casting

```
((Manager) e).setSecretary(s)
```

Inheritance

```
Employee e = new Manager(...)
```

- Can we force `e.setSecretary` to work?
- Type casting

```
((Manager) e).setSecretary(s)
```

- Cast fails (error) if `e` is not a `Manager`

Inheritance

```
Employee e = new Manager(...)
```

- Can we force `e.setSecretary` to work?
- Type casting

```
((Manager) e).setSecretary(s)
```

- Cast fails (error) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

Inheritance

```
Employee e = new Manager(...)
```

- Can we force `e.setSecretary` to work?
- Type casting

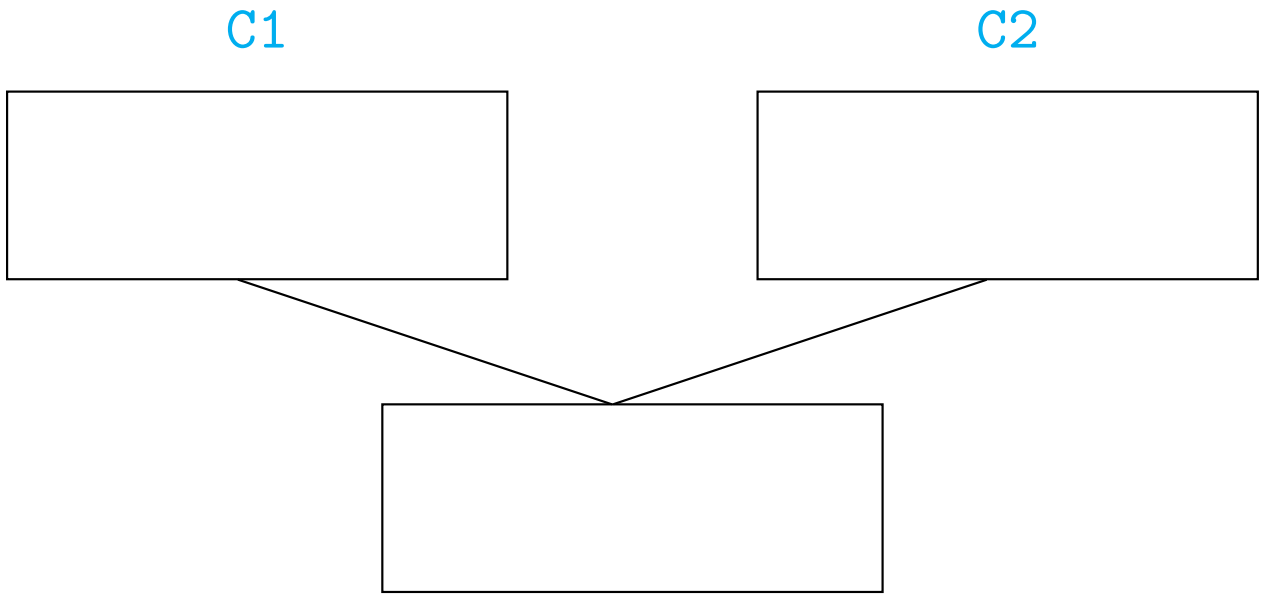
```
((Manager) e).setSecretary(s)
```

- Cast fails (error) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

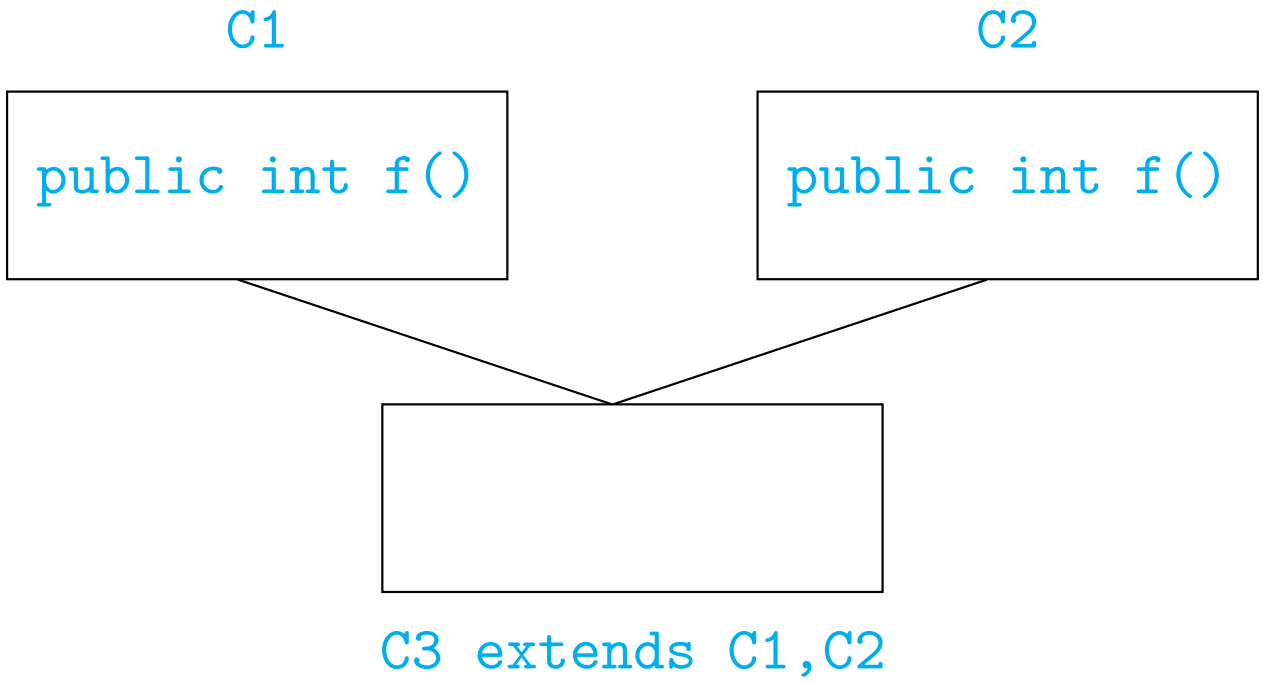
```
if (e instanceof Manager){
  ((Manager) e).setSecretary(s);
}
```

- Reflection — "think about oneself"

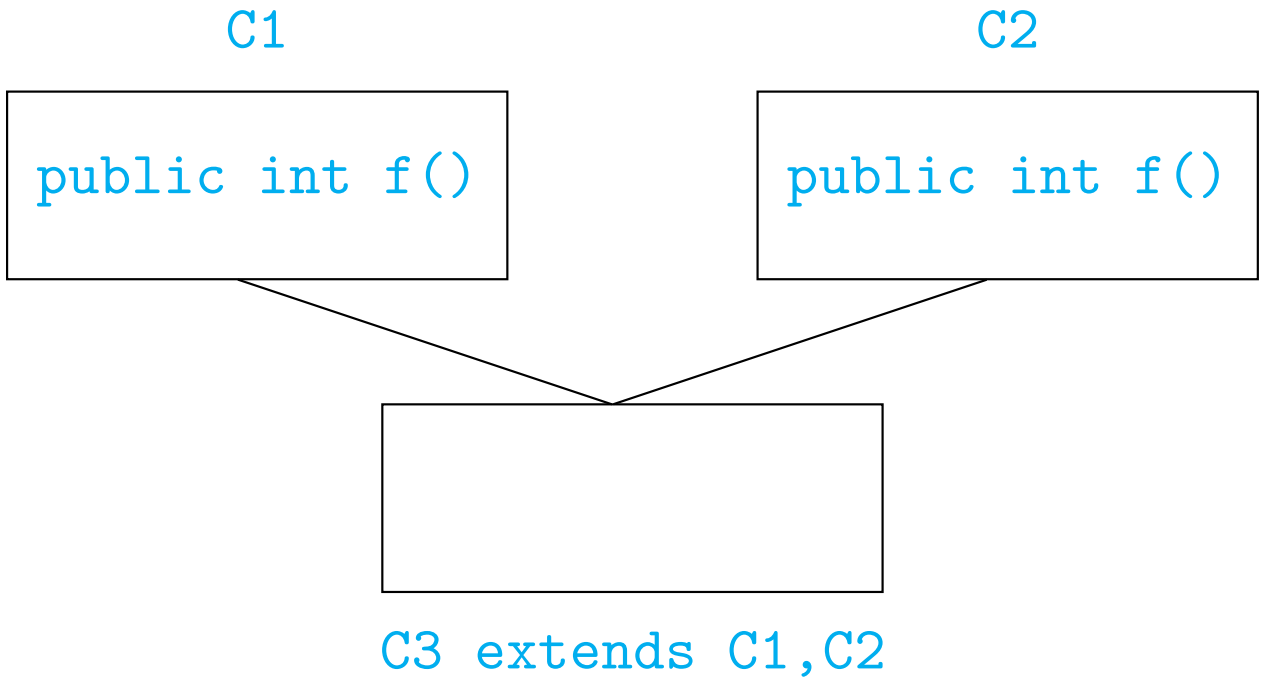
Multiple inheritance



Multiple inheritance



Multiple inheritance



- Which `f` do we use in `C3` (assuming `f` is not redefined)?
  - Java does not allow multiple inheritance
  - C++ allows this if `C1` and `C2` have no conflict

Java class hierarchy

- No multiple inheritance — tree-like
- In fact, there is a universal superclass `Object`
- Useful methods defined in `Object`

```
boolean equals(Object o)  // defaults to pointer equality

String toString()         // converts the values of the
                          // instance variable to String
```

- To print `o`, use `System.out.println(o+"");`

Java class hierarchy

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

Java class hierarchy

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

- Recall that == is pointer equality

Java class hierarchy

```
public int find (Object[] objarr, Object o){
  int i;
  for (i = 0; i < objarr.length(); i++){
      if (objarr[i] == o) {return i};
  }
  return (-1);
}
```

- Recall that == is pointer equality
- Redefine `equals`

```
boolean equals(Date d){
  return ((this.day == d.day) &&
          (this.month == d.month) &&
          (this.year == d.year));
}
```

Java class hierarchy

- `boolean equals(Date d)` does not override `boolean equals(Object o)`!

Java class hierarchy

- `boolean equals(Date d)` does not override `boolean equals(Object o)`!
- Should write

```
boolean equals(Object d){
  if (d instanceof Date){
    return ((this.day == d.day) &&
            (this.month == d.month) &&
            (this.year == d.year));
  }
  return(false);
}
```

Java class hierarchy

- Overriding looks for “closest” match

Java class hierarchy

- Overriding looks for "closest" match

  Suppose `boolean equals(Employee e)` but no `equals` in `Manager`

Java class hierarchy

- Overriding looks for "closest" match

  Suppose `boolean equals(Employee e)` but no `equals` in `Manager`

```
Manager m1 = new Manager(...);
Manager m2 = new Manager(...);
...
if (m1.equals(m2)){ ... }
```

Java class hierarchy

- Overriding looks for "closest" match

  Suppose `boolean equals(Employee e)` but no `equals` in `Manager`

  ```
  Manager m1 = new Manager(...);
  Manager m2 = new Manager(...);
  ...
  if (m1.equals(m2)){ ... }
  ```

  `boolean equals(Manager m)` compatible with both `boolean equals(Employee e)` and `boolean equals(Object o)`

Java class hierarchy

- Overriding looks for "closest" match

  Suppose `boolean equals(Employee e)` but no `equals` in `Manager`

  ```
  Manager m1 = new Manager(...);
  Manager m2 = new Manager(...);
  ...
  if (m1.equals(m2)){ ... }
  ```

  `boolean equals(Manager m)` compatible with both `boolean equals(Employee e)` and `boolean equals(Object o)`

  Use `boolean equals(Employee e)`