Java: class hierarchy, polymorphism, abstract classes

Madhavan Mukund, S P Suresh

Programming Language Concepts

Lecture 4, 23 January 2025

A Java class

- An `Employee` class

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

Subclasses

- Managers are special types of employees with extra features

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

class Square (Rectangle)

Subclasses

- Managers are special types of employees with extra features

```
public 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.

Subclasses

- Managers are special types of employees with extra features

```
public 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
- How can a constructor for `Manager` set instance variables that are private to `Employee`?

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?
- Some constructors for `Employee`

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

Subclasses

- `Manager` objects do not automatically have access to private data of parent class.
  - Common to extend a parent class written by someone else
- How can a constructor for `Manager` set instance variables that are private to `Employee`?
- Some constructors for `Employee`
- A constructor for `Manager`
- Use parent class's constructor using `super`

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

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

Inheritance

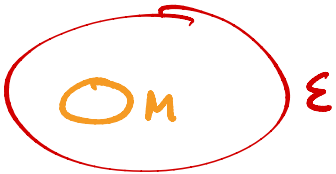
- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  `Employee e = new Manager(...)` ✓
- But the following will not work

  `Manager m = new Employee(...)` ✗



Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  ```
  Employee e = new Manager(...)
  ```
- But the following will not work

  ```
  Manager m = new Employee(...)
  ```
- Recall
  - `int[] a = new int[100];`
  - Why the seemingly redundant reference to `int` in `new`?

Inheritance

- In general, subclass has more features than parent class
  - Subclass inherits instance variables, methods from parent class
- Every `Manager` is an `Employee`, but not vice versa!
- Can use a subclass in place of a superclass

  `Employee e = new Manager(...)`
- But the following will not work

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

- Recall
  - `int[] a = new int[100];`
  - Why the seemingly redundant reference to `int` in `new`?
- One can now presumably write

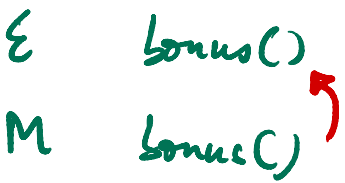
  `Employee[] e = new Manager[100];`

Dynamic dispatch

- `Manager` can redefine `bonus()`

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

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class



Dynamic dispatch

- `Manager` can redefine `bonus()`

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

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

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

- Can we invoke `e.setSecretary()`?

Manager m = --
m.bonus()
↑ Manager bonus()

e is actually a Manager

Legal?

Dynamic dispatch

- `Manager` can redefine `bonus()`

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

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
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`

Dynamic dispatch

- `Manager` can redefine `bonus()`

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

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
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

- `Manager` can redefine `bonus()`

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

  - Uses parent class `bonus()` via `super`
  - Overrides definition in parent class

- Consider the following assignment

```
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 languages like C++ (`virtual` function)

Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!

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

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

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

Polymorphism

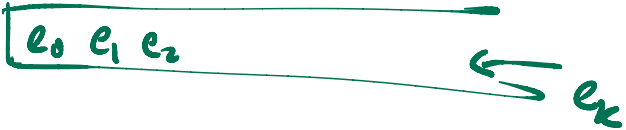
- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Object oriented programming originated in Simula — event simulation loop

Simulation

```
Q := make-queue(first event)
repeat
  remove next event e from Q
  simulate e
  place all events generated
     by e on Q
until Q is empty
```

dynamic dispatch

Event queue



Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Object oriented programming originated in Simula — event simulation loop
- Also referred to as runtime polymorphism or inheritance polymorphism

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

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

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

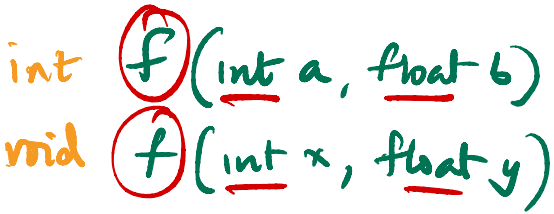
Polymorphism

- Every `Employee` in `emparray` "knows" how to calculate its `bonus` correctly!
- Object oriented programming originated in Simula — event simulation loop
- Also referred to as runtime polymorphism or inheritance polymorphism
- Different from structural polymorphism of Haskell etc — called generics in Java

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

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

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

Functions, signatures and overloading

- Signature of a function is its name and the list of argument types



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

- 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
- Java class `Arrays` has a 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

- 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
- Java class `Arrays` has a method `sort` to sort arbitrary scalar arrays
- Made possible by overloaded methods defined in class `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


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

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


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()`

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


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

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


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

Type casting

- Consider the following assignment

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

Type casting

- Consider the following assignment

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

- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`

(t) v

⇒ "casts" variable v to type t

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`

- Can test if `e` is a `Manager`

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

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`
- Can test if `e` is a `Manager`

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

- A simple example of reflection in Java
  - "Think about oneself"

Type casting

- Consider the following assignment
  `Employee e = new Manager(...)`
- Can we get `e.setSecretary()` to work?
  - Static type-checking disallows this
- Type casting — convert `e` to `Manager`
  `((Manager) e).setSecretary(s)`
- Cast fails (error at run time) if `e` is not a `Manager`

- Can test if `e` is a `Manager`

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

- A simple example of reflection in Java
  - "Think about oneself"
- Can also use type casting for basic types

```
double d = 29.98;
long nd = (long) d;
```

From C

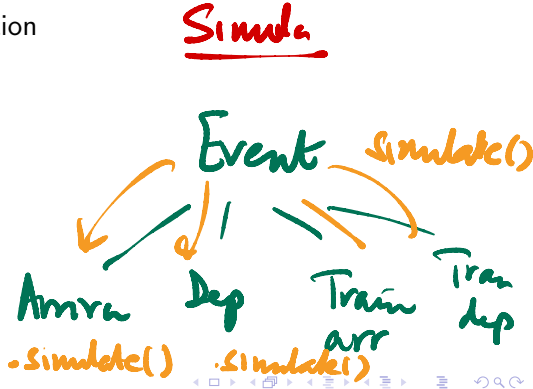
float f = 22/7;

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function
  `public double perimeter()`



Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function
  `public double perimeter()`
- Could define a function in `Shape` that returns an absurd value
  `public double perimeter() { return(-1.0); }`
- Rely on the subclass to redefine this function

Grouping together classes

- Sometimes we collect together classes under a common heading
- Classes `Circle`, `Square` and `Rectangle` are all shapes
- Create a class `Shape` so that `Circle`, `Square` and `Rectangle` extend `Shape`
- We want to force every `Shape` to define a function
  `public double perimeter()`
- Could define a function in `Shape` that returns an absurd value
  `public double perimeter() { return(-1.0); }`
- Rely on the subclass to redefine this function
- What if this doesn't happen?
  - Should not depend on programmer discipline

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

```
public abstract double perimeter();
```

Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

```
public abstract double perimeter();
```

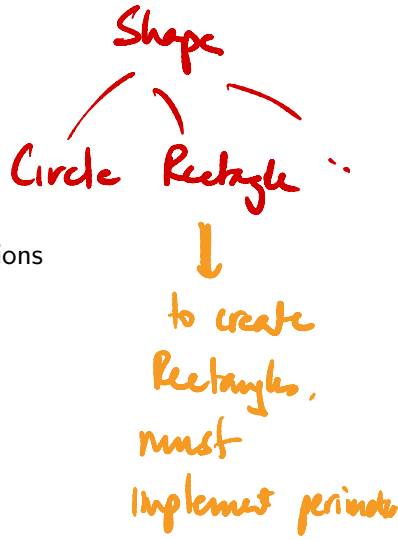
- Forces subclasses to provide a concrete implementation

Abstract classes

- A better solution
  - Provide an abstract definition in Shape

```
public abstract double perimeter();
```

- Forces subclasses to provide a concrete implementation
- Cannot create objects from a class that has abstract functions



Abstract classes

- A better solution
  - Provide an abstract definition in `Shape`

    ```
    public abstract double perimeter();
    ```
- Forces subclasses to provide a concrete implementation
- Cannot create objects from a class that has abstract functions
- `Shape` must itself be declared to be `abstract`

```
public abstract class Shape{
  ...
  public abstract double perimeter();
  ...
}
```

Abstract classes . . .

- Can still declare variables whose type is an abstract class

Abstract classes . . .

- Can still declare variables whose type is an abstract class

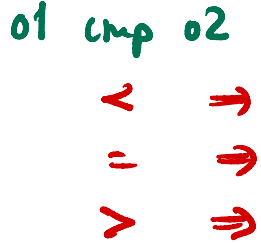
```
Shape shapearr[] = new Shape[3];
int sizearr[] = new int[3];

shapearr[0] = new Circle(...);
shapearr[1] = new Square(...);
shapearr[2] = new Rectangle(...);

for (i = 0; i < 3; i++){
  sizearr[i] = shapearr[i].perimeter();
     // each shapearr[i] calls the appropriate method
  ...
}
```

dynamic dispatch

Generic functions

- Use abstract classes to specify generic properties

```
public abstract class Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```



Generic functions

- Use abstract classes to specify generic properties

```
public abstract class Comparable{
  public abstract int cmp(Comparable s);
    // return -1 if this < s,
    //         0 if this == 0,
    //        +1 if this > s
}
```

Ord a ⇒ . .

- Now we can sort any array of objects that extend `Comparable`

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
    // Usual code for quicksort, except that
    // to compare a[i] and a[j] we use a[i].cmp(a[j])
  }
}
```

Generic functions . . .

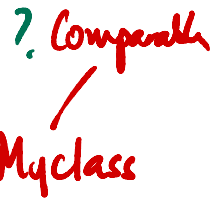
```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
  }
}
```

Generic functions . . .

```
public class SortFunctions{
  public static void quicksort(Comparable[] a){
    ...
  }
}
```

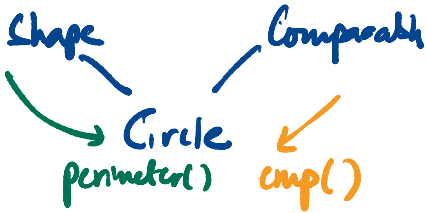
- To use this definition of `quicksort`, we write

```
public class Myclass extends Comparable{
  private double size;   // quantity used for comparison

  public int cmp(Comparable s){
    if (s instanceof Myclass){
      // compare this.size and ((Myclass) s).size
      // Note the cast to access s.size
    }
  }
}
```



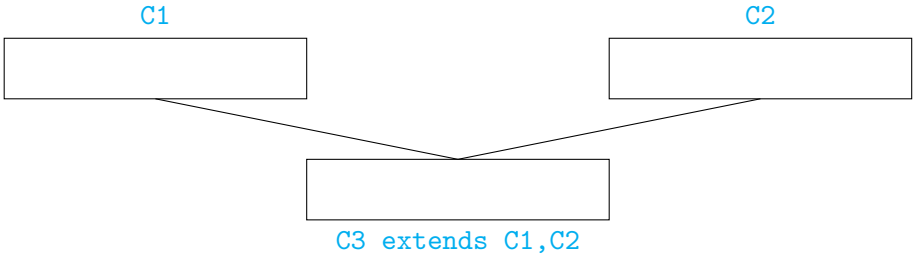
Multiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Need `Circle` to also extend `Comparable`



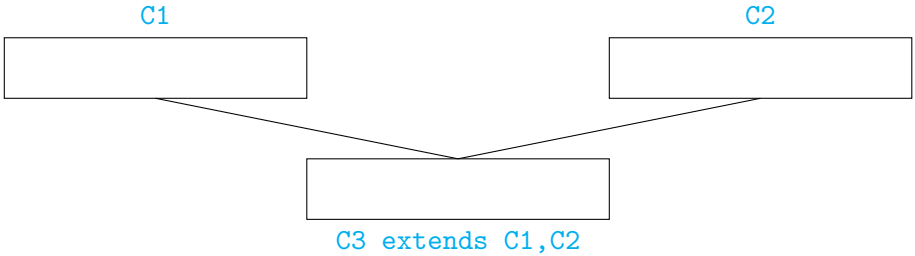
Multiple inheritance

- Can we sort `Circle` objects using the generic functions in `SortFunctions`?
  - `Circle` already extends `Shape`
  - Need `Circle` to also extend `Comparable`
- Can a subclass extend multiple parent classes?



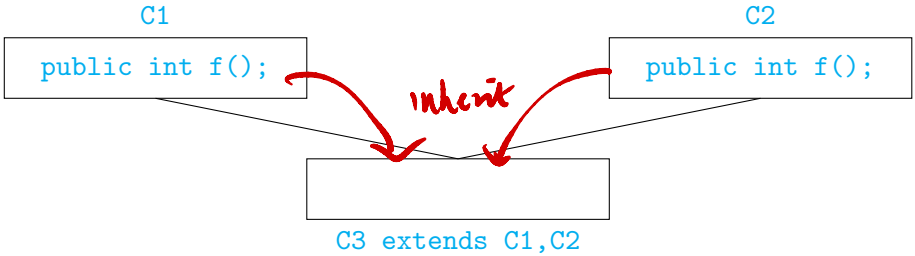
Multiple inheritance

- Can a subclass extend multiple parent classes?



Multiple inheritance

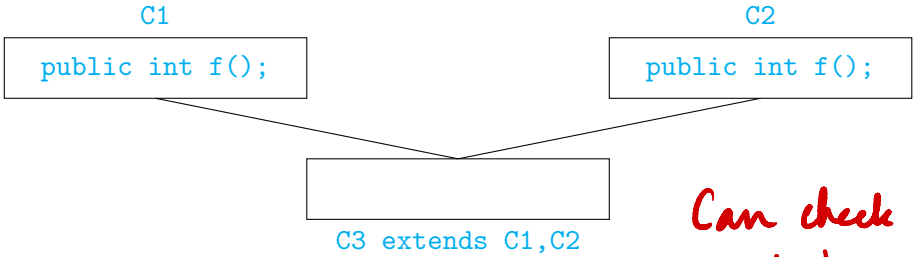
- Can a subclass extend multiple parent classes?



- If `f()` is not overridden, which `f()` do we use in `C3`?

Multiple inheritance

- Can a subclass extend multiple parent classes?



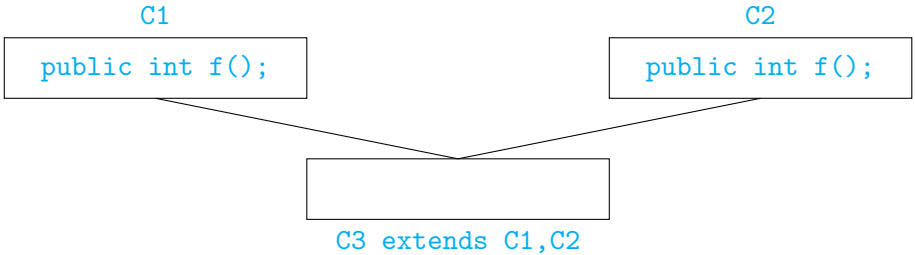
- If `f()` is not overridden, which `f()` do we use in `C3`?
- Java does not allow multiple inheritance

Can check at compile time

1. C3 overrides f()
2. No such clash

Multiple inheritance

- Can a subclass extend multiple parent classes?



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

Interfaces and Multiple inheritance

- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

"Fully" abstract class

Interfaces and Multiple inheritance

- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

- A class that extends an interface is said to implement it:

```
public class Circle extends Shape implements Comparable{
  public double perimeter(){...}
  public int cmp(Comparable s){...}
     ...
}
```

extends

Interfaces and Multiple inheritance

- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

- A class that extends an interface is said to implement it:

```
public class Circle extends Shape implements Comparable{
  public double perimeter(){...}
  public int cmp(Comparable s){...}
     ...
}
```

- Can extend only one class, but can implement multiple interfaces

Interfaces and Multiple inheritance

- An interface is an abstract class with no concrete components

```
public interface Comparable{
  public abstract int cmp(Comparable s);
}
```

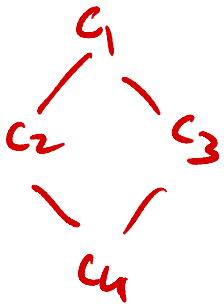
- A class that extends an interface is said to implement it:

```
public class Circle extends Shape implements Comparable{
  public double perimeter(){...}
  public int cmp(Comparable s){...}
     ...
}
```

- Can extend only one class, but can implement multiple interfaces
- Interfaces describe relevant aspects of a class
  - Abstract functions describe a specific "slice" of capabilities
  - Another class only needs to know about these capabilities

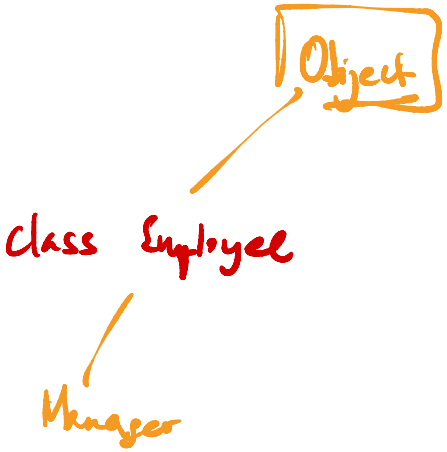
Java class hierarchy

- No multiple inheritance — tree-like



Java class hierarchy

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



Java class hierarchy

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

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

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

Java class hierarchy

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

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

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

- For Java objects `x` and `y`, `x == y` invokes `x.equals(y)`

Java class hierarchy

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

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

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

- For Java objects `x` and `y`, `x == y` invokes `x.equals(y)`
- To print `o`, use `System.out.println(o+"");`
  - Implicitly invokes `o.toString()`

forces conversion to String

Java class hierarchy

- Can exploit the tree structure to write generic functions
  - Example: search for an element in an array

```
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

- Can exploit the tree structure to write generic functions
  - Example: search for an element in an array

```
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, by default

Java class hierarchy

- Can exploit the tree structure to write generic functions
  - Example: search for an element in an array

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

Date.equals()
Object
objarr[i].equals(o)

- Recall that `==` is pointer equality, by default
- If a class overrides `equals()`, dynamic dispatch will use the redefined function instead of `Object.equals()` for `objarr[i] == o`

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`
- May wish to override `equals()` to compare the object state, as follows

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

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`
- May wish to override `equals()` to compare the object state, as follows

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

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

Overriding functions

- For instance, a class `Date` with instance variables `day`, `month` and `year`
- May wish to override `equals()` to compare the object state, as follows

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

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

- Should write, instead

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

type cast

  - Note the run-time type check and the cast

Overriding functions

- Overriding looks for “closest” match

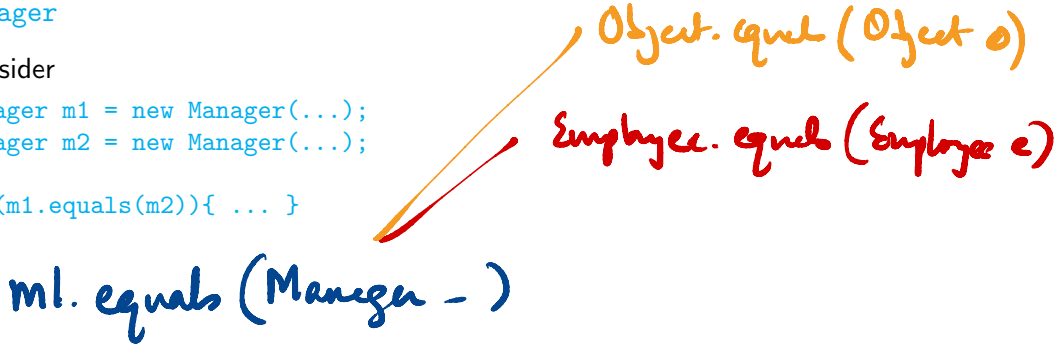
Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`

Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
- Consider

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



Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
- Consider

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

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

Overriding functions

- Overriding looks for "closest" match
- Suppose we have `public boolean equals(Employee e)` but no `equals()` in `Manager`
- Consider

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

- `public boolean equals(Manager m)` is compatible with both `boolean equals(Employee e)` and `boolean equals(Object o)`
- Use `boolean equals(Employee e)`

Subclasses, subtyping and inheritance

- Class hierarchy provides both subtyping and inheritance

Subclasses, subtyping and inheritance

- Class hierarchy provides both subtyping and inheritance
- Subtyping
  - Capabilities of the subtype are a superset of the main type
  - If `B` is a subtype of `A`, wherever we require an object of type `A`, we can use an object of type `B`
  - `Employee e = new Manager(...);` is legal

Subclasses, subtyping and inheritance

- Class hierarchy provides both subtyping and inheritance
- Subtyping
  - Capabilities of the subtype are a superset of the main type
  - If `B` is a subtype of `A`, wherever we require an object of type `A`, we can use an object of type `B`
  - `Employee e = new Manager(...);` is legal
- Inheritance
  - Subtype can reuse code of the main type
  - `B` inherits from `A` if some functions for `B` are written in terms of functions of `A`
  - `Manager.bonus()` uses `Employee.bonus()`

Subtyping vs inheritance

- Consider the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`

Subtyping vs inheritance

- Consider the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
- What are the subtype and inheritance relationships between these classes?

Subtyping vs inheritance

- Consider the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
- What are the subtype and inheritance relationships between these classes?
- Subtyping
  - `deque` has more functionality than `queue` or `stack`
  - `deque` is a subtype of both these types

Subtyping vs inheritance

- Consider the following example
  - `queue`, with methods `insert-rear`, `delete-front`
  - `stack`, with methods `insert-front`, `delete-front`
  - `deque`, with methods `insert-front`, `delete-front`, `insert-rear`, `delete-rear`
- What are the subtype and inheritance relationships between these classes?
- Subtyping
  - `deque` has more functionality than `queue` or `stack`
  - `deque` is a subtype of both these types
- Inheritance
  - Can suppress two functions in a `deque` and use it as a `queue` or `stack`
  - Both `queue` and `stack` inherit from `deque`

Subclasses, subtyping and inheritance

- Class hierarchy represents both subtyping and inheritance

Subclasses, subtyping and inheritance

- Class hierarchy represents both subtyping and inheritance
- Subtyping
  - Compatibility of interfaces.
  - `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.

Subclasses, subtyping and inheritance

- Class hierarchy represents both subtyping and inheritance
- Subtyping
  - Compatibility of interfaces.
  - `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.
- Inheritance
  - Reuse of implementations.
  - `B` inherits from `A` if some functions for `B` are written in terms of functions of `A`.

Subclasses, subtyping and inheritance

- Class hierarchy represents both subtyping and inheritance
- Subtyping
  - Compatibility of interfaces.
  - `B` is a subtype of `A` if every function that can be invoked on an object of type `A` can also be invoked on an object of type `B`.
- Inheritance
  - Reuse of implementations.
  - `B` inherits from `A` if some functions for `B` are written in terms of functions of `A`.
- Using one idea (hierarchy of classes) to implement both concepts blurs the distinction between the two