Inheritance

Big software

- **software engineering**: The practice of conceptualizing, designing, developing, documenting, and testing large-scale computer programs.
- Large-scale projects face many issues:
  - getting many programmers to work together
  - getting code finished on time
  - avoiding redundant code
  - finding and fixing bugs
  - maintaining, improving, and reusing existing code
- **code reuse**: The practice of writing program code once and using it in many contexts.

Example

- You have been tasked with writing a program that handles pay for the employees of a non-profit organization.
- The organization has several types of employees on staff:
  - Full-time employees
  - Hourly workers
  - Volunteers
  - Executives

Example

- Paying an employee:
  - Full-time employees – have a monthly pay
  - Hourly workers – hourly wages + hours worked
  - Volunteers – no pay
  - Executives – receive bonuses

Design

- Need class/classes that handle employee pay (should also store employee info such as name, phone #, address).
- Possible choices:
  - A single Employee class that knows how to handle different types of employees
  - A separate class for each type of employee.
- What are the advantages/disadvantages of each design?

Design

- All types of staff members need to have some basic functionality – capture that in a class called `StaffMember`

Design

All types of staff members need to have some basic functionality – capture that in a class called `StaffMember`

```
public class StaffMember {
    private String name;
    private String address;
    private String phone;

   public StaffMember (String name, String address,
                       String phone) {
       this.name = name;
       this.address = address;
       this.phone = phone;
    }
  // not shown:  getters and setters
}
```

Code re-use

- We'd like to be able to do the following:

```
// A class to represent a paid employee.
public class Employee {
     <copy all the contents from StaffMember class.>

     private double payRate;
     public double pay() {
        return payRate;
      }

}
```

- All this without explicitly copying any code!

Inheritance

- Creating a subclass, general syntax:

  `public class` ***<name>*** `extends` ***<superclass name>*** `{`

  - Example:

```
public class Employee extends StaffMember {
    ....
}
```

- By extending `StaffMember`, each `Employee` object now:
  - has `name, address, phone` instance variables and `get/setName(), get/setAddress(), get/setPhone()` methods automatically
  - can be treated as a `StaffMember` by any other code (seen later)

    (e.g. an `Employee` could be stored in a variable of type `StaffMember` or stored as an element of an array `StaffMember[]`)

Inheritance

- **inheritance**: A way to create new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between classes
- A class *extends* another by absorbing its state and behavior.
  - **super-class**: The parent class that is being extended.
  - **sub-class**: The child class that extends the super-class and inherits its behavior.
    - The subclass receives a copy of every field and method from its super-class.
    - The subclass is **a more specific** type than its super-class (an **is-a** relationship)

Single Inheritance in Java

- Creating a subclass, general syntax:
  - `public class` ***<name>*** `extends` ***<superclass name>***
  - ***Can only extend a single class in Java!***

- Extends creates an is-A relationship
  - `class` ***<name>*** `is-A` ***<superclass name>***
  - ***This means that anywhere a <superclass variable> is used, a <subclass variable> may be used.***
  - Classes get all the instance variables/methods of their ancestors, **but cannot necessarily directly access them...**

New access modifier - protected

- public - can be seen/used by everyone
- **protected** – can be seen/used within class and any subclass.
- private - can only be seen/used by code in class (not in subclass!)

Extends/protected/super

```
public class Employee extends StaffMember {
    protected String socialSecurityNumber;
    protected double payRate;

    public Employee (String name, String address,
        String phone, String socSecNumber, double rate){
        super(name, address, phone);
        socialSecurityNumber = socSecNumber;
        payRate = rate;
    }
    public double pay(){
        return payRate;
    }
}
```

StaffMember needs to change a bit

```
public class StaffMember {
    protected String name;
    protected String address;
    protected String phone;

    public StaffMember (String name, String address, String
    phone) {
        this.name = name;
        this.address = address;
        this.phone = phone;
    }
}
```

Overriding methods

- **override**: To write a new version of a method in a subclass that replaces the super-class's version.
  - There is no special syntax for overriding.
    To override a super-class method, just write a new version of it in the subclass. This will replace the inherited version.

  - Example:

```
public class Hourly extends Employee {
    // overrides the pay method in Employee class
    public double pay () {
      double payment = payRate * hoursWorked;
      hoursWorked = 0;
      return payment;
  }
```

Calling overridden methods

- The new method often relies on the overridden one. A subclass can call an overridden method with the `super` keyword.
- Calling an overridden method, syntax:

`super.`***<method name>*** `(` ***<parameter(s)>*** `)`

```
public class Executive extends Employee {
  public double pay() {
    double payment = super.pay() + bonus;
    bonus = 0;  // one time bonus
    return payment;
  }
```

Inheritance and Polymorphism

Constructors

- Constructors are not inherited.
  - Default constructor:

```
public Employee(){
    super();      // calls StaffMember() constructor
}
```

  - Constructor needs to call super-class constructors explicitly:

```
public Employee (String name, String address, String phone,
                 String socSecNumber, double rate) {
  super (name, address, phone);
  socialSecurityNumber = socSecNumber;
  payRate = rate;
}
```

The `super` call must be the **first statement** in the constructor.

Everything is an Object

- Every class in Java implicitly extends the Java `Object` class.
- Therefore every Java class inherits all the methods of the class `Object`, such as
  - `equals(Object other)`
  - `toString()`
- Often we want to override the standard implementation
- Note the difference between overloading and overriding!

The equals method

- You might think that the following is a valid implementation of the `equals` method:

```
public boolean equals(Object other) {
    if (name.equals(other.name)) {
        return true;
    } else {
        return false;
    }
}
```

However, it does not compile.

```
StaffMember.java:36: cannot find symbol
symbol  : variable name
location: class java.lang.Object
```

- **Why?** Because an Object does not have a name instance variable.

Type casting

- The object that is passed to `equals` can be cast from `Object` into your class's type.
  - Example:

```
public boolean equals(Object o) {
    StaffMember other = (StaffMember) o;
    return name == other.name;
}
```

- Type-casting with objects behaves differently than casting primitive values.
  - We are really casting a **reference of type `Object`** into a **reference of type `StaffMember`**.
  - We're promising the compiler that `o` refers to a `StaffMember` object, and thus has an instance variable `name`.

Type casting: equals example

- The object that is passed to `equals` can be cast from `Object` into your class's type.
- Equals example:

```
public boolean equals(Object o) {
    StaffMember other = (StaffMember) o;
    return name.equals(other.name);
}
```

instanceof

- We can use a keyword called `instanceof` to ask whether a variable refers to an object of a given type.
  - The `instanceof` keyword, general syntax:

    <variable> `instanceof` ***<type>***

  - The above is a `boolean` expression that can be used as the test in an `if` statement.

  - Examples:

```
String s = "hello";
StaffMember p = new StaffMember(…);
if(s instanceof String) ...
if(p instanceof String) ...
```

Our final version of equals

- This version of the `equals` method allows us to correctly compare `StaffMember` objects with any type of object:

```
// Returns whether o refers to a StaffMember
// object with the same name
public boolean equals(Object o) {
    if (o instanceof StaffMember) {
        StaffMember other = (StaffMember) o;
        return name.equals(other.name);
    } else {
        return false;
    }
}
```

instanceof

- In our payroll example, Employee extends StaffMember. Consider the following snippet of code:

```
Employee employee = new Employee(…);
Boolean result = (employee instanceof StaffMember);
```

What will be the value of result?

a) `true`
b) `false`

Binding: which method is called?

- Assume that the following four classes have been declared:

```
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }

    public void method2() {
        System.out.println("foo 2");
    }

    public String toString() {
        return "foo";
    }
}

public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}
```

Example

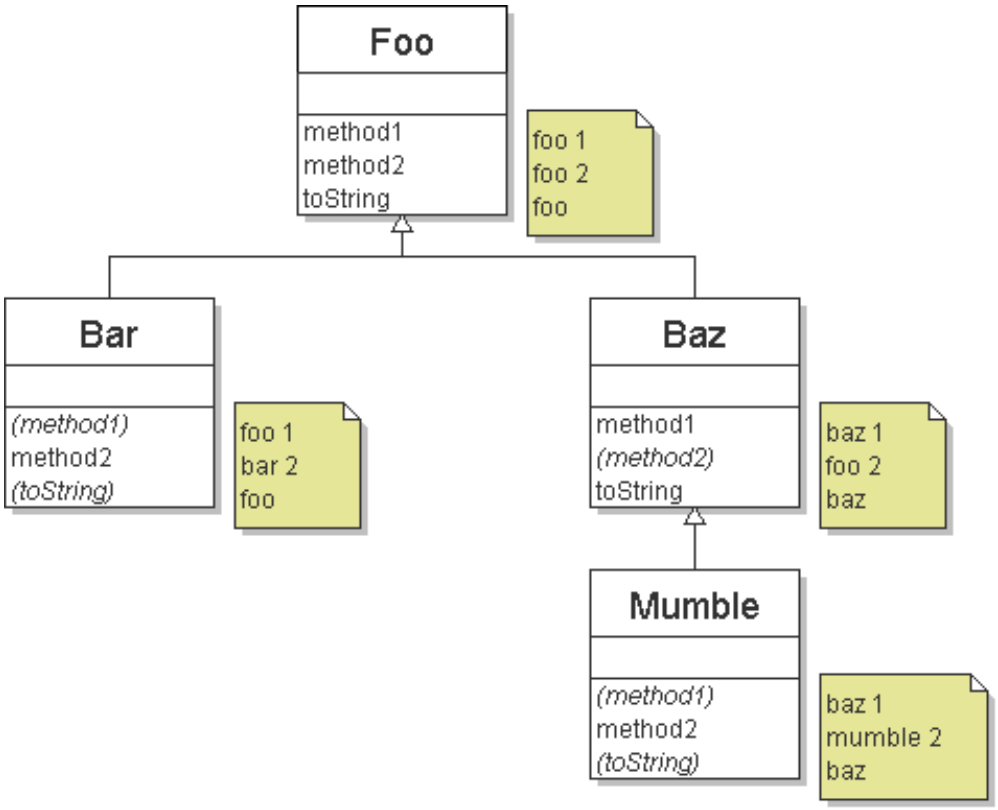
```
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }
    public String toString() {
        return "baz";
    }
}
public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```

- The output of the following client code?

```
Foo[] a = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
    a[i].method1();
    a[i].method2();
    System.out.println();
}
```

Describing inheritance and binding

- UML diagram: Subclasses point to their super-class
- List methods (inherited methods in parenthesis)
- Method called is the nearest in the hierarchy going up the tree
  - This is a dynamic (run time) phenomenon called **dynamic binding**



Example (solved)

```
Foo[] a = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
    a[i].method1();
    a[i].method2();
    System.out.println()
}
```

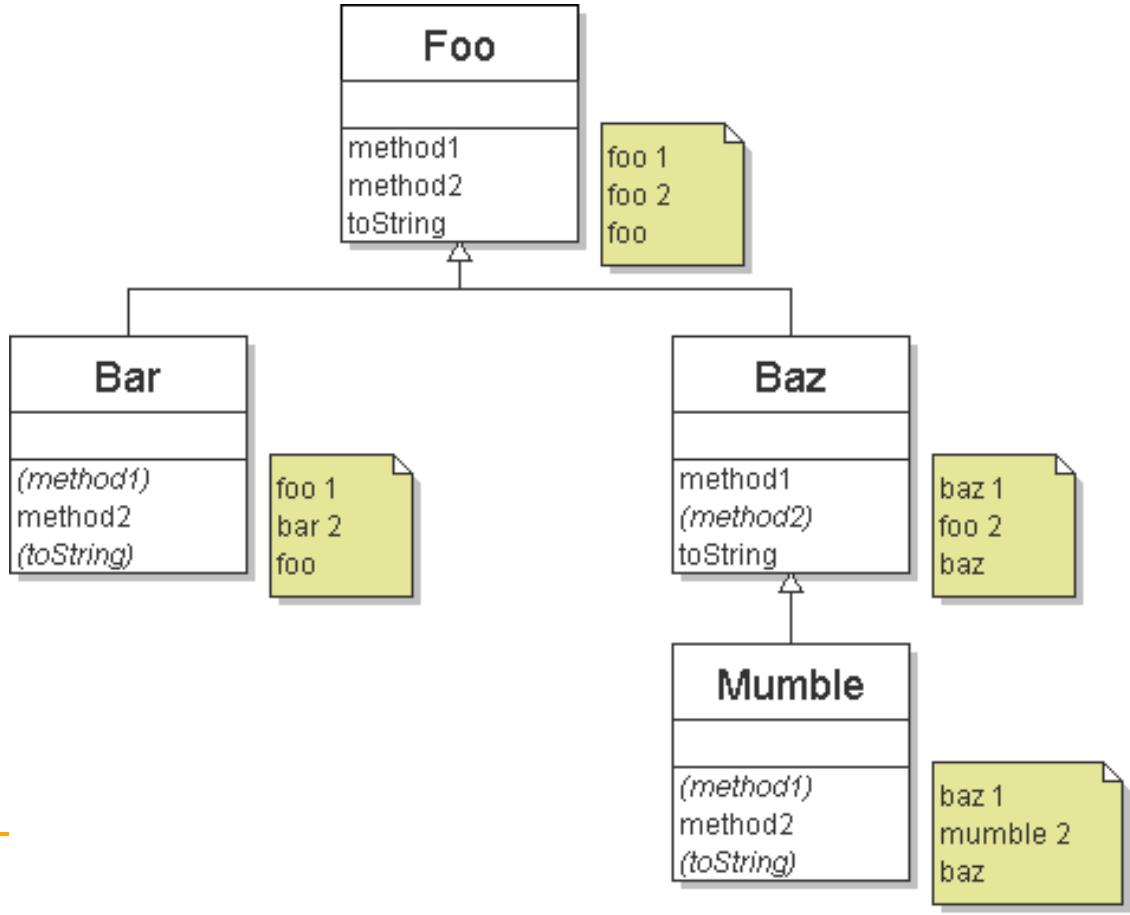
Output?

```
baz
baz 1
foo 2

foo
foo 1
bar 2

baz
baz 1

mumble 2
foo
foo 1
foo 2
```



Polymorphism

- It's legal for a variable of a super-class to refer to an object of one of its subclasses.

  Example:

```
staffList = new StaffMember[6];
staffList[0] = new Executive("Sam", "123 Main Line",
      "555-0469", "123-45-6789", 2423.07);
staffList[1] = new Employee("Carla", "456 Off Line",
      "555-0101", "987-65-4321", 1246.15);
staffList[2] = new Employee("Woody", "789 Off Rocker",
      "555-0000", "010-20-3040", 1169.23);
((Executive)staffList[0]).awardBonus (500.00);
```

Arrays of a super-class type can store any subtype as elements.

Polymorphism and casting

- When a primitive type is used to store a value of another type (e.g. an `int` in a `double` variable) conversion takes place.
- When a subclass is stored in a superclass no conversion occurs!

Polymorphism defined

- **Polymorphism**: the ability for the same code to be used with several different types of objects and behave differently depending on the actual type of object used.
- Example:

```
for (int count=0; count < staffList.length; count++)
{
    amount = staffList[count].pay();  // polymorphic
}
```

Polymorphism and parameters

- You can pass any subtype of a parameter's type.

```
public class EmployeeMain {
    public static void main(String[] args) {
        Executive lisa = new Executive(…);
        Volunteer steve = new Volunteer(…);
        payEmployee(lisa);
        payEmployee(steve);
    }

    public static void payEmployee(StaffMember s) {
        System.out.println("salary = " + s.pay());
    }

}
```

Notes about polymorphism

- The program doesn't know which pay method to call until it's actually running. This has many names: late binding, dynamic binding, virtual binding, and dynamic dispatch.
- You can only call methods known to the super-class, unless you explicitly cast.
- You **cannot** assign a super-class object to a sub-class variable (a cow is an animal, **but an animal is not a cow!)**

Abstract classes

- An **abstract class:** can leave one or more method implementations unspecified
- An abstract method has no body (i.e.,no implementation).
- Hence, an abstract class is incomplete and cannot be instantiated, but can be used as a base class.

```
abstract public class abstract-base-class-name {
    public abstract return-type method-name(params);
}
public class derived-class-name {
    public return-type method-name(params)
                { statements; }
}
```

A subclass is required to override the abstract method and provide an implementation.

Example

- Let's convert `StaffMember` to an abstract class....

Example

- Let's convert `StaffMember` to an abstract class.

```
public abstract class StaffMember {
   ...
   public abstract double pay();
}
```

- Now the sub classes must override pay(), thereby implementing pay() appropriately for each sub type

Abstract classes

- When to use abstract classes
  - To represent entities that are insufficiently defined
  - Group together data/behavior that is useful for its subclasses

Inheritance: FAQ

- How can a subclass call a method or a constructor defined in a super-class?
  - Use super() or super.method()
- Does Java support multiple inheritance?
  - No. Use interfaces (later) instead
- What restrictions are placed on method overriding?
  - Same name, argument list, and return type. May not throw exceptions that are not thrown by the overriden method, or limit the access to the method
- Does a class inherit the constructors of its super-class?
  - No. Need to call them explicitly

this and super in constructors

- `this(...)` calls a constructor of the same class.
- `super(...)` calls a constructor of the super-class.
- Both need to be the first action in a constructor.