Inheritance and Polymorphism

Constructors

- Constructors are not inherited.
  - They receive a default constructor that contains:
    ```java
    public Employee() {
        super();  // calls StaffMember() constructor
    }
    ```
  - Need to call superclass constructors explicitly:
    ```java
    public Employee(String eName, String eAddress, String ePhone, String socSecNumber, double rate) {
        super(eName, eAddress, ePhone);
        socialSecurityNumber = socSecNumber;
        payRate = rate;
    }
    ```
    - The `super` call must be the first statement in the constructor.

Everything is an Object

- Every object in Java implicitly extends the Java `Object` class.
- Therefore every Java class inherits all the methods of the class `Object`.
- `equals(Object other)`
- `toString()`
- May 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:
  ```java
  public boolean equals(Object other) {
      if (name == other.name) {
          return true;
      } else {
          return false;
      }
  }
  ```
  - However, it does not compile.
  ```java
  StaffMember.java:36: cannot find symbol
  symbol  : variable name
  location: class java.lang.Object
  if (name == other.name) {
  ^
  ```

Type casting

- The object that is passed to `equals` can be cast from `Object` into your class's type.
  - Example:
    ```java
    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 `Point`.
- We're promising the compiler that `o` refers to a `StaffMember` object.

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:
    ```java
    <variable> instanceof <type>
    ```
  - The above is a boolean expression that can be used as the test in an if statement.
  - Examples:
    ```java
    String s = "hello";
    StaffMember p = new StaffMember(...);
    s instanceof String?
    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:

```java
public boolean equals(Object o) {
    if (o instanceof StaffMember) {
        StaffMember other = (StaffMember) o;
        return name == other.name;
    } else {
        return false;
    }
}
```

Polymorphism

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

Example:

```java
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", "123-45-6789", 1246.15);
((Executive)staffList[0]).awardBonus (500.00);
```

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:

```java
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.

```java
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, virtual binding, and dynamic dispatch.
- You can only call methods known to the superclass (unless you explicitly cast).
A few classes with inheritance relationships are shown.
A client program calls methods on objects of each class.
You must read the code and determine the client's output.
A question like this will likely be on the midterm/final exam!

Assume that the following four classes have been declared:

```java
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");
    }
}
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?

```java
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
bar 2
baz
baz 1
mumble 2
foo
foo 1
```

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

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

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

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

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 superclass?
  - Use `super()` or `super.method()`
- Can a top-level class be private or protected?
  - No. The compiler will complain!
- Does Java support multiple inheritance?
  - No. Use interfaces instead (coming next!)
- What restrictions are placed on method overriding?
  - Same name, argument list, and return type. May not throw exceptions that are not thrown by the overridden method, or limit the access to the method
- Does a class inherit the constructors of its superclass?
  - No. Need to call them explicitly

this and super

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