Advanced Programming Techniques in Java

COSI 12B

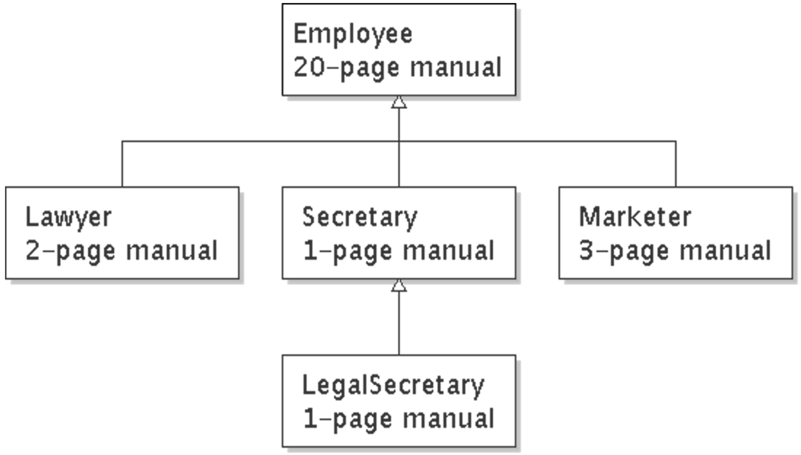
Casting & Polymorphism

Lecture 13

Class Objectives

- Casting & Polymorphism (9.3)

Review: Is-a relationships



- **Is-a relationship** is a hierarchical connection where one category can be treated as a specialized version of another
  - Every marketer **is-an** employee
  - Every legal secretary **is-a** secretary

- **Inheritance hierarchy** is a set of classes connected by is-a relationships that can share common code

Review: Overloading vs. Overriding

- What is the difference between method **overloading** and method **overriding**?
  - **Overloading**: one class contains multiple methods with the same name but different parameter signatures
  - **Overriding**: a subclass substitutes its own version of an otherwise inherited method, with the same name and the same parameters
  - Overloading lets you define a similar operation in different ways for different data
  - Overriding lets you define a similar operation in different ways for different object types

Review: The `super` reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The `super` reference can be used to refer to the parent class, and often is used to invoke the parent's constructor
- A child's constructor is responsible for calling the parent's constructor
- The first line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference other variables and methods defined in the parent's class

Review: One more level of information hiding

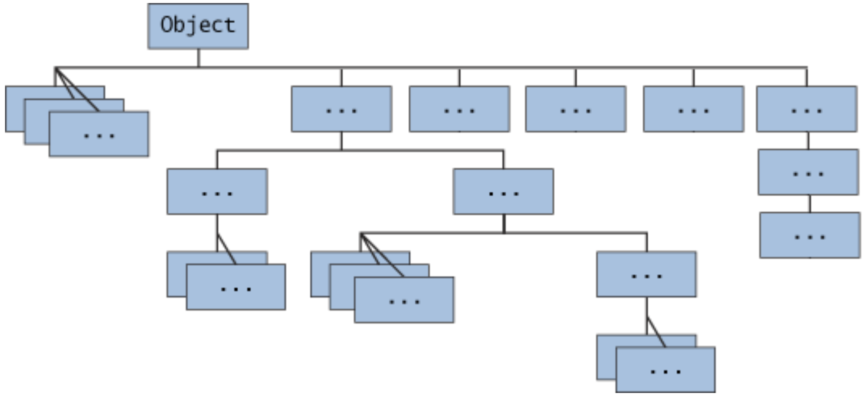
- Keyword `protected`
  - Provides intermediate level of security between `public` and `private` access
  - Allows a member of a superclass to be inherited into a subclass
  - Can be used within own class or in any classes extended from that class
  - Cannot be used by “outside” classes

- When might you need it? (RARELY USED)
  - If you want your fields to be `private` but you don’t want to have a public accessor method
  - `public` methods can be used by EVERY CLASS not only the subclasses

Inheritance and Polymorphism

- **Inheritance**: A way to form new classes based on existing classes, taking on their attributes/behavior
- **Polymorphism**: Ability for an object to be used as if it was of different type

The `Object` class



- The `Object` class is the parent class of all the classes in java
  - All classes are derived from the `Object` class (i.e. every class implicitly extends `Object`)
  - It defines and implement the behavior common to all classes
  - It is defined in the `java.lang` package
- If a class is not explicitly defined to be the child of an existing class, it is assumed to be the child of the `Object` class

The `Object` class and inheritance

- The `Object` class contains a few useful methods, which are inherited by all classes
- Other classes can override these methods
  - e.g., `toString()`, `equals(Object obj)`
  - Otherwise, the default implementation is used

The `Object` class `toString` method

- Every time we have defined `toString`, we have been overriding an existing definition
- The `toString` method in the `Object` class is defined to return a `String` that contains the name of the object’s class together along with some other information
- All objects are guaranteed to have a `toString` method via inheritance
- Thus, the `println` method can call `toString` for any object that is passed to it

Object variables

- You can store any object in a variable of type Object

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";
Object o3 = new Scanner(System.in);
```

- An Object variable only knows how to do general things

```
String s = o1.toString();      // ok
int len = o2.length();         // error
String line = o3.nextLine();   // error
```

- You can write methods that accept an Object parameter

```
public void checkForNull(Object o) {
    if (o == null) {
        throw new IllegalArgumentException();
    }
}
```

The `Object` class `equals` method

- The `equals` method of the `Object` class returns true if two references are the same
- We can override `equals` in any class to define equality in some more appropriate way
  - The `String` class (as we've seen) defines the `equals` method to return `true` if two `String` objects contain the same characters

equals and Object

```
public boolean equals(Object name) {
        statement(s) that return a boolean value ;
}
```

- The parameter to `equals` must be of type `Object`
- `Object` is a general type that can match any object
- Having an `Object` parameter means *any* object can be passed

Final `equals` method

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and compare it
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        // o is not a Point; cannot be equal
        return false;
    }
}
```

Review: Data Conversions (primitive)

- *Assignment conversion* occurs when a value of one type is assigned to a variable of another
- *Arithmetic promotion* happens automatically when operators in expressions convert their operands
- *Casting* is accomplished by explicitly casting a value
  - To cast, the type is put in parentheses in front of the value being converted
  - For example, if `total` and `count` are integers, but we want a floating-point result when dividing them, we can cast `total`:

```
result = (double) total / count;
```

Type cast operator

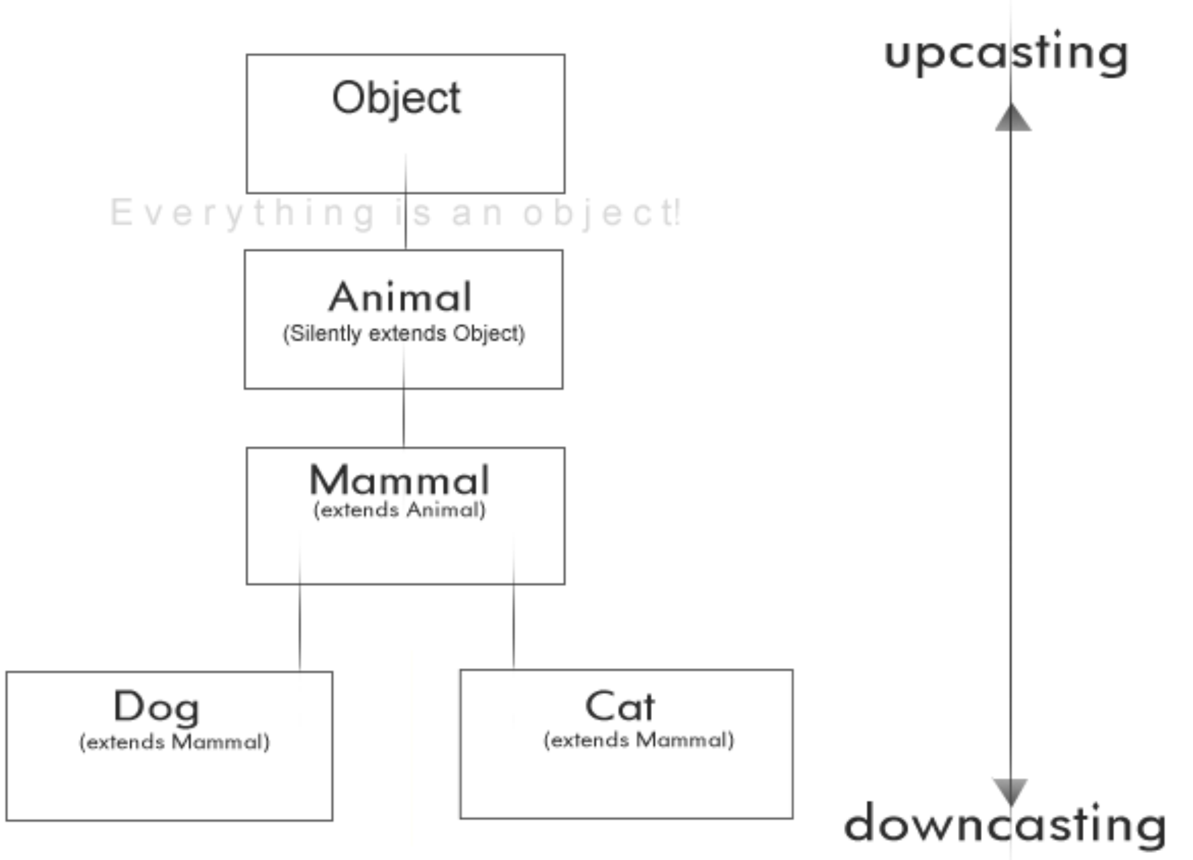
Object Casting

- Casting allows the use of an object of one type in place of another type
  - It applies among the objects permitted by inheritance

- **Upcasting**: an object of a subclass type can be treated as an object of any superclass type
  - Upcasting is automatic in Java (**implicit casting**)

- **Downcasting**: treating a superclass object as its real subclass
  - Downcasting must be specified (**explicit casting**)

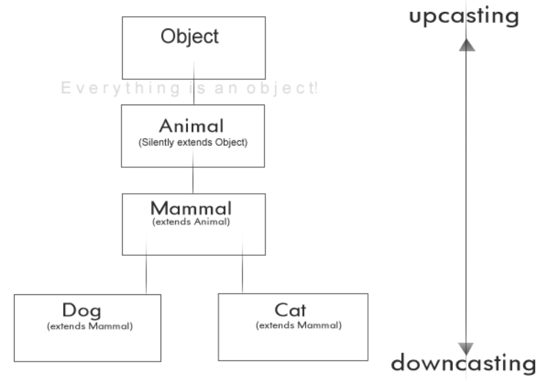
Object Casting



by Sinipull for codecall.net

Upcasting

```
public class Animal {
    protected int health = 100;
}
public class Mammal extends Animal {
}
public class Cat extends Mammal {
}
public class Dog extends Mammal {
}
```



Object

Everything is an object!

Animal
(Silently extends Object)

Mammal
(extends Animal)

Dog
(extends Mammal)

Cat
(extends Mammal)

upcasting

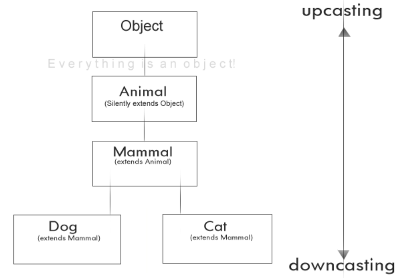
downcasting

by Sinipull for codecall.net

```
public class Test {
    public static void main(String[] args) {
          Cat c = new Cat();
          System.out.println(c.health);
          Dog d = new Dog();
          System.out.println(d.health);
    }
}
```

Output:

```
100
100
```



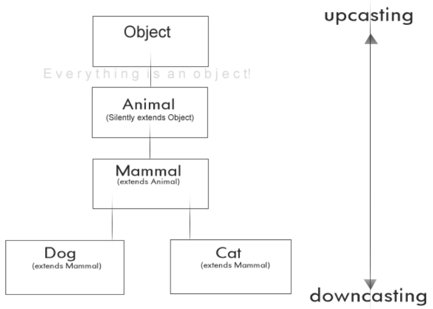
by Sinipull for codecall.net

Upcasting

- By casting an object, you are NOT actually changing the object itself. You are just labeling it differently
- `Animal cat1 = new Cat();`
- The object doesn't stop from being a `Cat`. It's still a `Cat`, but it's just treated as any other `Animal` and its `Cat` properties are hidden until it's downcasted to a `Cat` again.

```
public class Test {
     public static void main(String[] args) {

          Cat c = new Cat();
          System.out.println(c);
          Mammal m = c; // upcasting
          System.out.println(m);
}
```

```
Cat@65ae6ba4
Cat@65ae6ba4
```



by Sinipull for codecall.net

Upcasting and Downcasting

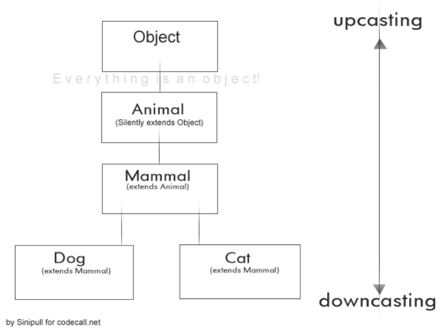
- Upcasting is done automatically

```
Mammal m = new Cat();
```

- But downcasting must always be done manually

```
Cat c1 = new Cat();
Animal a = c1; //automatic upcasting to Animal
Cat c2 = (Cat) a; //manual downcasting back to a Cat
```

Upcasting and Downcasting



by Sinipull for codecall.net

- Why upcasting is automatic, but downcasting must be manual?
- Upcasting can never fail. But if you have a group of different `Animals` and want to downcast them all to a `Cat`, then there's a chance, that some of these `Animals` are `Dogs`, and process fails, by `throwing ClassCastException`

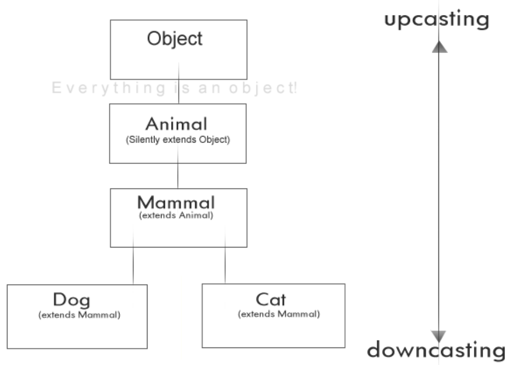
```
Cat c1 = new Cat();
Animal a = c1; //upcasting to Animal
if(a instanceof Cat){ // testing if the Animal is a Cat
            System.out.println("It's a Cat!");
            Cat c2 = (Cat)a;
}
```

I'm still a Cat after upcasting, but compiler treats me as an Object
That means i can't do anything, that's specific for Animals... or Cats.
object
animal
mammal
cat
i can't!
Be a dog!
java.lang.ClassCastException: Cat cannot be cast to Dog

by Sinipull for codecall.net

Downcasting

```
Mammal m = new Mammal();
Cat c = (Cat)m;
```



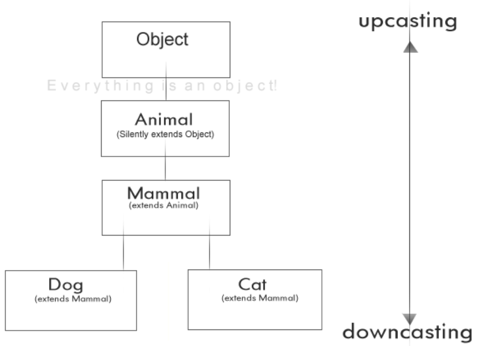
by Sinipull for codecall.net

- Such code passes compiling but throws “`java.lang.ClassCastException`: `Mammal cannot be cast to Cat`" exception during running, because trying to cast a `Mammal`, which is not a `Cat`, to a `Cat`

- Do not confuse **variables** with **instances**.
  - `Cat` from `Mammal` variable can be cast to a `Cat`, but `Mammal` from `Mammal` variable cannot be cast to a `Cat`

Is Cat a Mammal? Yes, it is - that means, it can be cast
Is Mammal a Cat? No, it isn't - it cannot be cast.
Is Cat a Dog? No, it cannot be cast

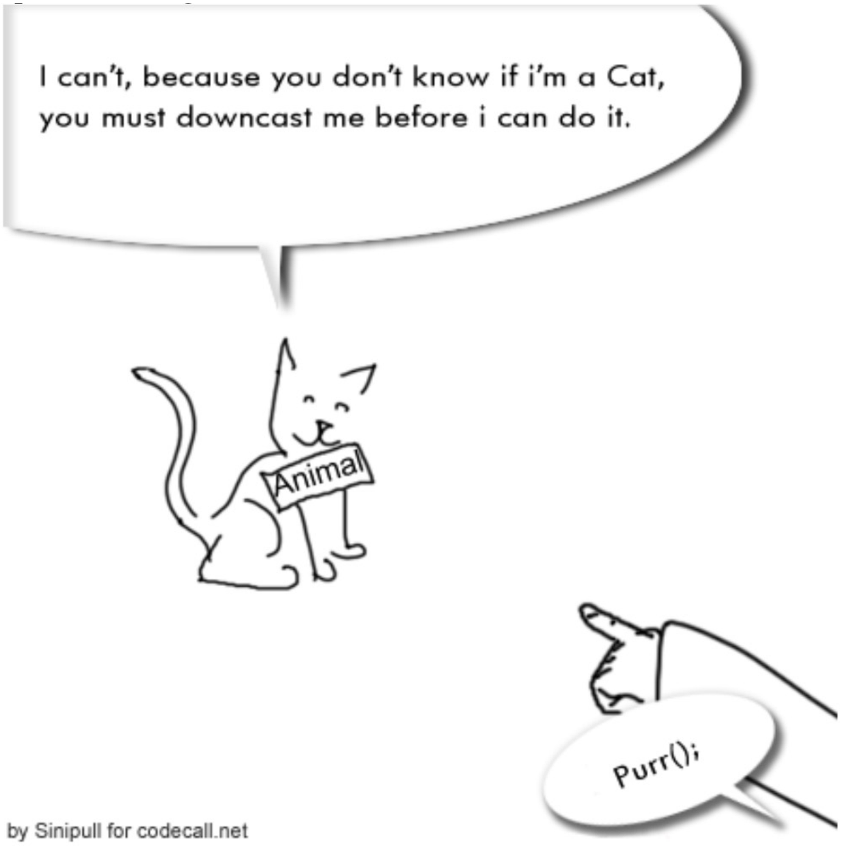
Upcasting and Downcasting



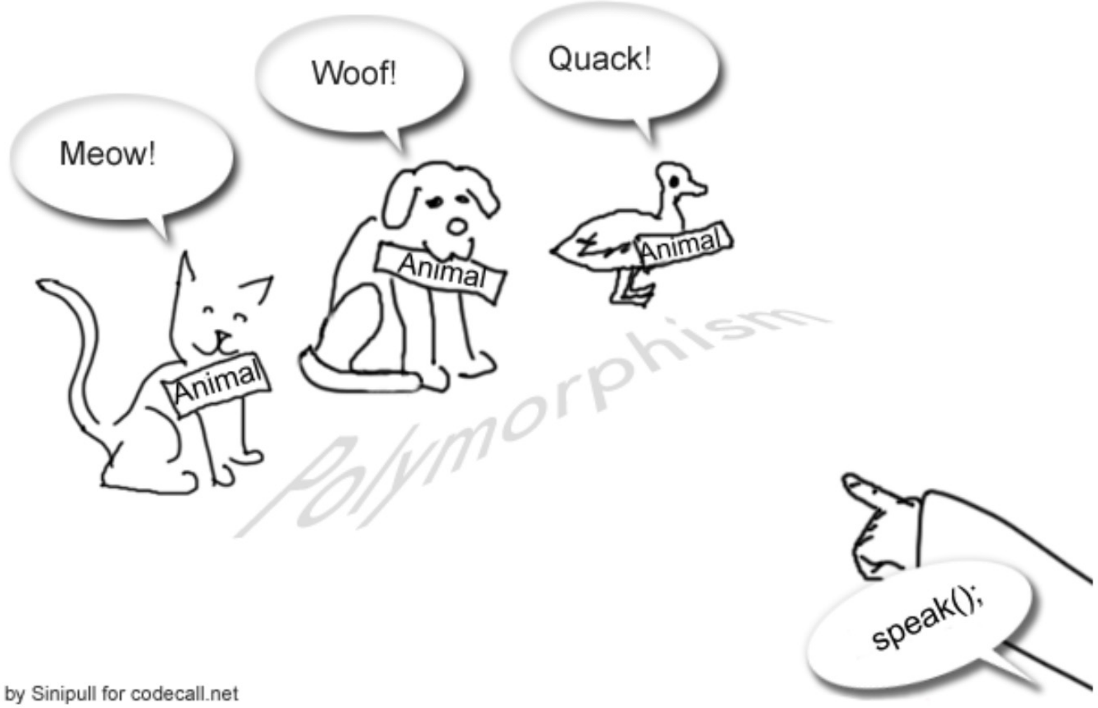
by Sinipull for codecall.net

- If you upcast an object, it will lose all its properties, which were inherited from below its current position
- For example, if you cast a `Cat` to an `Animal`, it will lose properties inherited from `Mammal` and `Cat`.
    - The data will not be lost, you just can't use it, until you downcast the object to the right level

Upcasting and Downcasting

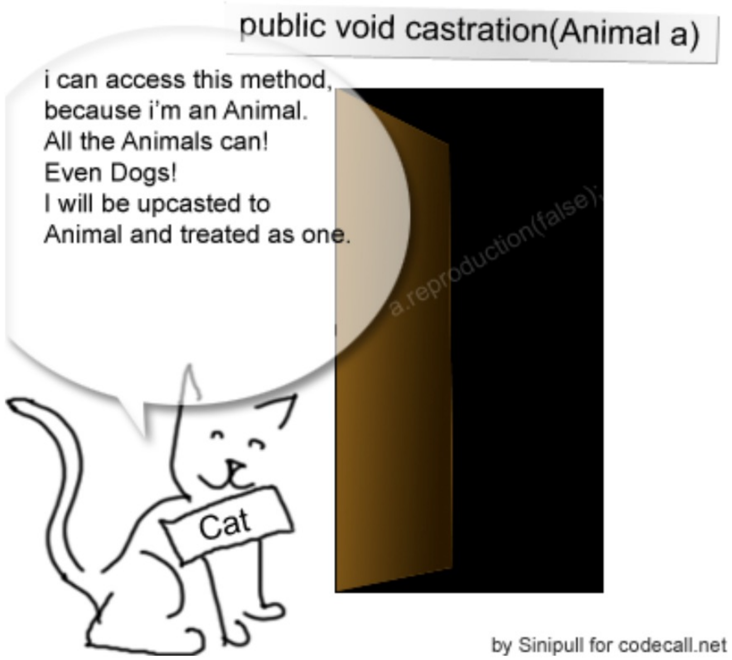

I can't, because you don't know if i'm a Cat, you must downcast me before i can do it.
Animal
Purr();

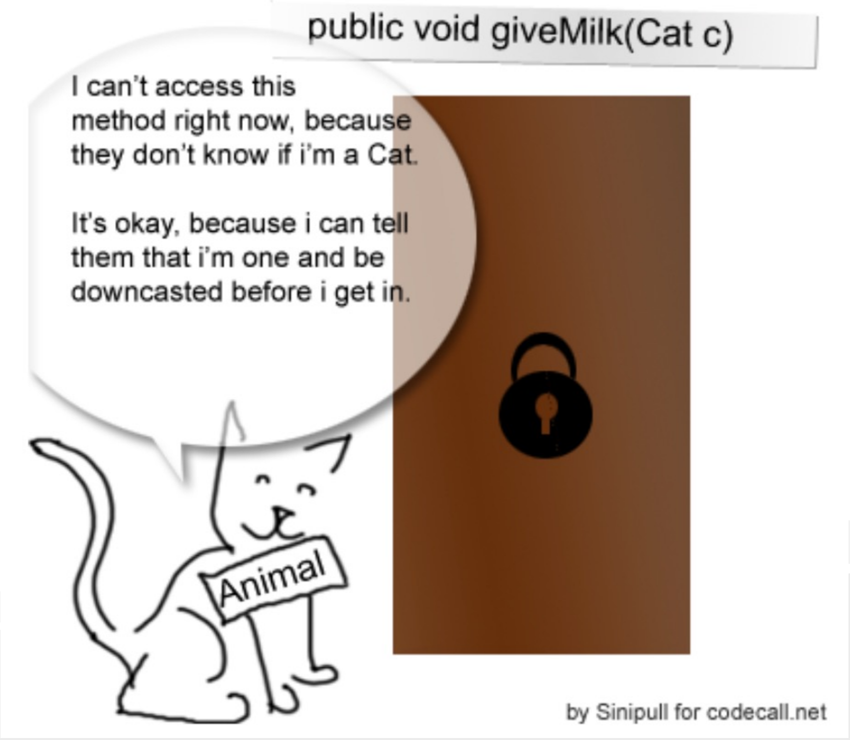
by Sinipull for codecall.net



by Sinipull for codecall.net

Upcasting and Downcasting


public void castration(Animal a)
i can access this method, because i'm an Animal. All the Animals can! Even Dogs! I will be upcasted to Animal and treated as one.
a.reproduction(false);
Cat
by Sinipull for codecall.net


public void giveMilk(Cat c)
I can't access this method right now, because they don't know if i'm a Cat.
It's okay, because i can tell them that i'm one and be downcasted before i get in.
Animal
by Sinipull for codecall.net

Final words on casting …

- An object of a subclass type can be treated as an object of any superclass type.
  - This is called upcasting. Upcasting is done automatically
- An object of a superclass type can be treated as an object of a subclass type.
  - This is called downcasting. Downcasting must be done manually
- Upcasting and downcasting objects are **NOT** like casting primitives from one to other

Polymorphism

Me: *explains polymorphism*

Friend: So the subclass the same thing as the superclass?

Me:

Polymorphism

- **Polymorphism** indicates the ability for the same code to be used with different types of objects and behave differently with each
- `System.out.println()` can print any type of object
  - Each one displays in its own way on the console

Coding with polymorphism

- A variable of type `T` can hold an object of any subclass of `T`

```
Employee employee1 = new Lawyer();
```

  - You can call any methods from `Employee` on `employee1`
  - You can not call any methods specific to `Lawyer` (e.g. `sue`)

- When a method is called on `employee1`, it behaves as a `Lawyer`

```
System.out.println(employee1.getVacationDays()); // 3 weeks
System.out.println(employee1.getVacationForm());   // pink
```

- But you cannot call this

```
System.out.println(employee1.sue());
```

Polymorphism and parameters

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

```
public class EmployeeMain3 {
    public static void main(String[] args) {
        Lawyer law = new Lawyer();
        Secretary sec= new Secretary();
        printInfo(law);
        printInfo(sec);
    }

    public static void printInfo(Employee empl){
          empl.getSalary();
          empl.getVacationDays();
          empl.getVacationForm();
    }
}
```

You can pass both `Lawyer` and `Secretary` objects

Depending on the type you passed it calls the corresponding method

OUTPUT

```
I earn $40,000
I receive 3 weeks vacation
Use the pink vacation form

I earn $40,000
I receive 2 weeks vacation
Use the yellow vacation
form
```

Polymorphism and parameters

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

```
public class EmployeeMain3 {
    public static void main(String[] args) {
        Lawyer law= new Lawyer();
        Secretary sec= new Secretary();
        printInfo(law);
        printInfo(sec);
    }

    public static void printInfo(Employee empl){
        empl.getSalary();
        empl.getVacationDays();
        empl.getVacationForm();
    }
}
```

You can only call methods of the `Employee` class

`empl.sue()` is illegal because it is a method of the `Lawyer` class

When sending messages to an object through a reference to a superclass type, it is only legal to call methods known to the superclass.

Late Binding

- You can pass to a method many different types of `Employees` as parameters, and the method produces a behavior depending on which type is passed
- The program doesn't know which method to call until the runtime (called "**late binding**")

Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements

```
public class EmployeeMain4 {
    public static void main(String[] args) {
        Employee[] e = {new Lawyer(), new Secretary(),
                        new Marketer(), new Employee() };
        for (int i = 0; i < e.length; i++) {
            e[i].getSalary();
            e[i].getVacationDays();
             System.out.println();
        }
    }
}
```

You can store objects of different subtypes or of the superclass

You can only call methods of the `Employee` class e.g., `empl.sue()` is illegal because it is a method of the `Lawyer` class

Output:

```
I earn $40,000
I receive 3 weeks vacation
I earn $40,000
I receive 2 weeks vacation
I earn $50,000
I receive 2 weeks vacation
I earn $40,000
I receive 2 weeks vacation
```

Lawer

Secretary

Marketer

Employee

Back to our `Employee` example

- A variable can only call that type's methods, not a subtype’s

```
Employee ed = new Lawyer();
int hours = ed.getHours();   // ok; it's in Employee
ed.sue();                    // compiler error
```

- The compiler's reasoning is, variable `ed` could store any kind of `Employee`, and not all kinds know how to sue

- To use `Lawyer` methods on `ed`, we can type-cast it

```
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue();                  // ok

((Lawyer) ed).sue();              // shorter version
```

Implicit casting (upcasting)

- Upcasting example:

```
CASE 1: Employee ed1 = new LegalSecretary();
CASE 2: Secretary ed2 =  new LegalSecretary() ;
```

- Variable `ed1` is an `Employee` and a `LegalSecretary`
- Variable `ed2` is a `Secretary` and a `LegalSecretary`

- The compiler can handle the assignments since the types are compatible (a `LegalSecretary` is-a `Employee` and is-a `Secretary`)

Explicit casting (downcasting)

- Downcasting example:

```
Object obj = new Lawyer();
obj.getSalary();   // compiler error
Lawyer law = obj; // compiler error
```

- The compiler cannot automatically cast `obj` to a `Lawyer` because an `Object` is not necessarily a `Lawyer` (the compiler does not know whether `obj` is a `Lawyer` or not)

```
Lawyer law = (Lawyer) obj; // explicit casting
```

- We can now call methods of the `Lawyer` class on the `obj` object

```
((Lawyer)obj).getSalary();
```

More about casting

- The code crashes if you cast an object too far down the tree

```
Employee eric = new Secretary();
((Secretary) eric).takeDictation("hi");     // ok
((LegalSecretary) eric).fileLegalBriefs();  // exception
```

- You can cast only up and down the tree, not sideways

```
Lawyer linda = new Lawyer();
((Secretary) linda).takeDictation("hi");    // error
```

- Casting doesn't change the object's behavior. It just gets the code to compile/run.

```
((Employee) linda).getVacationForm();  // pink
```

Polymorphism problem

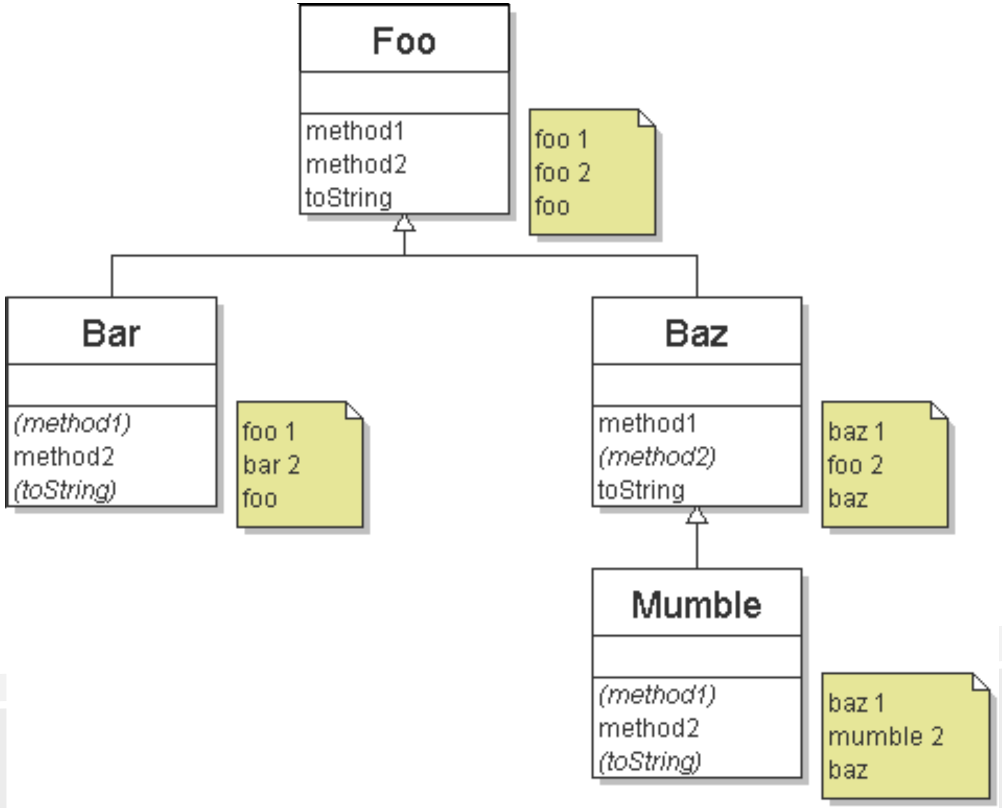
- 4-5 classes with inheritance relationships are shown
- A client program calls methods on objects of each class
- You must read the code and determine the client's output

Polymorphism problem 1

```
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");
    }
}
```

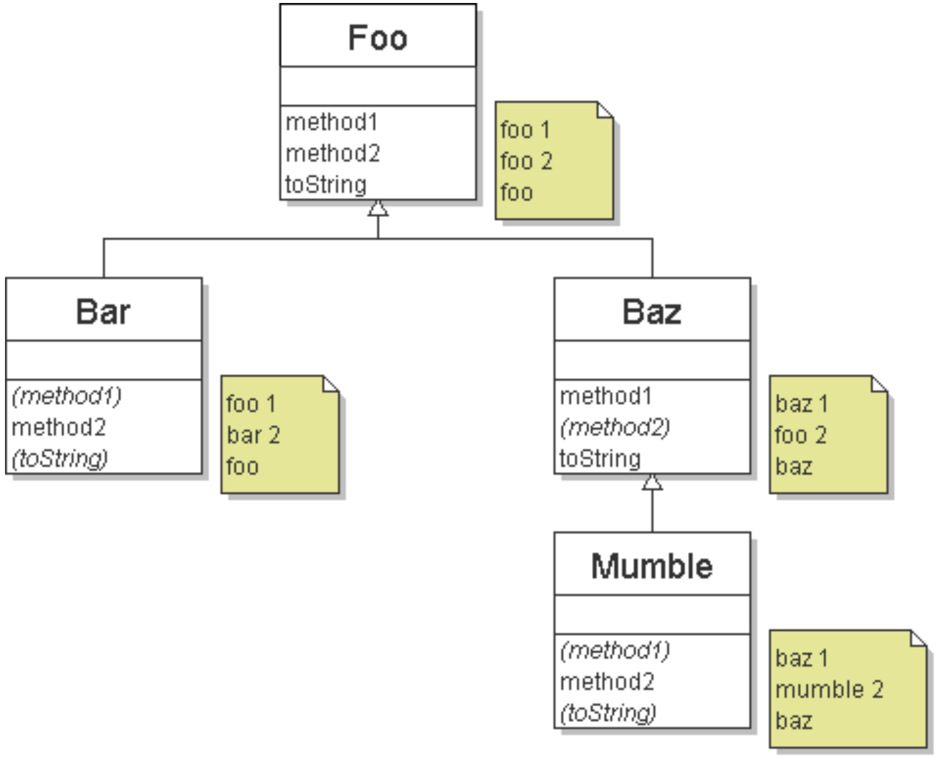
Diagramming the classes

- Add classes from top (superclass) to bottom (subclass)
- Include all inherited methods



Polymorphism problem1

- What would be the output of the following client code?



Foo

method1
method2
toString

foo 1
foo 2
foo

Bar

(method1)
method2
(toString)

foo 1
bar 2
foo

Baz

method1
(method2)
toString

baz 1
foo 2
baz

Mumble

(method1)
method2
(toString)

baz 1
mumble 2
baz

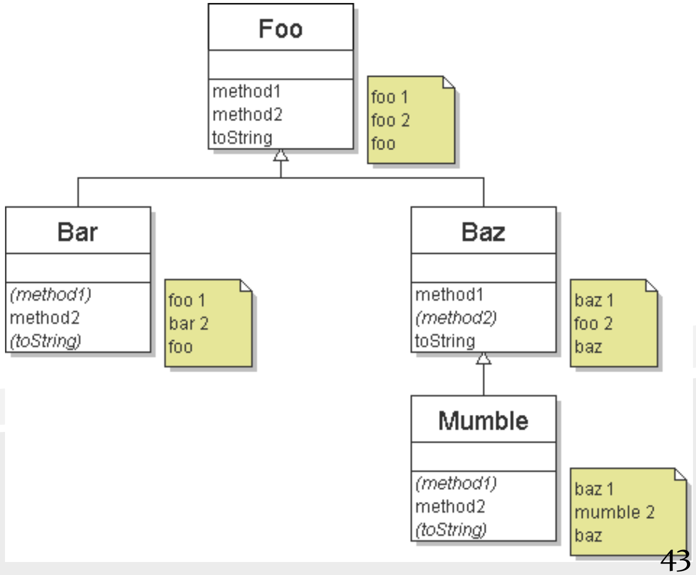
```
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Polymorphism answer1

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



Foo
method1
method2
toString
foo 1
foo 2
foo
Bar
(method1)
method2
(toString)
foo 1
bar 2
foo
Baz
method1
(method2)
toString
baz 1
foo 2
baz
Mumble
(method1)
method2
(toString)
baz 1
mumble 2
baz