Advanced Programming Techniques in Java

Casting & Polymorphism

Lecture 13

Class Objectives

- Casting & Polymorphism (9.3)

Review: Is-a relationships

- **Is-a relationship** is a hierarchical connection wh
  specialized version of another
- Every marketer **is-an** employee
- Every legal secretary **is-a** secretary

- **Inheritance hierarchy** is a set of classes connec
  share common code

Review: Overloading vs. Overridin

- What is the difference between method **overloa**
- **Overloading**: one class contains multiple methods w
  parameter signatures
- **Overriding**: a subclass substitutes its own version of
  same name and the same parameters
- Overloading lets you define a similar operation in diff
- Overriding lets you define a similar operation in differ

Review: The `super` reference

- Constructors are not inherited, even though they hav
- Yet we often want to use the parent's constructor to
- The `super` reference can be used to refer to the pa the parent's constructor
- A child’s constructor is responsible for calling the pa
- The first line of a child’s constructor should use the s constructor

- The `super` reference can also be used to reference in the parent's class

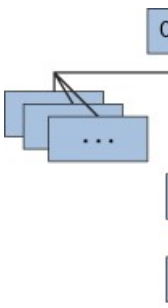
Review: One more level of inform

- Keyword `protected`
- Provides intermediate level of security b access
- Allows a member of a superclass to be
- Can be used within own class or in any
- Cannot be used by "outside" classes
- `public` methods can be used by EVER subclasses

- When might you need it? (RARELY
- If you want your fields to be `private` b
  accessor method

Inheritance and Polymorphism

- **Inheritance**: A way to form new classes based on e
  attributes/behavior
- **Polymorphism**: Ability for an object to be used as i



The `Object` class

- The `Object` class is the parent class of all the
- All classes are derived from the `Object` class (i.e.
- It defines and implement the behavior common to all

- It is defined in the `java.lang` package
- If a class is not explicitly defined to be the child
  be the child of the `Object` class

The `Object` class and inheritan

- The `Object` class contains a few useful method
- Other classes can override these methods
- e.g., `toString()`, `equals(Object obj)`

- Otherwise, the default implementation is used

The `Object` class `toString` m

- Every time we have defined `toString`, we have be
- The `toString` method in the `Object` class is def
  the name of the object's class together along with so
- All objects are guaranteed to have a `toString` me
- Thus, the `println` method can call `toString` for

`Object` variables

- You can store any object in a variable of type `Obje`

```
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

```
String s = o1.toString();       // ok
int len = o2.length();          //
error
public void checkForNull(Object o) {
    if (o == null) {
        throw new IllegalArgumentExceptio
    }
}
```

```
String line = o3.nextLine();   // error
```

- You can write methods that accept an `Object` paramete

The `Object` class `equals` me

- The `equals` method of the `Object` class returns
- We can override `equals` in any class to define eq
- The `String` class (as we've seen) defines the `equals` objects contain the same characters

`equals` and `Object`

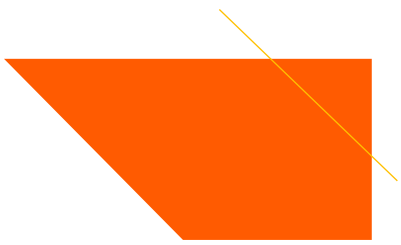
```
public boolean equals(Object name) {
    statement(s) that return a
boolean value ; }
```

- The parameter to `equals` must be of type `Objec`
- `Object` is a general type that can match any obje

- Having an `Object` parameter means *any* object c

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

Final `equals` method



Review: Data Conv

- *Assignment conversion* occurs when a value of
  another
- *Arithmetic promotion* happens automatically w
  their operands
- *Casting* is accomplished by explicitly casting a
- To cast, the type is put in parentheses in front of the val
- For example, if `total` and `count` are integers, but we wan
  can cast `total`:

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

Type cast operator

Object Casting

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

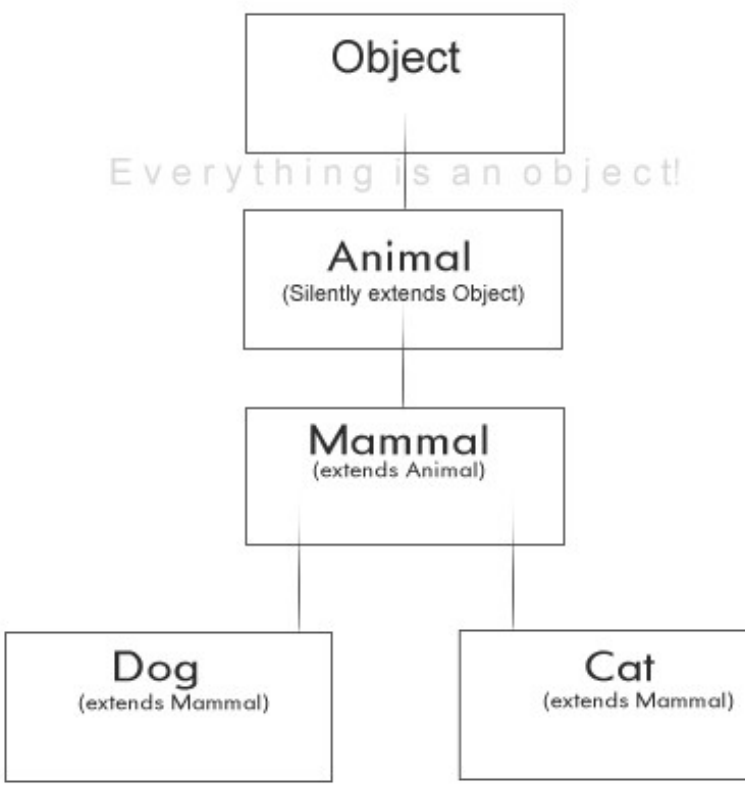
- **Upcasting**: an object of a subclass type can be trea

  Upcasting is automatic in Java (**implicit casting**)

- **Downcasting**: treating a superclass object as its re

Object Casting

- Downcasting must be specified (**explicit casting**)



by Sinipull for codecall.net

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

Upcasting

```
public class Test {
    public static vo
        Cat c = ne
        System.out
        Dog d = ne
        System.out
    }
}
```

Output:

```
100
100
```

Upcasting

- By casting an object, you are NOT actually changing labeling it differently
- `Animal cat1 = new Cat();`

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

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

- The object doesn't stop from being a `Cat`. It's s

other `Animal` and its `Cat` properties are hidden u

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
```

Upcasting and Downcasting

- Why upcasting is automatic, but downcasting must b

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

- Upcasting can never fail. But if you have a group of
  downcast them all to a `Cat`, then there's a chance, t
  `Dogs`, and process fails, by `throwing ClassCast`

Downcasting

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

- Such code passes compiling but throws "`java.lang.ClassCastException: Mammal ca`
  during running, because trying to cast a `Mammal`, w

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

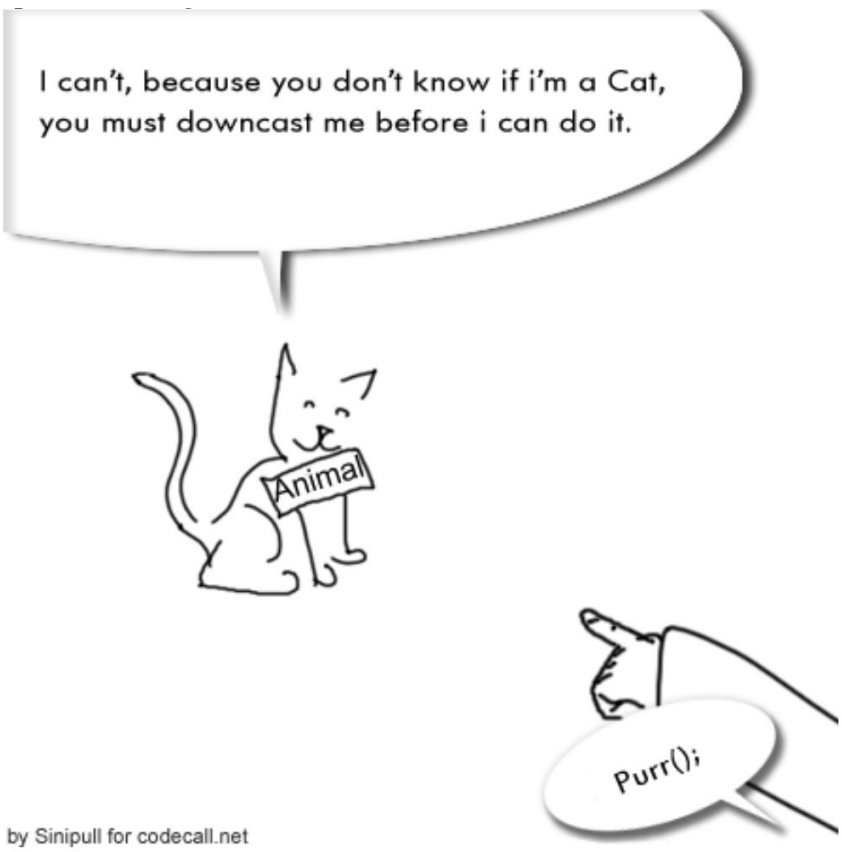
Upcasting and Downcasting

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

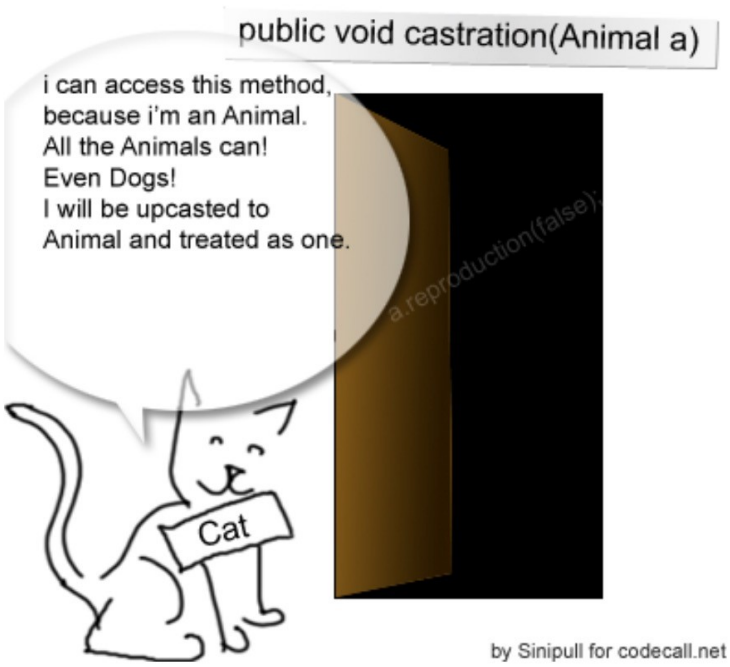
Upcasting and Downcasting

- If you upcast an object, it will lose all its properties, v
  current position
- For example, if you cast a `Cat` to an `Animal`, it wil
  `Mammal` and `Cat`.
- The data will not be lost, you just can't use it, until you do

Upcasting and Downcasting



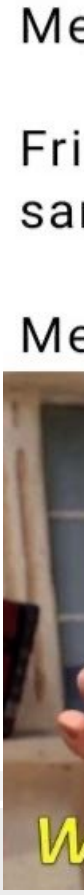
Upcasting and Downcasting



Final words on casting …

- An object of a subclass type can be treated as an ob
- This is called upcasting. Upcasting is done automatically
- An object of a superclass type can be treated as an
- This is called downcasting. Downcasting must be done ma
- Upcasting and downcasting objects are **NOT** like ca

Polymorphism



Polymorphism

- **Polymorphism** indicates the ability for the same c of objects and behave differently with each
- `System.out.println()` can print any type of d
- Each one displays in its own way on the console

Coding with polymorphism

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

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

- You can call any methods from `Employee` on `employe`
- You can not call any methods specific to `Lawyer` (e.g. `s`
- When a method is called on `employee1`, it behave

```
System.out.println(employee1.getVacationDa
System.out.println(employee1.getVacationFo
```

- But you cannot call this

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

Polymorphism and parameters

```
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 any subtype of a parameter's type

Polymorphism and parameters

```
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 any subtype of a parameter's type

Late Binding

- You can pass to a method many different types of `E`
  method produces a behavior depending on which ty
- The program doesn't know which method to call un
  binding")

Polymorphism and arrays

- Arrays of superclass types can store any subtype a

```
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
```

```
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();
        }
    }
}
```

Output:

```
I earn $40,000
```

Back to our `Employee` example

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

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

- The compiler's reasoning is, variable `ed` could sto all kinds know how to sue

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

((Lawyer) ed).sue();                // shorte
```

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

Implicit casting (upcasting)

- Upcasting example:

```
CASE 1: Employee ed1 = new LegalSecretary
CASE 2: Secretary ed2 =  new LegalSecreta
```

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

- The compiler can handle the assignments since the `LegalSecretary` is-a `Employee` and is-a `Secr`

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 `L` necessarily a `Lawyer` (the compiler does not know

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

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

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

More about casting

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

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


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

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

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

- Casting doesn't change the object's behavior. It just

Polymorphism problem

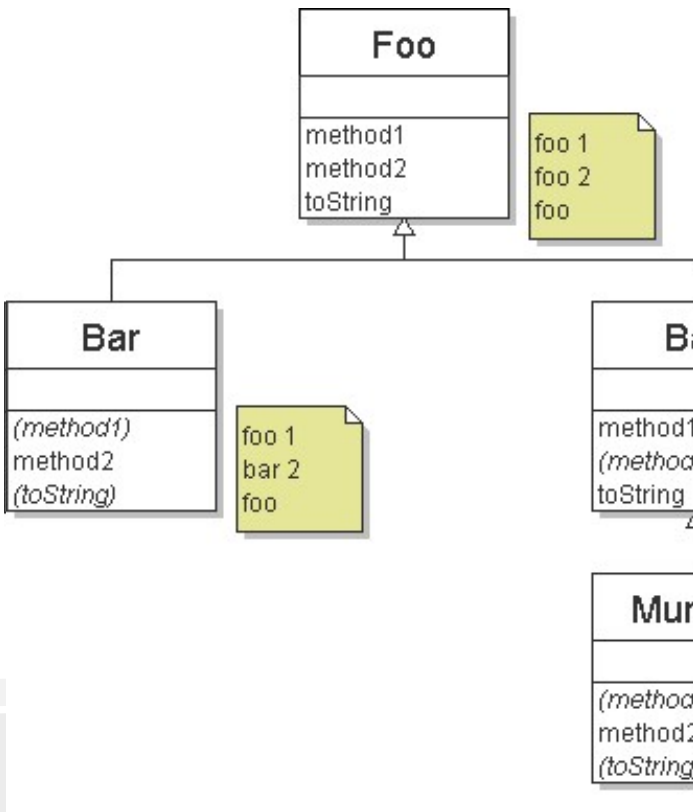
- 4-5 classes with inheritance relationships are shown
- A client program calls methods on objects of each c

- You must read the code and determine the client's c

```
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 (subcla
- Include all inherited methods



Polymorphism problem1

- What would be the output of the following client coc

```
Foo[] pity = {new Baz(), new Bar(), new Mumble
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(), n
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
```