Inheritance and Casting

A Cat is a Mammal is an Animal
Remember Casting?

- Casting converts one basic data type to another.
  - `double d = 97.2;`
  - `int i = (int) d;`
  - `char c = (char) i; //becomes letter ‘a’`

- Casting also works for complex types.
  - Can only cast from derived class to one of its ancestor (base) classes.
Objects Have More Than One Type

Suppose Cat extends Mammal extends Animal.

- Clearly a Cat is a Mammal.
- This means that an object of type Cat is also an object of type Mammal!
- So can cast a Cat object to a Mammal object.

- Clearly a Mammal is an Animal.
- This means that an object of type Mammal is also an object of type Animal!
- So can cast a Mammal object to an Animal object.

- Clearly a Cat is an Animal.
- This means that an object of type Cat is also an object of type Animal!
- So can cast a Cat object to an Animal object.
Example

- public class Cat extends Mammal {}

  Cat kitty = new Cat();
  Mammal m = kitty; // WORKS!

Why’s it work?

- A Cat is a Mammal.
- So can assign Cat object to a Mammal object.

- The casting happens implicitly.
- Could have said Mammal m = (Mammal) kitty;
Bad Example

- public class Mammal extends Animal {...}

  Animal fish = new Animal();
  Mammal furryThing = fish;  //FAILS!

- Why’s it fail?
  - A Mammal is an Animal.
  - So can assign Mammal object to an Animal object.

  - But an Animal is not necessarily a Mammal.
  - So cannot assign Animal to a Mammal.
Which are valid?

- public class Mammal {…}
  public class Dog extends Mammal {…}
  public class Cat extends Mammal {…}

- Mammal m;
  Dog fido = new Dog();
  Cat fluffy = new Cat();

  fido = m; // Compile error! A Mammal is not a Dog
  // (and m not instantiated).
  m = fido; // Valid.
  fido = (Dog) m; // Explicit casting, so compiles. But will break at
  // run time if you lied, and m wasn’t
  // assigned a Dog earlier.
  fido = (Dog) fluffy; // Compile error! Bad, bad, bad! A Dog is
  // never a Cat. Neither class inherits from
  // the other.
Available Methods

If I cast a Cat to a Mammal

- Then can only use Mammal methods.

```java
public class Mammal {
    private String hairColor;
    public String getHairColor() { … }
    public void setHairColor(String color) { … }
}

public class Cat extends Mammal {
    private String name;
    public String getName() { … }
    public void setName(String name) { … }
}

Cat fufu = new Cat();
Mammal m = fufu;
System.out.println( m.getHairColor() ); //works!
System.out.println( m.getName() ); //fails!
```
Test Yourself

**Which are valid?**

```java
public class Computer {...}
public class Desktop extends Computer {...}
public class Laptop extends Computer {...}
public class Handheld extends Laptop {...}

Desktop d1 = new Computer();
Computer c1 = new Desktop();
Computer c2 = new Handheld();
Handheld hh1 = (Handheld) c2;
Laptop lp1 = (Handheld) c1;
Laptop lp2 = (Handheld) c2;
hh1 = (Desktop) c1;

//No. A computer “is not a” desktop
//Yes. A desktop “is a” computer
```
Casting Objects Into Methods

Any method that accepts a super-class will also accept the sub-class

Example:

```java
public class Truck {
    ...
}
public class MonsterTruck extends Truck {
    ...
    public void setTruck(Truck t) {
    ...
}
}
```

MonsterTruck tonka = new MonsterTruck();
NoiseFest tooLoud = new NoiseFest();

```
tooloud.setTruck(tonka);
```

Slick! Works because tonka is a MonsterTruck which is a Truck.
Why Cast Into Methods?

- Can write one method that handles many scenarios.
  
  - public void setTruck(Truck t)
    - Works on both Trucks and MonsterTrucks.

  - If later create “public class UrbanCrawler extends Truck”, then works for UrbanCrawler also!
Planning for Future

Can plan for future by writing methods that accept the base class.

- e.g., setTruck(Truck t)
- e.g., equals(Object obj)
  - This method lives in Object class, and all classes extend Object.
  - So the existing equals method works for everything!

    Cat butternut = new Cat();
    Dog kujo = new Dog();
    if( butternut.equals(kujo) )
    {
        System.out.println("That’s very weird.");
    }

BTW, checks if equal by comparing the hashcode of two objects.
See Data Structures notes!
Cast Into Confusion?

Suppose Mammal has a `getName()` method, but is overridden in Cat and Dog.

Suppose Vet class has method that takes a Mammal.
```
public class Vet {
    public void printName(Mammal m) {
        System.out.println(m.getName());
    }
}
```

Which `getName()` will the following use?  
```
Mammal m = new Mammal();
Cat fluffy = new Cat();
Dog rover = new Dog();

Vet clinic = new Vet();
clinic/printName(fluffy); //will it use Mammal’s getName or Cat’s getName?
clinic/printName(rover); //will it use Mammal’s getName or Dog’s getName?
```
Answer

Told you that
1. If cast Cat object to Mammal, then can only use methods in Mammal.
2. Mammal has a getName() method.
3. So the object “m” can use that method.

But getName() is overridden by the Cat, so it uses the Cat’s version of getName().

Ditto for the Dog.
No Confusion: Late Binding

Also called **Dynamic Binding**

- An object always behaves the way you expect.
- So calls the method appropriate to the object.
- If “public void printName(Mammal m)” is given a Cat object, then “m.getName()” will call the Cat’s getName().

- How’s it do this? Doesn’t completely compile. At run time it figures out which method to use.
  - Hence “late” or “dynamic binding”.

- Very handy, but slows down Java code.
More Late Binding

Consider following, and look at next page.

MusicInstrument
- isLoud()
- isBrassInstrument()

Trumpet

Piano

Guitar

GlassTrumpet
- isBrassInstrument()

overrides this method in MusicInstrument
More Late Binding Code

```java
public class MusicInstrument {
    ...

    public boolean isBrassInstrument() {
        return this.brassInstrument;
    }

    public boolean isLoud() {
        return isBrassInstrument();
    }

    ...
}

public class GlassTrumpet extends Trumpet {
    ...

    /**
     * Overrides method in grandparent!
     */
    public boolean isBrassInstrument() {
        return false;
    }

    ...
}

public class Trumpet extends MusicInstrument {
    ...

    /* no overriding of parent methods */
}
```
Trumpet inst = new GlassTrumpet();
inst.isLoud();

- Uses isLoud() from MusicInstrument.
- But isLoud() in MusicInstrument uses isBrassInstrument() from GlassTrumpet!

- Late Binding at work! Waits until run time. Then decides to use methods appropriate to the object.
Key Ingredients for Late Binding

1. Child extends Parent
2. The Child has a method that overrides the same Parent method.
3. The Child is cast to a Parent.
4. Then if we call parent.overriddenMethod() it waits until the last possible moment to decide whether it should use the parent’s version of the method or the child’s version of the method.
Another Look at Late Binding

public class Tree
{
    private boolean choppedDown = false;

    public boolean waveInWind(boolean windy)
    {
        boolean wave = false;
        if(windy && !this.isChoppedDown())
        {
            wave = true;
        }
        return wave;
    }

    public boolean isChoppedDown()
    {
        return choppedDown;
    }
}

public class CherryTree extends Tree
{
    private boolean belongsToGeorgeWash = true;
    /**
     * Overrides method in parent!
     */
    public boolean isChoppedDown()
    {
        return super.isChoppedDown() ||
               belongsToGeorgeWash;
    }
}

Of course, the “this” is optional, but I want to make it clear that I am calling a method from within this class.
Another look at LateBinding In Action

Tree t = new CherryTree()
Boolean wave = t.waveInWind(true);

Late binding because has key ingredients.
1. CherryTree extends Tree
2. CherryTree overrides isChoppedDown()
3. The CherryTree is cast to the Tree
4. And we call parent.overriddenMethod().

Wait? What? Where do we call the overridden method? It is called inside of the waveInWind() method, which calls this.isChoppedDown().

Late binding because has key ingredients.
1. CherryTree extends Tree
2. CherryTree overrides isChoppedDown()
3. The CherryTree could be cast to the Tree before it is passed in! We won’t know until run time!
4. And we call parent.overriddenMethod().