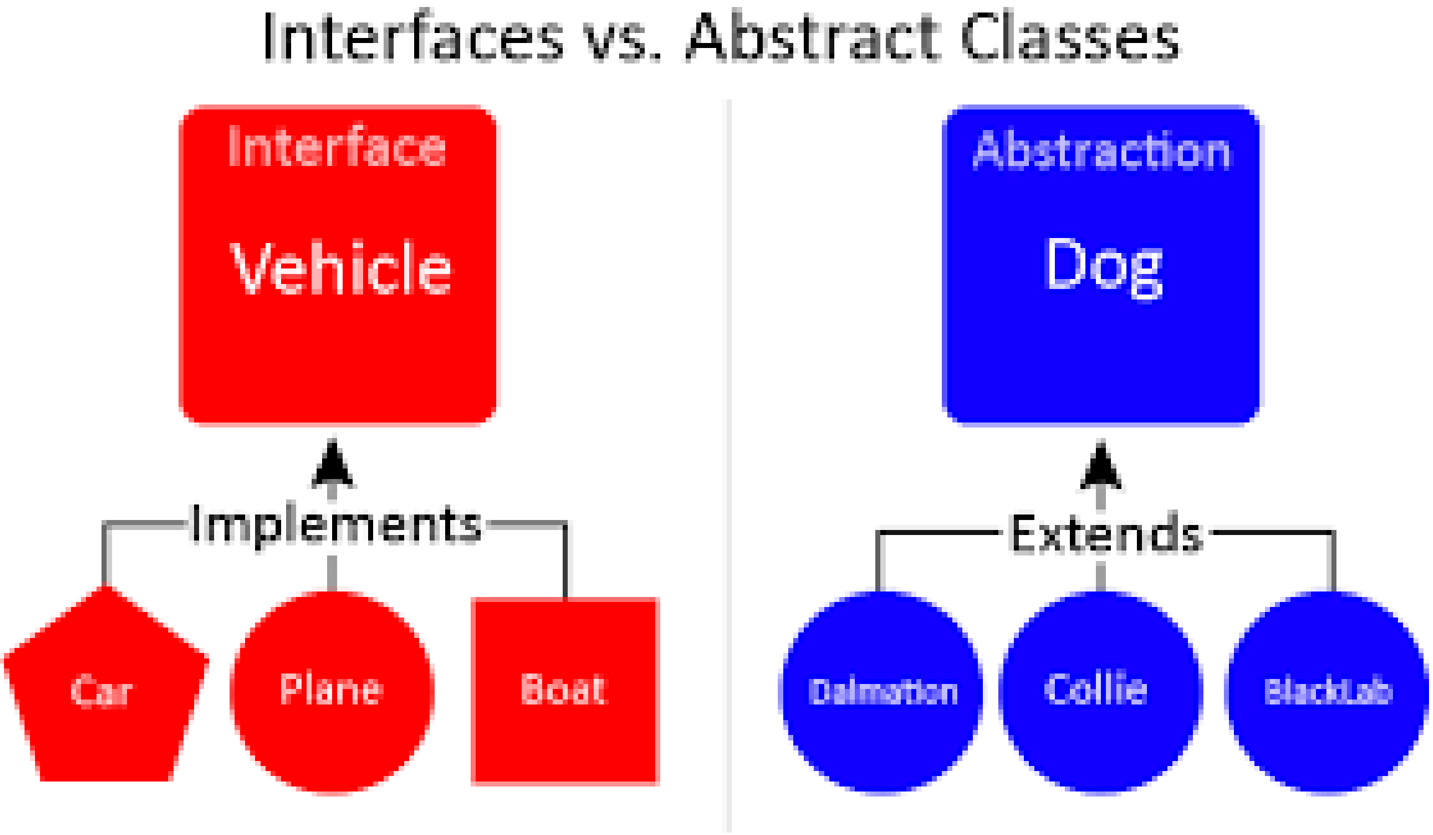
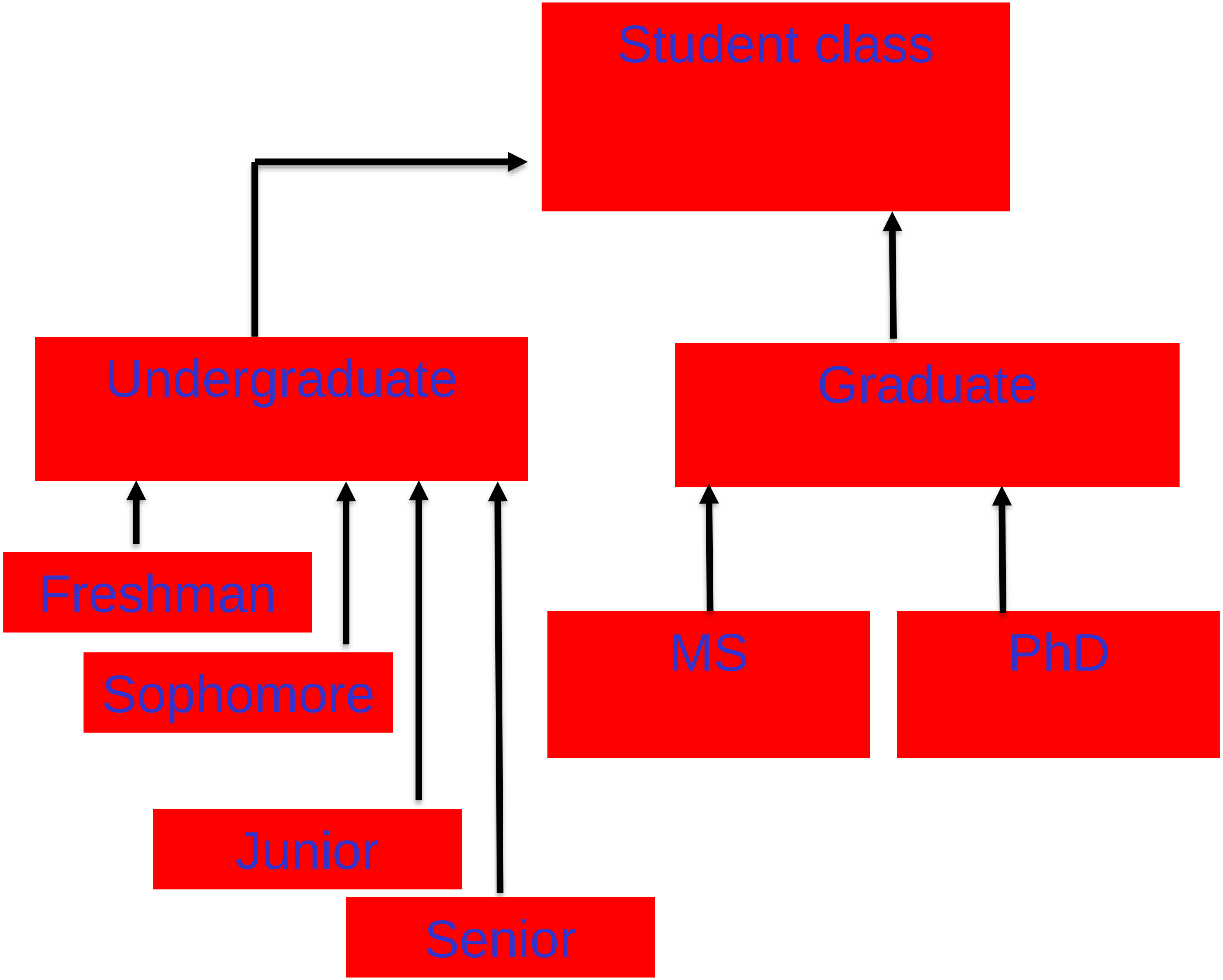
Principles of the Object-Oriented Paradigm



Student Hierarchy



Case Study

```
public class Student {
    private Name sname;
    private String sid;
    private Date dob;
    private double gpa;

    public Student() {
        System.out.println( "Student()" );
        sname = new Name( ... );
        dob = new Date( ... );
        sid = "UNDEFINED";
        gpa = 0.0;
    }

    public Student( String fname, char mi, String lname, String sid ) {

        System.out.println( "Student(" + fname + ", " + mi + ", " + lname + ", " + si
        sname = new Name( fname, mi, lname );
        this.sid = sid;
    }

    public Student( String fname, char mi, String lname, String sid, int bm, int bd,
        this(fname, mi, lname, sid);
        System.out.println( "Student(" + fname + ", ..., " + sid + "..." + gpa + ")"
        this.dob = new Date( bm, bd, by );
        this.gpa = gpa;
    }
```

Case Study

```
public class Student {
    private Name sname;
    private String sid;
    private Date dob;
    private double gpa;

    public Student() {
        System.out.println( "Student()" );
        sname = new Name( ... );
        dob = new Date( ... );
        sid = "UNDEFINED";
        gpa = 0.0;
    }

    public Student( String fname, char mi, String lname, String sid ) {

        System.out.println( "Student(" + fname + ", " + mi + ", " + lname + ", " + si
        sname = new Name( fname, mi, lname );
        this.sid = sid;
    }

    public Student( String fname, char mi, String lname, String sid, int bm, int bd,
        this(fname, mi, lname, sid);
        System.out.println( "Student(" + fname + ", ..., " + sid + "..." + gpa + ")"
        this.dob = new Date( bm, bd, by );
        this.gpa = gpa;
    }
```

Case Study

```
public class Student {
    private Name sname;
    private String sid;
    private Date dob;
    private double gpa;

    .
    .
    .

    public double calculateGPA() {
        System.out.println( "Student::calculateGPA(): " );

        return( 4.0 );
    }

    public String toString() {
        return( sname + ", (" + sid + ") " + dob + " " + gpa );
    }


} // class
```

Case Study

```
public class UndergraduateStudent extends Student {

    private String year;

    public UndergraduateStudent() {
        System.out.println( "UndergraduateStudent()" );

        year = "NON DEGREE";
    }

    public UndergraduateStudent( String fname, char mi, String lname, String
sid, int bm, int bd, int by, double gpa, String year ) {
        super( fname, mi, lname, sid, bm, bd, by, gpa );

        System.out.println( "UndergraduateStudent(" + fname + " ....  " + sid
+ " ....  " + gpa + ")" );
        this.year = year;
    }

    public void degreeClearing() {
        System.out.println( "UndergraduateStudent::degreeClearing()" );
    }

    public String toString() {
        return( super.toString() + " " + year);
    }
```

Case Study

```
public class UndergraduateStudent extends Student {

    private String year;

    public UndergraduateStudent() {
        System.out.println( "UndergraduateStudent()" );

        year = "NON DEGREE";
    }

    public UndergraduateStudent(                                        ring
sid, int bm, int bd, int by, dou
        super( fname, mi, lname,

        System.out.println( "Und                                        sid
+ " ....  " + gpa + ")" );
        this.year = year;
    }

    public void degreeClearing() {
        System.out.println( "UndergraduateStudent::degreeClearing()" );
    }

    public String toString() {
        return( super.toString() + " " + year);
    }
```

- **Inherits** the `calculateGPA` method
- **Implements** a `degreeClearing` method.
- **Overrides** the `toString` method

Case Study

```
public class GraduateStudent extends Student {

    private String undergraduateMajor;  // major in undergraduate studies
    private String specialization;      // MS specialization

    public GraduateStudent() {
        System.out.println( "GraduateStudent()" );
        undergraduateMajor = "UNKNOWN";
        specialization = "GENERAL";
    }
    public GraduateStudent( String fname, char mi, String lname, String sid,
int bm, int bd, int by, double gpa, String um ) {
        super( fname, mi, lname, sid, bm, bd, by, gpa );
        System.out.println( "GraduateStudent( ... ...  " + um + ")" );
        this.undergraduateMajor = um;
    }
    public String toString() {
        return( super.toString() + " " + undergraduateMajor + " " +
specialization );
    }
    public double calculateGPA() {
        System.out.println( "GraduateStudent::calculateGPA() " );
        return( 5.0 );
    }
    public void degreeClearing() {
        System.out.println( "GraduateStudent::degreeClearing()" );
    }
```

Case Study

```
public class GraduateStudent extends Student {

    private String undergraduateMajor;  // major in undergraduate studies
    private String specialization;      // MS specialization

    public GraduateStudent() {
        System.out.println( "GraduateStudent()" );
        undergraduateMajor = "UN
        specialization = "GENERA
    }
    public GraduateStudent( Stri                                        sid,
int bm, int bd, int by, double gp
        super( fname, mi, lname,
        System.out.println( "Grad
        this.undergraduateMajor = um;
    }
    public String toString() {
        return( super.toString() + " " + undergraduateMajor + " " +
specialization );
    }
    public double calculateGPA() {
        System.out.println( "GraduateStudent::calculateGPA() " );
        return( 5.0 );
    }
    public void degreeClearing() {
        System.out.println( "GraduateStudent::degreeClearing()" );
    }
```

- **Overrides** the `calculateGPA` method
- **Overrides** the `toString` method

Case Study

```
public class GraduateStudent extends Student {

    private String undergraduateMajor;  // major in undergraduate studies
    private String specialization;      // MS specialization

    public GraduateStudent() {
        System.out.println( "GraduateStudent()" );
        undergraduateMajor = "UNKNOWN";
        specialization = "GENERAL";
    }
    public GraduateStudent( String fname, char mi, String lname, String sid,
int bm, int bd, int by, double gpa, String um ) {
        super( fname, mi, lname, sid, bm, bd, by, gpa );
        System.out.println( "Grad
        this.undergraduateMajor =
    }
    public String toString() {
        return( super.toString()
specialization );
    }
    public double calculateGPA() {
        System.out.println( "GraduateStudent::calculateGPA() " );
        return( 5.0 );
    }
    public void degreeClearing() {
        System.out.println( "GraduateStudent::degreeClearing()" );
    }
```

- Overrides the `calculateGPA` method
- Overrides the `toString` method
- **Implements** a `degreeClearing` method

Case Study

```
public class GraduateStudent extends Student {

    private String undergraduateMajor;  // major in undergraduate studies
    private String specialization;      // MS specialization

    public GraduateStudent() {
        System.out.println( "GraduateStudent()" );
        undergradua
        specializat
    }
    public Graduate
int bm,  int bd,  int
        super( fnam
        System.out.
        this.undergraduateMajor = um;
    }
    public String toString() {
        return( super.toString() + " " + undergraduateMajor + " " +
specialization );
    }
    public double calculateGPA() {
        System.out.println( "GraduateStudent::calculateGPA() " );
        return( 5.0 );
    }
    public void degreeClearing() {
        System.out.println( "GraduateStudent::degreeClearing()" );
    }
```

Can the following call be made?

```
Student s = new GraduateStudent( .. );
System.out.println( s.toString() );
```

✓

Case Study

```
public class GraduateStudent extends Student {

    private String undergraduateMajor;  // major in undergraduate studies
    private String specialization;      // MS specialization

    public GraduateStudent() {
        System.out.println( "GraduateStudent()" );
        undergradua
        specializat
    }
    public Graduate
int bm, int bd, int
        super( fnam
        System.out.
        this.underg
    }
    public String toString() {
        return( super.toString() + " " + undergraduateMajor + " " +
specialization );
    }
    public double calculateGPA() {
        System.out.println( "GraduateStudent::calculateGPA() " );
        return( 5.0 );
    }
    public void degreeClearing() {
        System.out.println( "GraduateStudent::degreeClearing()" );
    }
```

Can the following call be made?

```
Student s = new GraduateStudent( .. );
System.out.println( s.calculateGPA() ); ✓
// method is defined in Student class and
// overridden in GraduateStudent class.
```

Case Study

```
public class GraduateStudent extends Student {

    private String undergraduateMajor;  // major in undergraduate studies
    private String specialization;      // MS specialization

    public GraduateStudent() {
        System.out.println( "GraduateStudent()" );
        undergradua
        specializat
    }
    public Graduate
int bm, int bd, int
        super( fnam
        System.out.
        this.underg
    }
    public String toString() {
        return( super.toString() + " " + undergraduateMajor + " " +
specialization );
    }
    public double calculateGPA() {
        System.out.println( "GraduateStudent::calculateGPA() " );
        return( 5.0 );
    }
    public void degreeClearing() {
        System.out.println( "GraduateStudent::degreeClearing()" );
    }
```

Can the following call be made?

```
Student s = new GraduateStudent( .. ); ✗
System.out.println( s.degreeClearing() );
// Method is defined in Undergraduate and
// Graduate class but not known in Student.
```

Case Study

```
public class GraduateStudent extends Student {

    private String undergraduateMajor;  // major in undergraduate studies
    private String specialization;      // MS specialization

    public GraduateStudent() {
        System.out.println( "GraduateStudent()" );
        undergraduateMajor = "UNKNOWN";
        specialization = "GENERAL";
    }
    public GraduateStudent( String fname                           tring sid,
int bm, int bd, int by, double
        super( fname, mi, l
        System.out.println(
        this.undergradu
    }
    public String toStri
        return( super.to
specialization );
    }
    public double calculateG
        System.out.printl    aduateStudent::calculateGPA() " );
        return( 5.0 );
    }
    public void degreeClearing() {
        System.out.println( "GraduateStudent::degreeClearing()" );
    }
```

Should this method be declared abstract in the Student class?

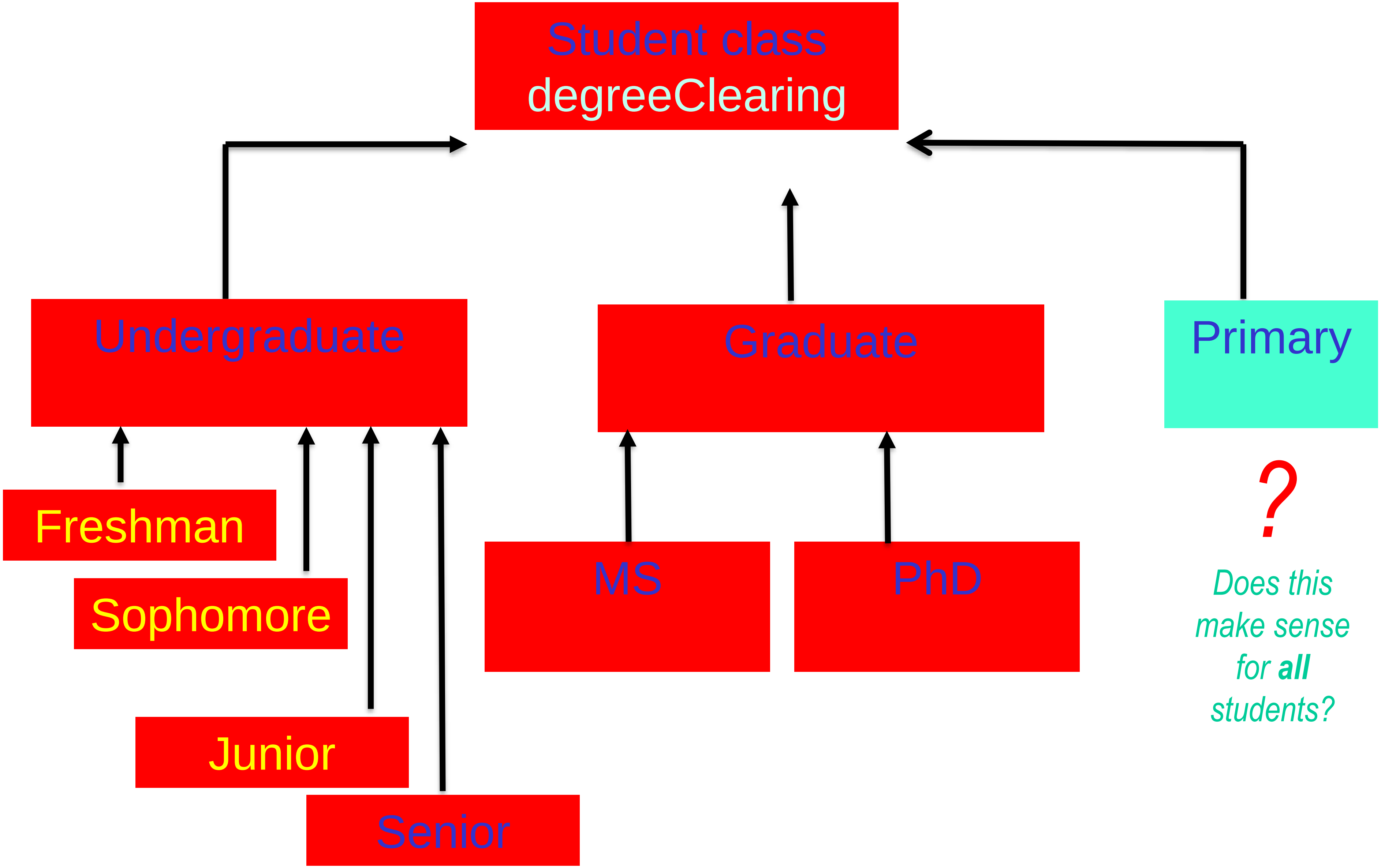
Case Study

```
public class GraduateStudent extends Student {

    private String undergraduateMajor;  // major in undergraduate studies
    private String specialization;      // MS specialization

    public GraduateStudent() {
        System.out.println( "GraduateStudent()" );
        undergraduateMajor = "UNKNO
        specialization = "GENERA
    }
    public GraduateStudent( St                                                id,
int bm, int bd, int by, doub
        super( fname, mi,
        System.out.println
        this.undergraduateM
    }
    public String toString(
        return( super.toStr
specialization );
    }
    public double calculateGPA(
        System.out.println( "      teStudent::calculateGPA() " );
        return( 5.0 );
    }
    public void degreeClearing() {
        System.out.println( "GraduateStudent::degreeClearing()" );
    }
```

An elementary student is a student!

Is this *behavior* appropriate for ***all*** students?

Student Hierarchy



Student class
degreeClearing

Undergraduate

Graduate

Primary

Freshman

Sophomore

Junior

Senior

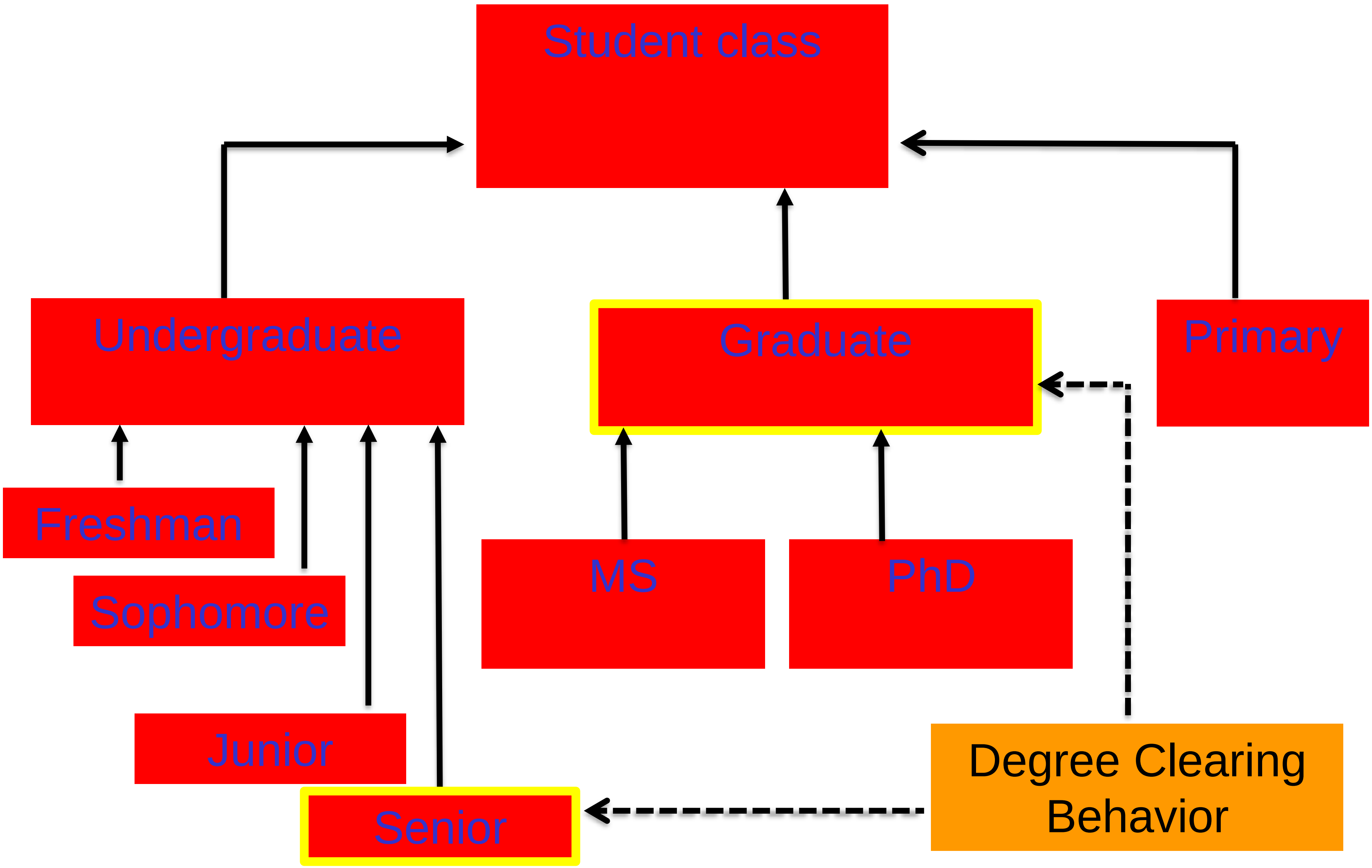
MS

PhD

?

*Does this make sense for **all** students?*

Student Hierarchy

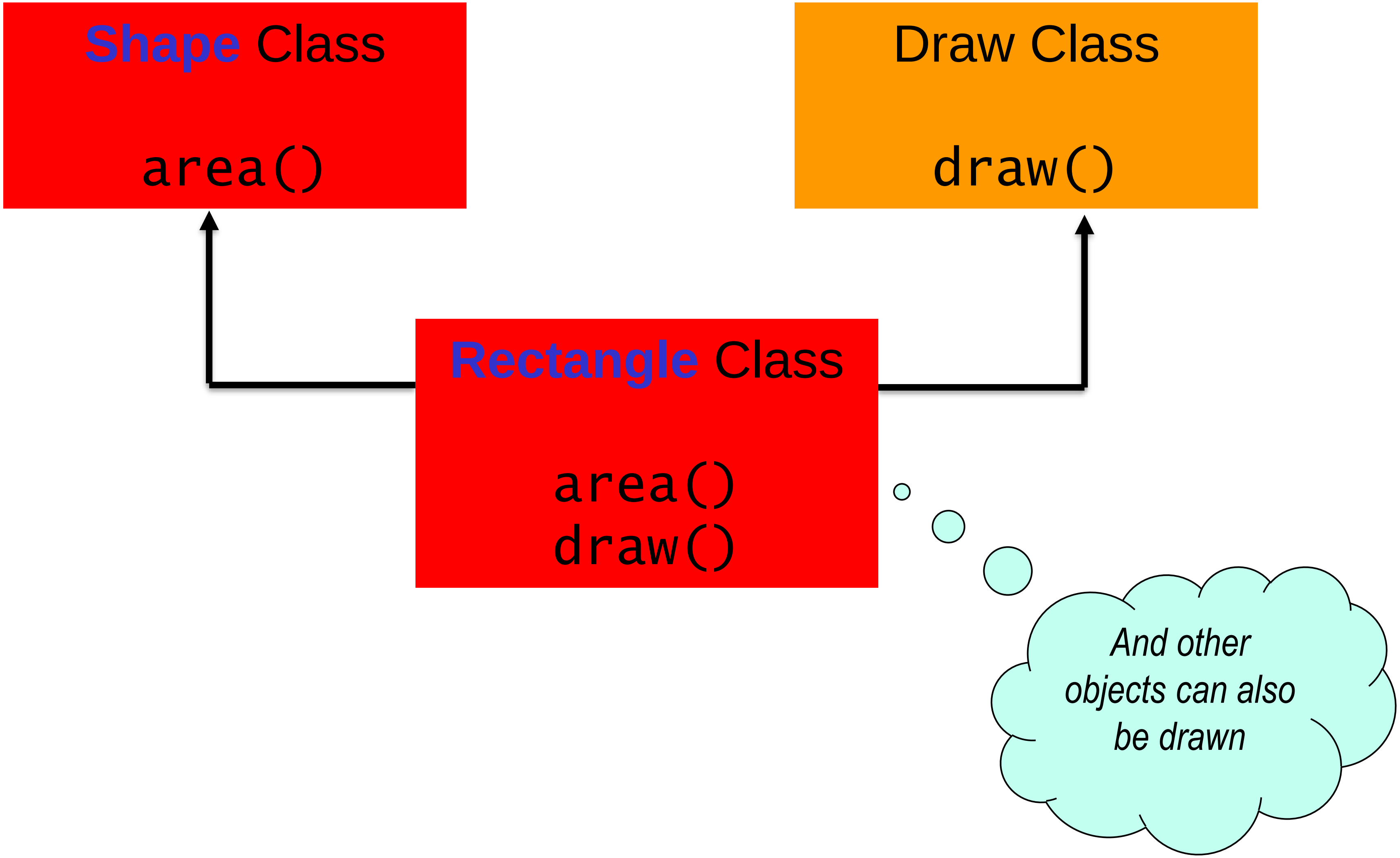


Shape, *an abstract class*

```
public abstract class Shape {
    // members common to all shapes
    String shapeName;             // name of the shape
    Point p;                      // some x, y coordinates
    Color c;                      // color

    // constructors
    Shape() {
        // assign default values to n
    }
    Shape(String name) {
        this();                   // in
        shapeName = name;
    }
    // methods common to all shap
    public String toString() {
        return( shapeName );
    }
    abstract public double area();
    abstract public void draw();


}
```

Are shapes the only objects which can be drawn?

Need for Multiple Inheritance



Shape Class

area()

Draw Class

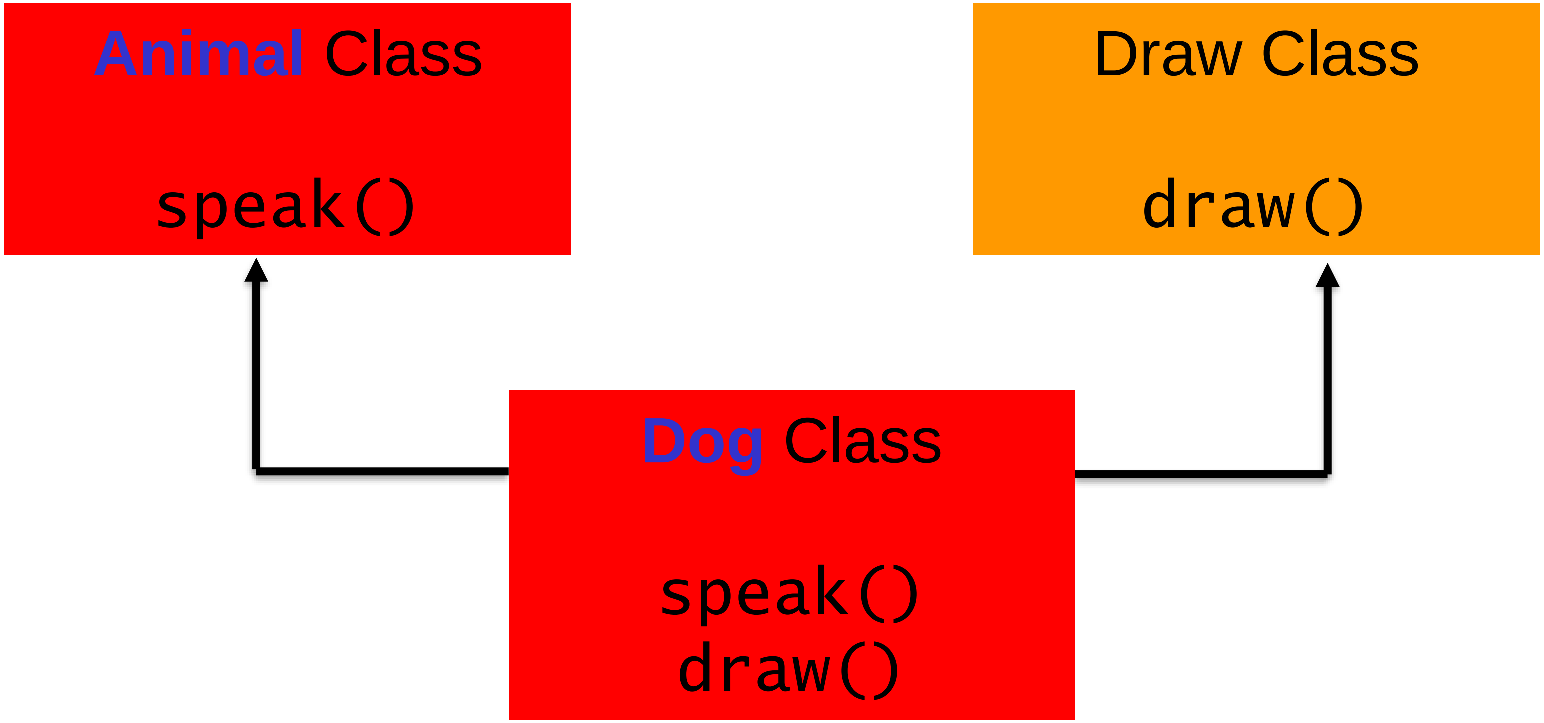
draw()

Rectangle Class

area()
draw()

And other objects can also be drawn

Need for Multiple Inheritance



Animal Class

speak()

Draw Class

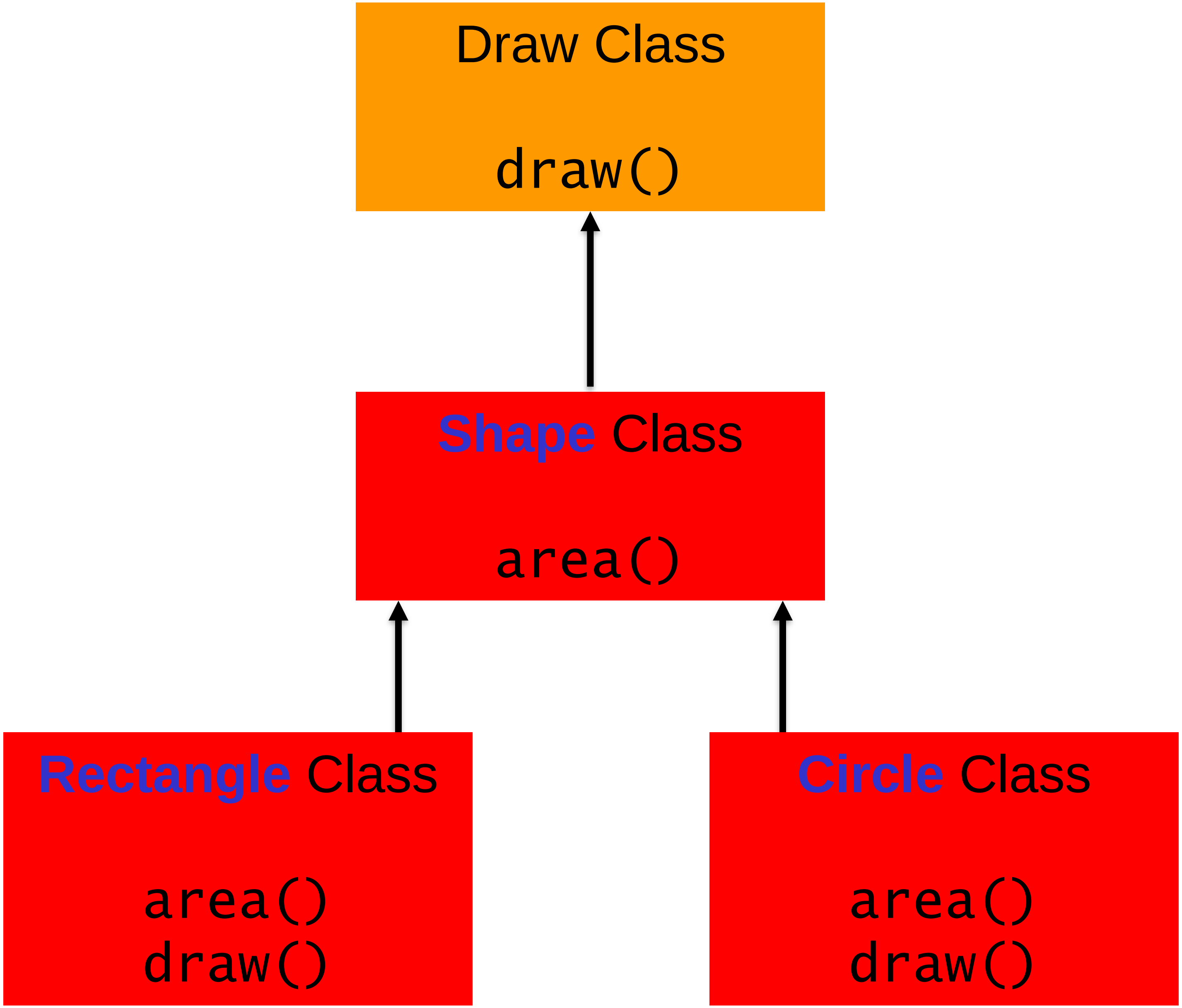
draw()

Dog Class

speak()
draw()

Problem with Single Inheritance:

Deep *Inheritance Hierarchy*



Draw Class

draw()

Shape Class

area()

Rectangle Class

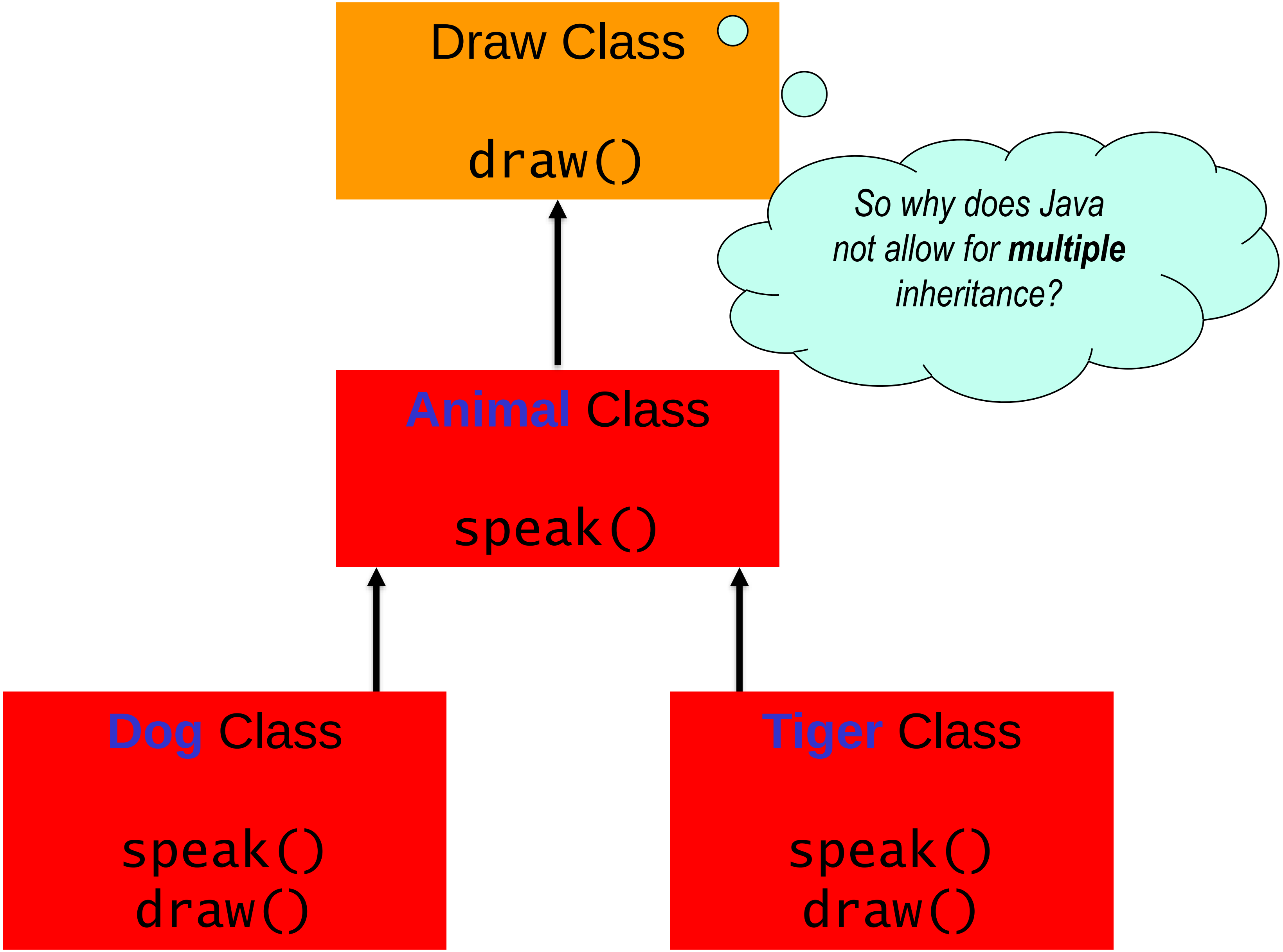
area()
draw()

Circle Class

area()
draw()

Problem with Single Inheritance:

Deep *Inheritance Hierarchy*



Draw Class

`draw()`

Animal Class

`speak()`

Dog Class

`speak()`
`draw()`

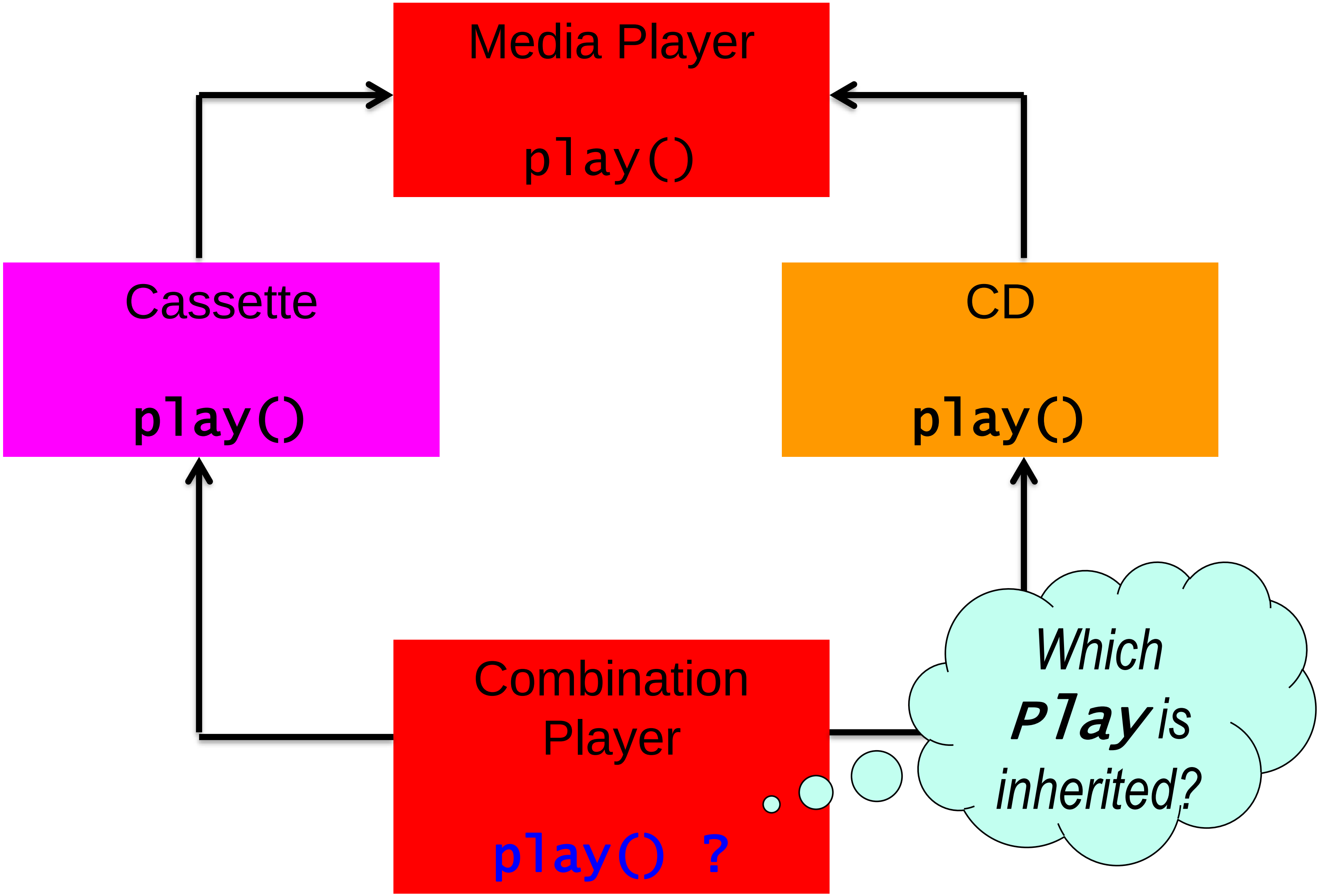
Tiger Class

`speak()`
`draw()`

So why does Java not allow for ***multiple*** *inheritance?*

Problem with Multiple Inheritance:

conflicting inherited methods



Media Player

play()

Cassette

play()

CD

play()

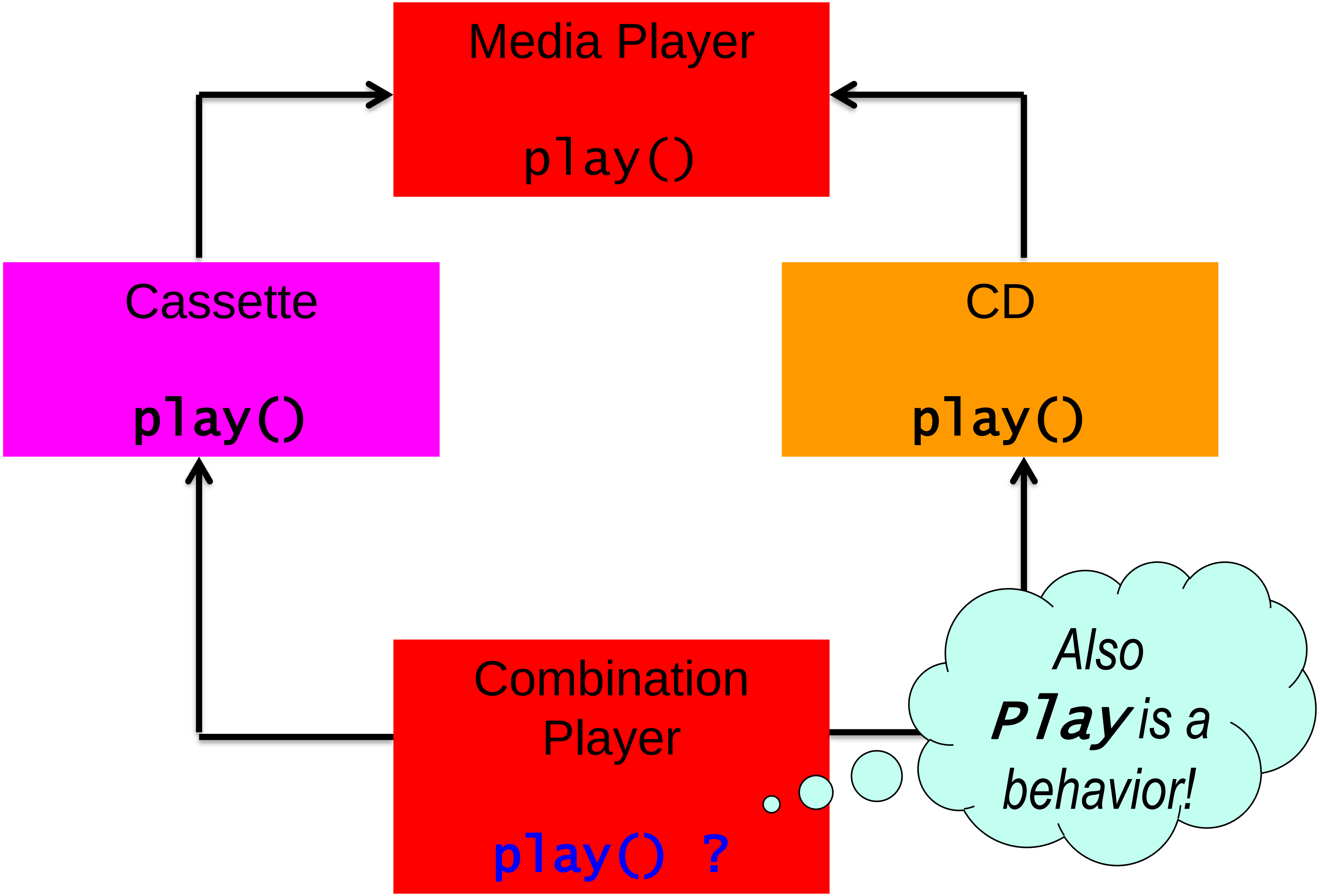
Combination Player

play() ?

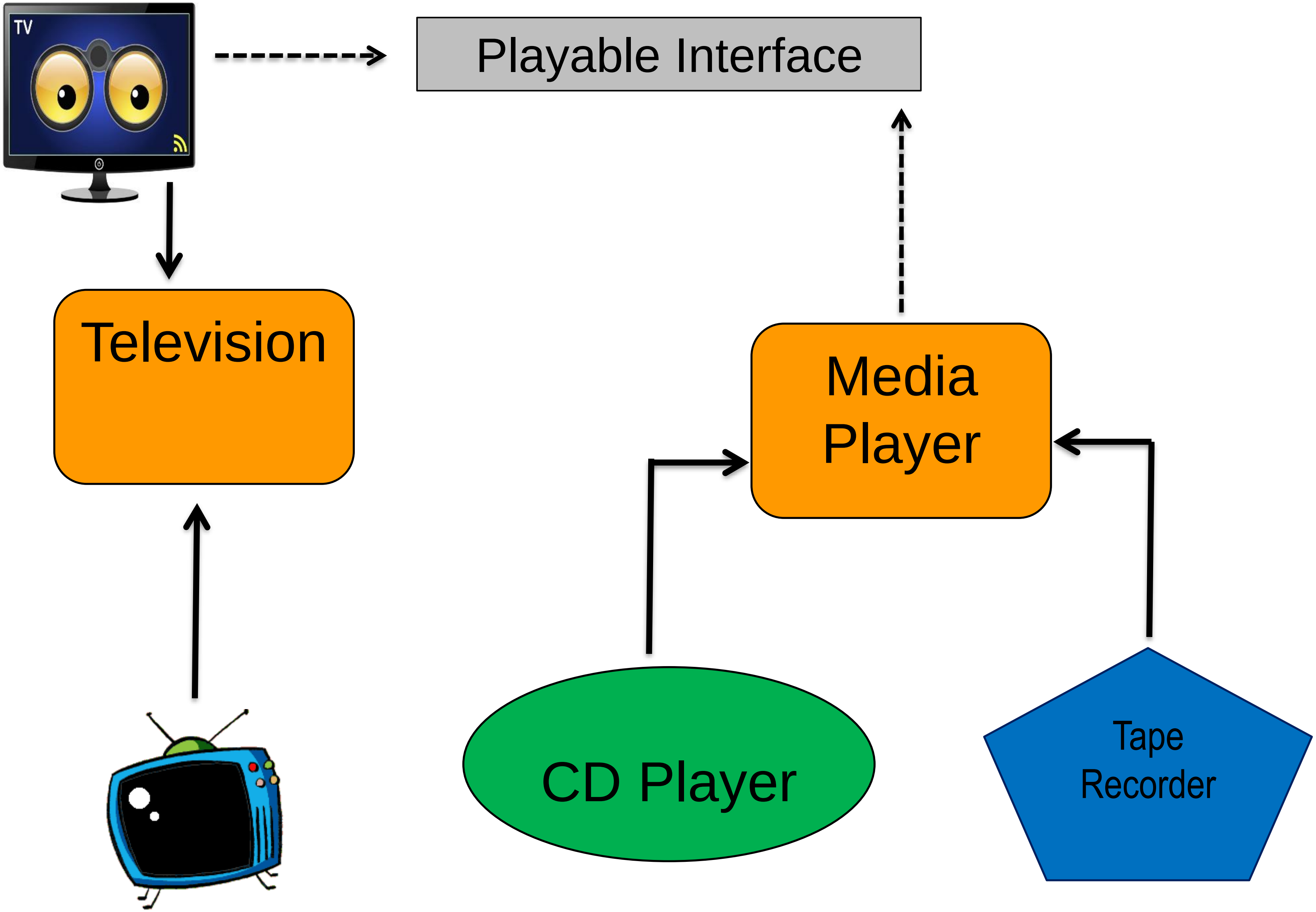
Which ***Play*** *is inherited?*

Problem with Multiple Inheritance:

conflicting inherited methods



Java Interfaces



Playable Interface

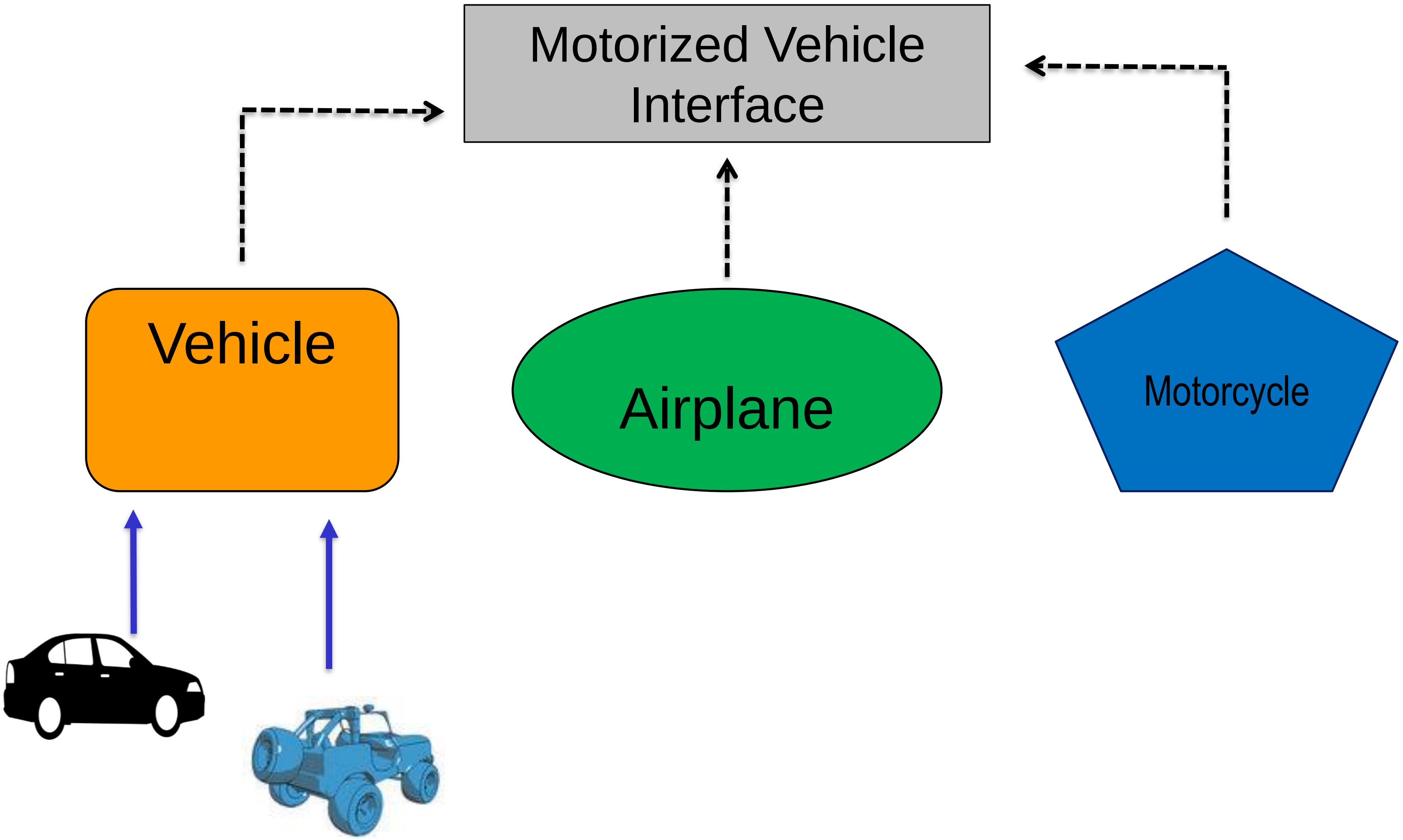
Television

Media Player

CD Player

Tape Recorder

Java Interfaces



Motorized Vehicle Interface

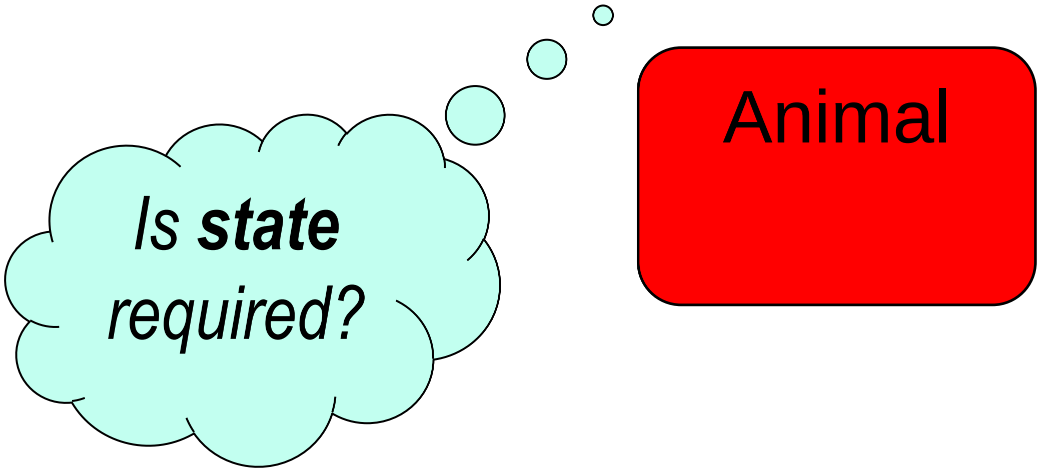
Vehicle

Airplane

Motorcycle

Problem with Multiple Inheritance:

representing behaviors as classes

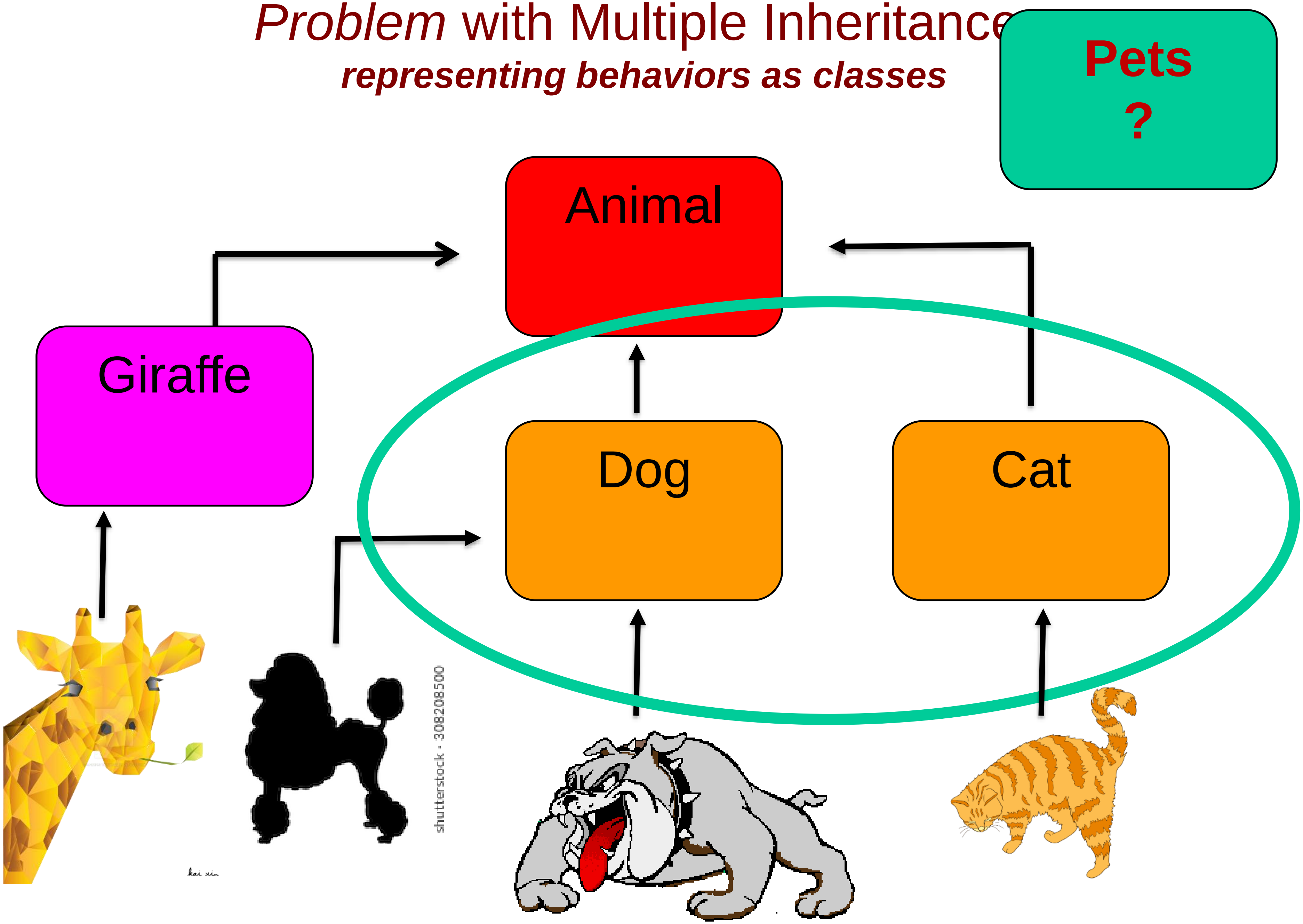


*Is **state** required?*

Animal

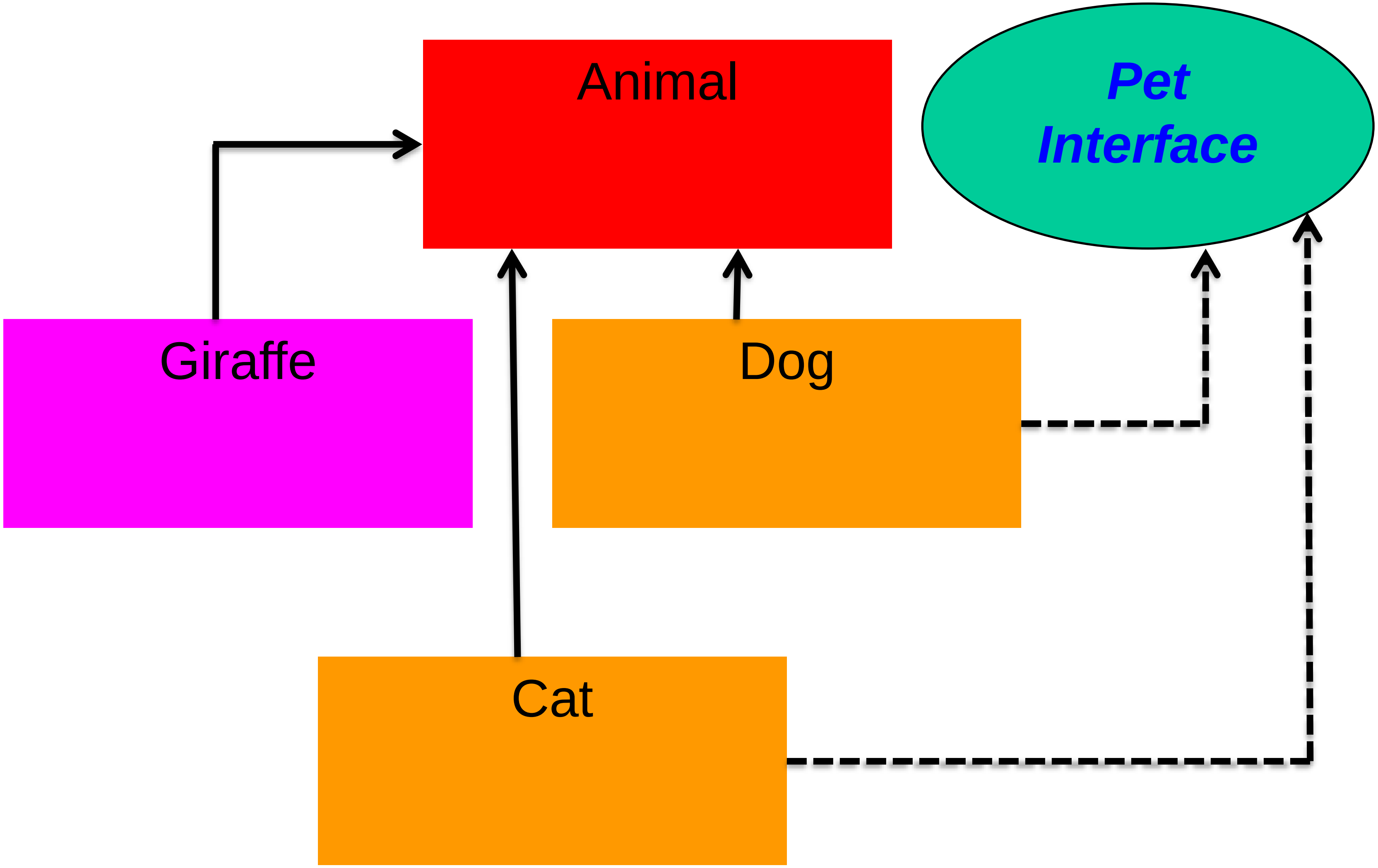
Problem with Multiple Inheritance

representing behaviors as classes



Solution:

Single** Inheritance with **multiple interfaces



Solution:

Single *Inheritance with* ***multiple interfaces***

d has all the behaviors of a ***dog***!

```
{

    Dog d = new Dog();
    Cat c = new Cat();
    Animal d = new Dog();
    Animal c = new Cat();
    Pet d = new Dog();
    Pet c = new Cat();


}
```

Solution:

Single *Inheritance with* ***multiple interfaces***

```
{

    Dog d = new
    Cat c = new Cat();
    Animal d = new Dog();
    Animal c = new Cat();
    Pet d = new Dog();
    Pet c = new Cat();

}
```

c has all the behaviors of a ***cat***!

Solution:

Single *Inheritance with* ***multiple interfaces***

```
{

    Dog d = new Dog
    Cat c = new Cat();
    Animal d = new Dog();
    Animal c = new Cat();
    Pet d = new Dog();
    Pet c = new Cat();

}
```

d and **c** have the behaviors of all ***animals!***

Solution:

Single** Inheritance with **multiple interfaces

```
{

    Dog d = new Dog();
    Cat c = new
    Animal d = n
    Animal c = new Cat();
    Pet d = new Dog();
    Pet c = new Cat();

}
```

d and **c** have the behaviors of all ***pets!***

Solution:

Single** Inheritance with **multiple interfaces

In all cases **d** references the appropriate behavior of a dog!

```
{

    Dog d = new Dog();
    Cat c = new Cat();
    Animal d = new Dog();
    Animal c = new Cat();
    Pet d = new Dog();
    Pet c = new Cat();

}
```

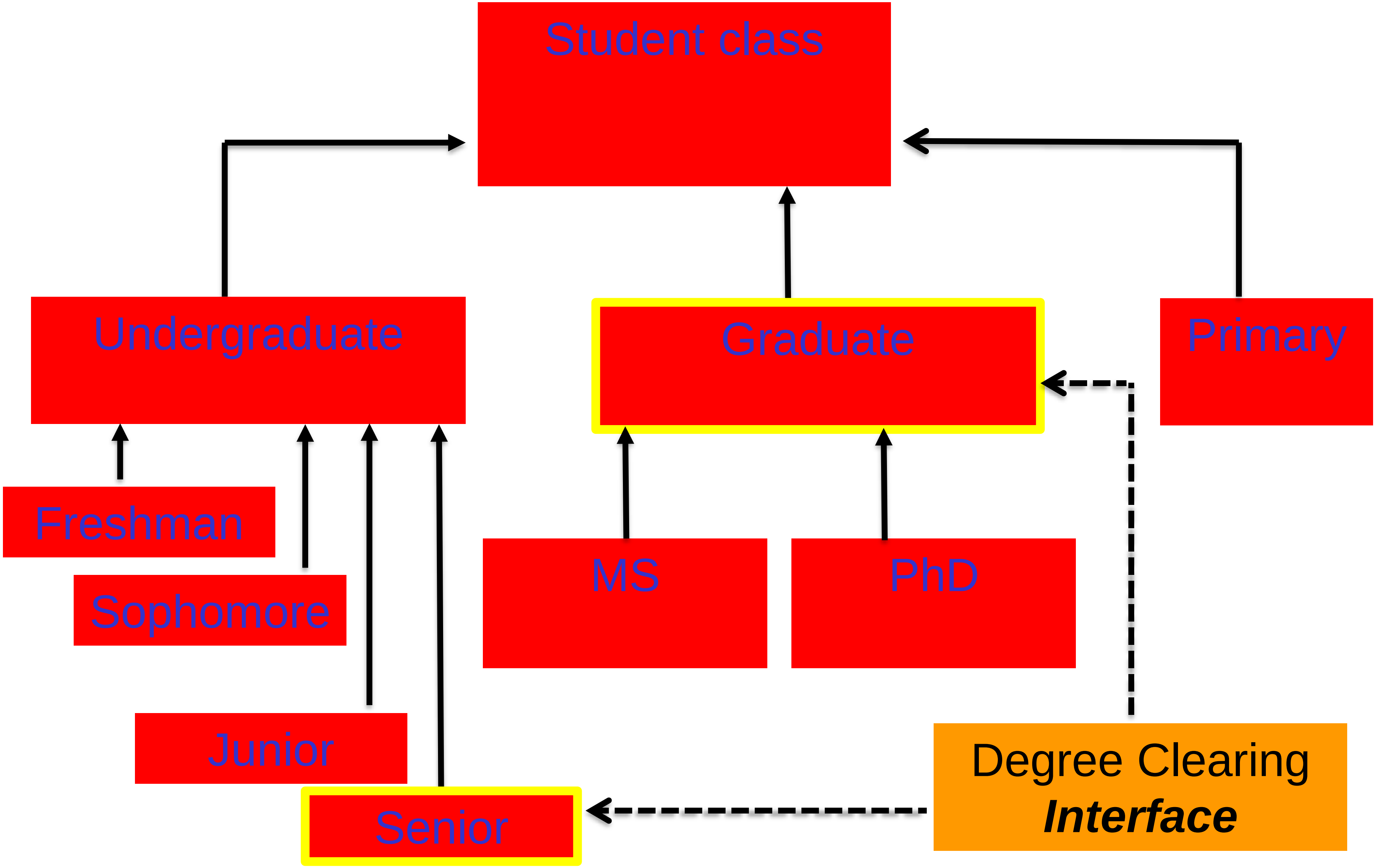
Solution:

Single *Inheritance with* ***multiple interfaces***

In all cases **c** references the appropriate behavior of a cat!

```
{

    Dog d = new Dog();
    Cat c = new Cat();
    Animal d = new Dog();
    Animal c = new Cat();
    Pet d = new Dog();
    Pet c = new Cat();

}
```

Degree clearing as an interface



Case Study

```
public class UndergraduateStudent extends Student implements degreeClearing {

    private String year;

    public UndergraduateStudent() {
        System.out.println( "UndergraduateStudent()" );

        year = "NON DEGREE";
    }

    public UndergraduateStudent( String fname, char mi, String lname, String
sid, int bm, int bd, int by, double gpa, String year ) {
        super( fname, mi, lname, sid, bm, bd, by, gpa );

        System.out.println( "UndergraduateStudent(" + fname + " .... " + sid
+ " .... " + gpa + ")" );
        this.year = year;
    }

    public void degreeClearing() {
        System.out.println( "UndergraduateStudent::degreeClearing()" );
    }

    public String toString() {
        return( super.toString() + " " + year);
    }
```

Case Study

```
public class GraduateStudent extends Student implements degreeClearing, … {

    private String undergraduateMajor;  // major in undergraduate studies
    private String specialization;      // MS specialization

    public GraduateStudent() {
        System.out.println( "GraduateStudent()" );
        undergraduateMajor = "UNKNOWN";
        specialization = "GENERAL";
    }
    public GraduateStudent( String fname, char mi, String lname, String sid,
int bm, int bd, int by, double gpa, String um ) {
        super( fname, mi, lname, sid, bm, bd, by, gpa );
        System.out.println( "GraduateStudent( ... ... " + um + ")" );
        this.undergraduateMajor = um;
    }
    public String toString() {
        return( super.toString() + " " + undergraduateMajor + " " +
specialization );
    }
    public double calculateGPA() {
        System.out.println( "GraduateStudent::calculateGPA() " );
        return( 5.0 );
    }
    public void degreeClearing() {
        System.out.println( "GraduateStudent::degreeClearing()" );
    }
```

Defining a Java Interface

Interfaces are used to represent a property that objects of all classes which implement the interface have in common. Interfaces represent a weak “is-a” relationship.

Interfaces are how Java implements a variation of multiple inheritance. A class can only extend one class but it can implement multiple interfaces! Example:

```
public class NewClass extends BaseClass
    implements Interface1, Interface2, etc. {
    ...
}
```

Defining a Java Int

Interfaces are used to represent ll
classes which implement the i
Interfaces represent a weak “is

Interfaces are how Java implem e
inheritance. A class can only exten it can
implement multiple interfaces! xample:

```
public class NewClass extends BaseClass
   implements Interface1, Interface2, etc. {
   ...
}
```

Class `NewClass` must implement all the methods specified in all the interfaces!

Defining a Java Interface

Interfaces are used to represent a property that objects of all classes which implement the interface have in common. Interfaces represent a weak “is-a” relationship.

Interfaces are how Java implements a variation of multiple inheritance. A class can only extend one class but it can implement multiple interfaces! Example:

```
public class NewClass extends BaseClass
  implements Interface1, Interface2, etc. {
  ...
}
```

Interfaces can extend other Interfaces. Example:

```
public interface NewInterface extends Interface1,
  Interface2, Interface3 etc. {
  ...
}
```

Defining a Java Interface

Interfaces are used to represent a property that objects of all classes which implement the interface have in common. Interfaces represent a weak “is-a” relationshi

Interfaces are how Java imple
inheritance. A class can only
implement multiple interfac

```
public class NewClass e
   implements Interface1
   ...
}
```

Any class which implements `NewInterface` must implement all the methods specified in all the interfaces!

Interfaces can extend ot r Interfaces. Example:

```
public interface NewInterface extends Interface1,
   Interface2, Interface3 etc. {
   ...
}
```

Interfaces vs. Abstract Classes

Interface is a contract. It cannot hold state. If we need state, must use a class.

Java Interfaces

- An interface can only contain:
  - constant (static) variable declarations
  - abstract method signatures
- Interfaces are implemented by classes and their purpose is to specify and enforce common behavior for all objects of classes which implement the interface.
- Well known Java interfaces include:
  - Comparable Interface
  - Cloneable Interface
  - ActionListener in Swing GUI

Defining a Java Interface

```
Modifier interface InterfaceName {

   /* constant variable declarations */

   /* method signatures */

}
```

Defining a Java Interface

```
Modifier interface InterfaceName {

   /* public static final assumed state */

   /* public abstract assumed for methods */

}
```

The interface must be defined in a Java file of the same name as the interface.

Example: `InterfaceName.java`

Defining a Java Interface

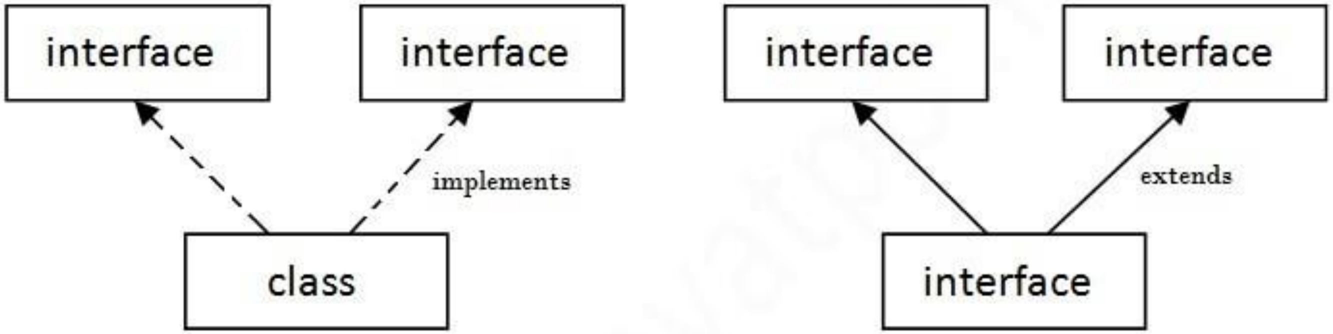
```
Modifier interface InterfaceName {

   /* public static f

   /* public abstr

}
```

The same object can exhibit different behaviors based on its type!

An interface is another way of specify

polymorphic behavior in Java.

Java
Interfaces



Multiple Inheritance in Java

Computer Science OOD
Boston University

Christine Papadakis-Kanaris

Defining a Java Interface,

the edible interface

What if we wanted to capture an operation that described the best way to eat something. Let’s say we wanted objects to have a method `howToEat()` that represented the best way that specific types of objects could be eaten.

Example:

- apples are best eaten when b
- oranges are best eaten whe
- chicken is best eaten whe

Should this be a method in Java’s Object class?

Defining a Java Interface,

the edible interface

What if we wanted to capture an operation that described the best way to eat something. Let’s say we wanted objects to have a method `howToEat()` that represented the best way that specific types of objects could be eaten.

Example:

- apples are best eaten when b
- oranges are best eaten whe
- chicken is best eaten whe

No! All classes extend the object class, but not all objects are edible objects!

Defining a Java Interface,

the edible interface

What if we wanted to capture an operation that described the best way to eat something. Let's say we wanted objects to have a method `howToEat()` that represented the best way that specific types of objects could be eaten.

Example:

- apples are best eaten when b
- oranges are best eaten whe
- chicken is best eaten whe

How about an edible abstract class?

Is there state associated with this class?

Defining a Java Interface,

the edible interface

What if we wanted to capture an operation that described the best way to eat something. Let’s say we wanted objects to have a method `howToEat()` that represented the best way that specific types of objects could be eaten.

Example:

- apples are best eaten when b
- oranges are best eaten whe
- chicken is best eaten whe

Edible is not really a type of object. It is a *property* that objects of *some* types share.

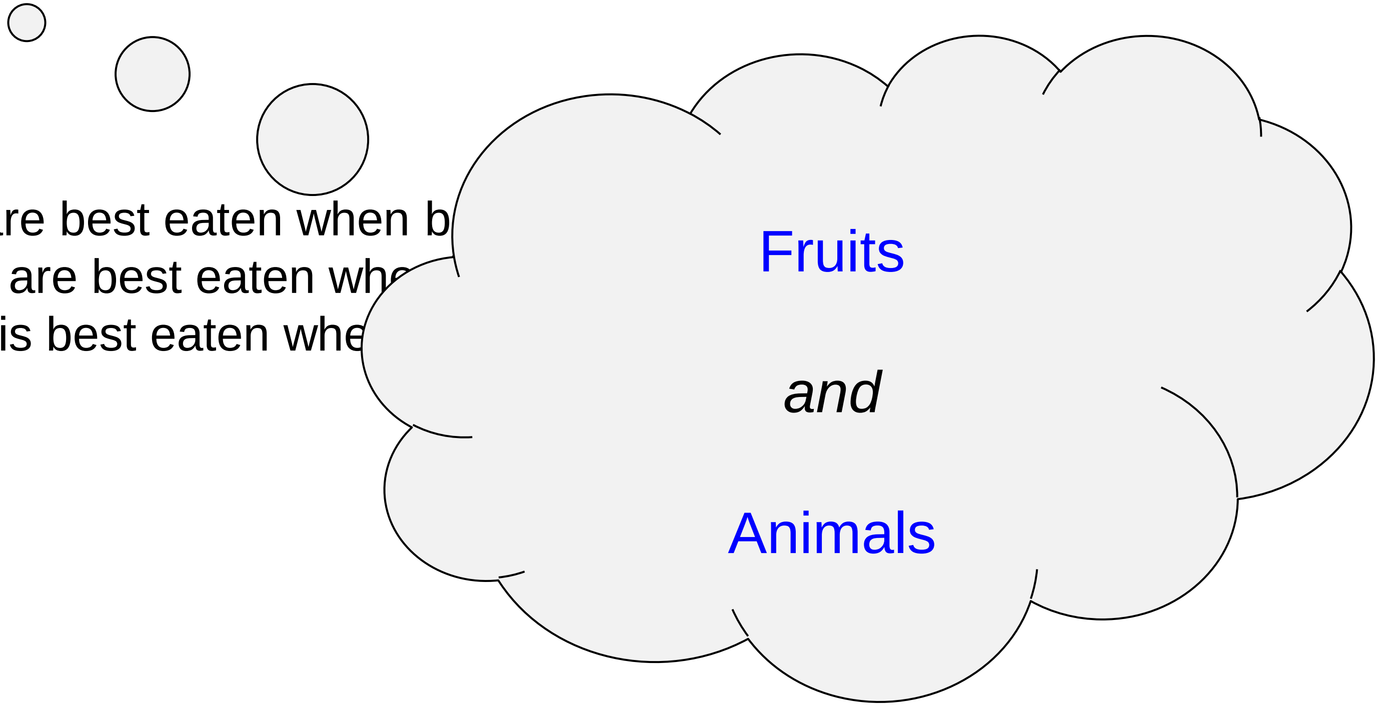
Defining a Java Interface,

the edible interface

What if we wanted to capture an operation that described the best way to eat something. Let’s say we wanted objects to have a method `howToEat()` that represented the best way that specific types of objects could be eaten.

Example:

- apples are best eaten when b
- oranges are best eaten whe
- chicken is best eaten whe



Fruits

and

Animals

The *edible* interface

```
public interface Edible {
    /* method that describes how
     * the object is eaten.
     */
     public abstract String howToEat();
}
```

```
public class Fruit
   implements Edible {

    /* Common data attributes */

    /* Constructors */

    /* Methods */

    /* must include the method
       howToEat even as an
       abstract method */
}
```

The *edibl*

```
public interface Ed
    /* method tha
     * the object
     */
     public abstra
}
```

As the interface is implemented in the superclass, all subclasses must provide the implementation!

```
public abstract class Fruit
   implements Edible {

    /* Common data attributes */

    /* Constructors */

    /* Methods */



}
```

```
public class Apple
   extends Fruit {
       /* must provide an
          implementation of
          howToEat
       */
}
```

```
public class Orange
   extends Fruit {
       /* must provide an
          implementation of
          howToEat
       */
}
```

The *edible* interface

```
public interface Edible {
    /* method that describes how
     * the object is eaten.
     */
     public abstract String howToEat();
}
```

```
public abstract class Animal
   implements Edible {

    /* Common data attributes */

    /* Constructors */

    /* Methods */




}
```

```
public class Chicken
   extends Animal {
   /* must provide an
      implementation of
      howToEat
   */
}
```

```
public class Tiger
   extends Animal {
   /* provide an
      implementation of
      howToEat ???
   */
}
```

The *edible* interface

```
public interface Edible {
    /* method that describes how
     * the object is eaten.
     */
     public abstract String howToEat();
}
```

```
public abstract class Animal  {


   /* Common data attributes */

   /* Constructors */

   /* Methods */



}
```

```
public class Chicken
   extends Animal
   implements Edible {
   /* must provide an
      implementation of
      howToEat
   */
}
public class Tiger
   extends Animal {
   /* no implementation is
      required.
   */

}
```

The *Pet* interface

```
public interface Pet {
    /* method that describes how
     * the is cared for.
     */
     public abstract String howToCare();
}
```

```
public abstract class Animal  {


    /* Common data attributes */

    /* Constructors */

    /* Methods */




}
```

```
public class Rabbit
   extends Animal
   implements Pet {
   /* must provide an
      implementation of
      howToCare
   */
}
```

The *Pet* interface

```
public interface Pet {
    /* method that describes how
     * the is cared for.
     *
     p                        wToCare();
}
```

```
public abst



   /* Comm


   /* Cons


   /* Meth




}
```

```
 class Rabbit
ends Animal
lements Pet {
must provide an
implementation of
howToCare
```

The *Pet* interface

```
public interfac
    /*

     *
     pu                                     wToCare();
}
```

Can only **extend** one class but can ***implement*** multiple interfaces!

```
public abstract class Animal  {


    /* Common data attributes */

    /* Constructors */

    /* Methods */




}
```

```
public class Rabbit
    extends Animal
    implements Pet, Edible {
    /* must provide an
       implementation of
       howToCare
    */

    /* must provide an
       implementation of
       howToEat
    */


}
```

The *edible* interface

```
public class TestEdible {

   public static void main( String [] a ) {
      Object[] objects = { new Tiger(),
                           new Chicken();
                           new Apple();
                         }

      for (int i = 0; i < objects.length i++ ) {
         // call the hotToEat method on each
         System.out.println( objects[i].howToEat() );
      }

   } // main()

} // TestEdible
```

The *edible* interface

```
public class TestEdible {

   public static void main(
      Object[] objects

                  }

      for (int i = 0; i < objects.length i++ ) {
         // call the hotToEat method on each
         System.out.println( objects[i].howToEat() );
      }

   } // main()

} // TestEdible
```

Calling the method on an Object will cause a **compiler** Error!

The *edible* inte

The object class does not know about the Edible interface!

```
public class TestEdible {

   public static void main( St
      Object[] objects = { new Tiger();
                           new Chicken();
                           new Apple();
                         }

      for (int i = 0; i < objects.length i++ ) {
         // call the hotToEat method on each
         System.out.println( objects[i].howToEat() );
      }

   } // main()

} // TestEdible
```

The *edible* inte

The method howToEat was not inherited from the Object class.

```
public class TestEdible {

   public static void main( St
      Object[] objects = { new Tiger();
                           new Chicken();
                           new Apple();
                         }

      for (int i = 0; i < objects.length i++ ) {
         // call the hotToEat method on each
         System.out.println( objects[i].howToEat() );
      }

   } // main()

} // TestEdible
```

The *edible* interface

```
public class TestEdible {

   public static void main( String [] a ) {
      Object[] objects = { new Tiger(),
                           new Chicken();
                           new Apple();
                         }

      for (int i = 0; i < objects.length i++ ) {
         // call the hotToEat method
         System.out.println(((Edible) o[i]).howToEat());
      }

   } // main()

} // TestEdible
```

Cast the object to be an *object* of type Edible, but..

The *edible* interface

```
public class TestEdible {

   public static void main(
      Object[] objects =

                         }

      for (int i = 0; i < objects.length i++ ) {
         // call the hotToEat method
         System.out.println(((Edible) o[i]).howToEat());
      }

   } // main()

} // TestEdible
```

… now a run time error because Tiger is not an Edible object!

The *edible* interface

```
public class TestEdible {

   public static void main(
      Object[] objects =



                         }


      for (int i = 0; i < objects.length i++ ) {
         if ( objects[i] instanceOf Edible )
            System.out.println(((Edible) o[i]).howToEat());
      }

   } // main()

} // TestEdible
```

Use the `instanceOf` method **or** …

The *edible* interface

```
public class TestEdible {

   public static void main( String [] a ) {
      Edible[] objects = { new Tiger(),
                           new Chicken();
                           new Apple();


                                         th i++ ) {
                                          Edible )
                              ((Edible) o[i]).howToEat());
      }

   } // main()

} // TestEdible
```

Declare these to be Edible
objects and the compiler
will catch the Error!