COMP 6481 PROGRAMMING AND PROBLEM SOLVING

Chap 8 - Polymorphism and Abstract Classes

Just Checking : True or False?...

Which of the following is true regarding Java classes?

A. Overriding is when a derived class overloads a method from the base class
B. All classes must have 1 child (derived or extended) class but may have any number of parent classes.
C. A derived class is a class defined by adding instance variables and methods to an existing class.
D. Private methods of the base class are not available for use by derived classes.
E. All classes can have either 0 or 1 parent class and any number of children (derived or extended) classes.
F. When you define a derived class, you give the added instance variables and the added methods as well as all the methods from the base class.

Introduction to Polymorphism

There are 3 main programming mechanisms that constitute object-oriented programming (OOP)

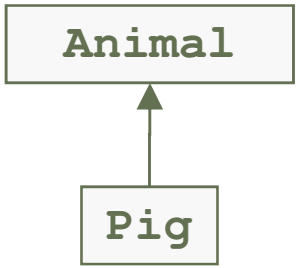
1. Encapsulation
2. Inheritance
3. Polymorphism

Polymorphism is the ability to associate many meanings to one method name

- It does this through a special mechanism known as *late binding* or *dynamic binding*

Upcasting & Downcasting

Let's say I have this hierarchy:



- Can you do this?

```
Animal a;
Pig p = new Pig();
a = p;   // OK?
```

- it's called upcasting
- why?/why not?

- Can you do this?

```
Animal a = new Animal();
Pig p;
p = a;   // OK?
```

- it's called downcasting
- why?/why not?

Example: Up.java

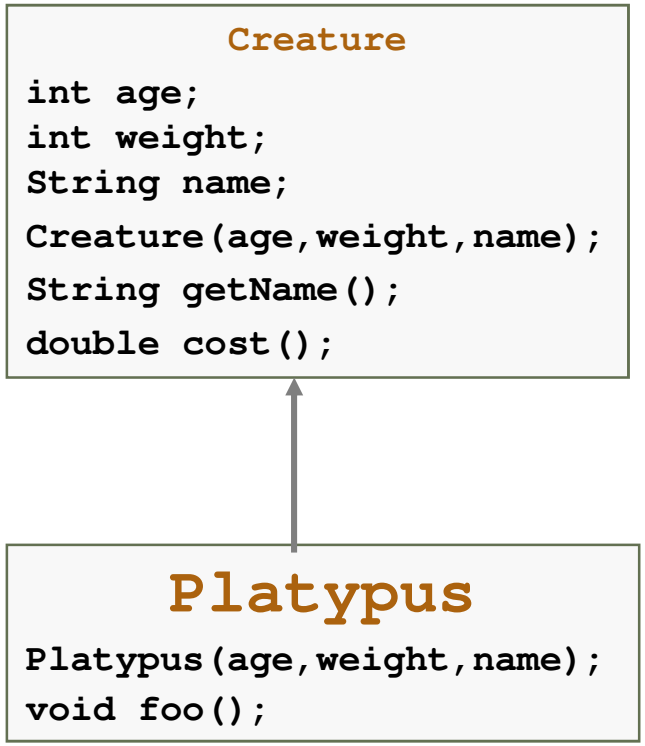
```
class Creature { // the parent class
  private int age;
  private int weight;
  private String name;

  public Creature(int age, int weight, String name){
        this.age = age;
        this.weight = weight;
        this.name = name;
  }
  public String getName(){
        return (name);
  }
  public double getCost(){
        return (weight * age) / 2;
  }
}
```

Creature

```
int age;
int weight;
Creature( ... );
String name;
String getName();
double cost();
```

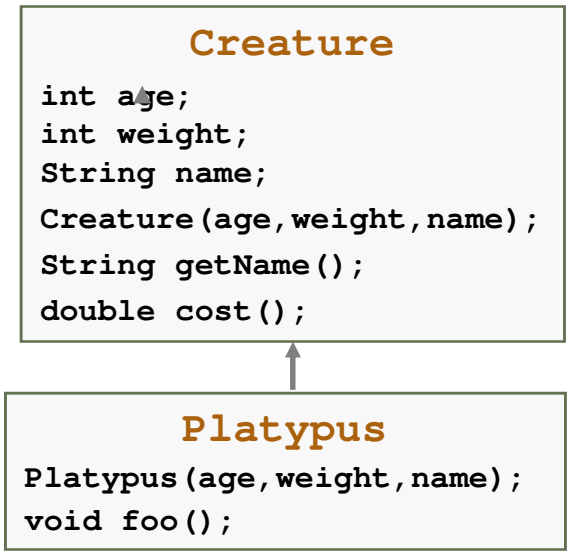
Example: Up.java

```
class Platypus extends Creature { // child class

  public Platypus(int age, int weight, String name){
     super(age, weight, name);
   }

  public void print_foo(){
       System.out.println("foo");
   }
}
```



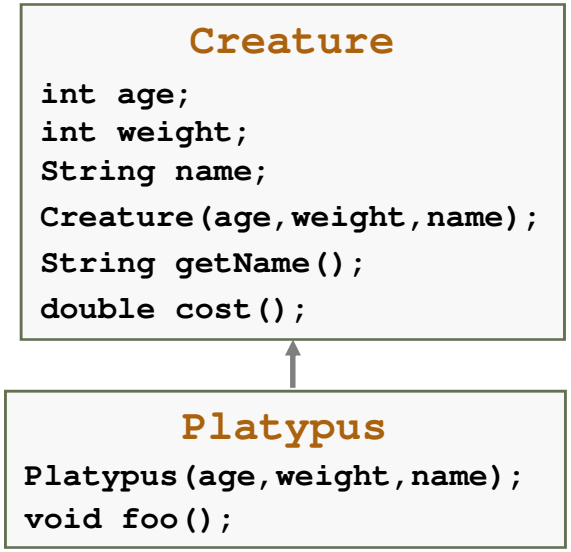
Example: Up.java

```
public class Up {
   static void show_animal_cost(Creature myCreature) {
      System.out.println("I'm " + myCreature.getName() +
                         and my cost is " + myCreature.getCost());
   }
-------------------------------------------------------------------
   public static void main(String[] args) {
     Creature myCreature1 = new Creature(4,60,"Jellybean");
     show_animal_cost(myCreature1);

     // This works because of upcasting
     Creature myCreature2 = new Platypus(44,17,"Natasha");
     show_animal_cost(myCreature2);

         …
   }
}
```



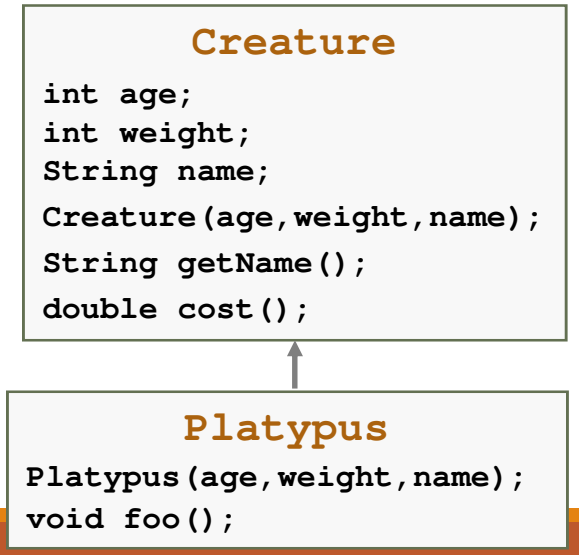
Example: Up.java

```
public class Up {
   static void show_animal_cost(Creature myCreature) {
      System.out.println("I'm " + myCreature.getName() +
                         and my cost is " + myCreature.getCost());
   }
-------------------------------------------------------------------
   public static void main(String[] args) {
     Creature myCreature1 = new Creature(4,60,"Jellybean");
     show_animal_cost(myCreature1);
          …
     // This will not even compile
     Platypus myCreature3 = new Creature(44,17,"Wally");
     show_animal_cost(myCreature3);

   }
}
```



Example: Up.java

```
public class Up {

   …

   // This compiles but it will produce a class
   // cast exception at run-time

   Platypus myCreature4=(Platypus)new Creature(44,17,"Wally");

   //show_animal_cost(myCreature4);


   // This is a very peculiar example but it shows
   // that downcasts are possible.

   Platypus myPlatypus1 = new Platypus(4, 60, "Peeps");

   Creature myCreature5 = myPlatypus1;


  Platypus myPlatypus2 = (Platypus)myCreature5;

   myPlatypus2.print_foo();

   show_animal_cost(myPlatypus2);

 }

}
```



Binding

Binding = The process of associating a method definition with a specific method call

They are 2 basic forms:

- static binding - also called early binding
  - method definition is associated with its invocation when the code is compiled
- dynamic binding - also called late binding
  - method definition is associated with its invocation when the method is invoked (at run time)

Static Binding

Occurs at compile time

- when the compiler turns your source code into executable byte code
- it decides right away what code it will execute for a method call

Example:

class **`Animal`** has method:

```
void dumb() {int x = 0;}
```

```
Animal a = new Animal();
a.dumb();
```

the compiler can directly insert the body **`{int x = 0;}`**

Dynamic (or late) Binding

Occurs at run-time

- Delay the binding process (deciding what code to execute) until it is time to execute it

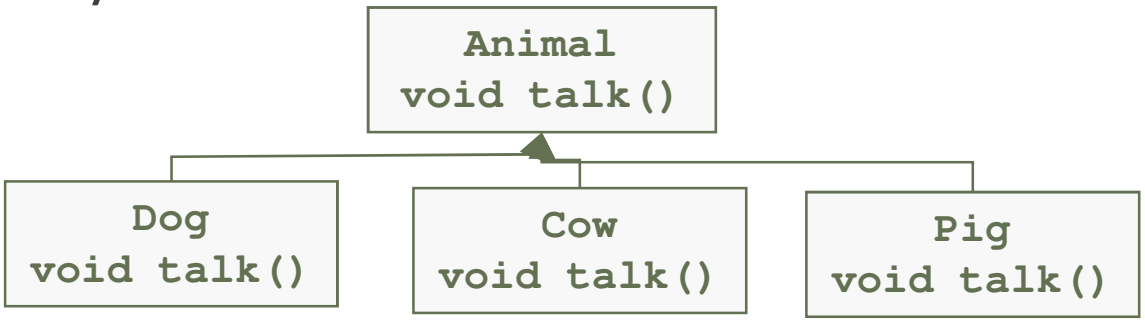
Cost: slight performance penalty

But, why would you want to do this?

Polymorphism

Literally means many shapes

Let's say we have some kind of inheritance




```
Animal myAnimal;
Pig myPig = new Pig();
Dog myDog = new Dog();
Random gen = new Random();
int num = gen.nextInt();
if (num > 0)  myAnimal = myPig;
else          myAnimal = myDog;
myAnimal.talk();
```

- what type is **myAnimal**?
- **myAnimal** is polymorphic… it can refer to different types of objects at different times
- what **myAnimal** refers to is decided at run-time
- How will the compiler know which **talk()** method to call?

Polymorphism

Suppose we create the following reference variable:

```
Animal a;
```

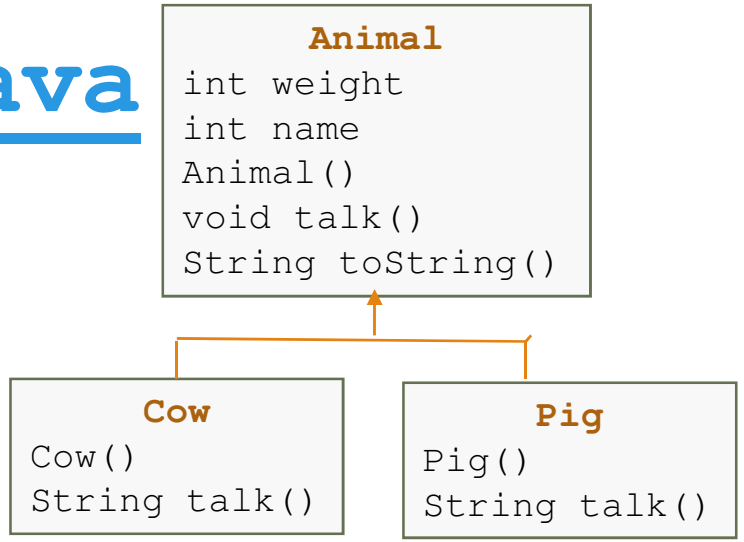
this reference can point to an **`Animal`** object, or to any object of any compatible type

This compatibility can be done:

- using inheritance or
- using interfaces

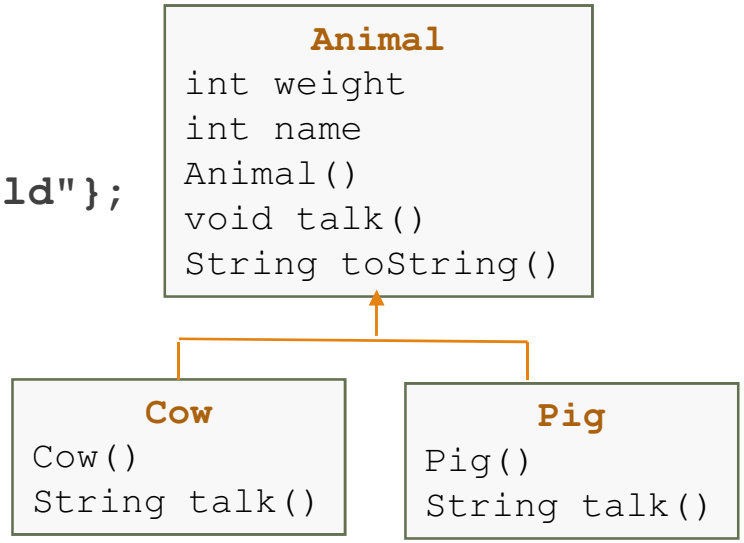
Careful use of polymorphic references can lead to elegant, robust software designs

Example: Farmpoly.java




```
class Animal {
   private int weight;
   private String name;
   public Animal(int weight, String name) {
      this.weight = weight;
      this.name = name;
   }
   public String talk() { return "generic animal talk!"; }
   public String toString(){ return("I say " + talk()+"\n"); }
}
//-------------------------------------------------------------
class Pig extends Animal {
   public Pig(int weight, String name){ super(weight, name); }
   public String talk(){ return "Oink, Oink!"; }
}
//-------------------------------------------------------------
class Cow extends Animal {
   public Cow(int weight, String name){ super(weight, name); }
   public String talk(){ return "Moo, Moo!"; }
}
```

```
public class Farmpoly {
  public static final int FARMSIZE = 3 ;
  String[] aType = {"pig", "cow", "pig"};
  String[] aName = {"Billy", "Lucy", "Arnold"};
  int[] aWeight = {43, 1789, 899};
  Animal[] aList = new Animal[FARMSIZE];

  public Farmpoly() {
   String type;
   for (int i = 0; i < FARMSIZE; i++) {
     type = aType[i];
     if (type.equals("pig")){
        aList[i] = new Pig(aWeight[i], aName[i]);
     }
     else if (type.equals("cow")){
        aList[i] = new Cow(aWeight[i],aName[i]);
     }
   }
}
```



Example: Farmpoly.java

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

    Farmpoly myFarm = new Farmpoly();
    for (int i = 0; i < FARMSIZE; i++)     {
        System.out.print(myFarm.aList[i]); // ???
    }
}
```

Output

…SO

We create an array of animal references

We place an object of animal type in each of these references

- an animal type includes any sub-class of animal

Java delays the binding of **`talk()`** until run time.

In the main loop, the right version (the most specific one) of **`talk()`** will be called

Polymorphism in a nutshell

A polymorphic reference can refer to an object whose type belongs to a specific inheritance chain

Is this polymorphism?

```
public class PrivateOverride {
   private void f() {
      System.out.println("private f()"); }

public static void main(String[] args) {
            PrivateOverride po = new Derived();
            po.f(); }}

public class Derived extends PrivateOverride {

   public void f() {                    //this method is never run.
      System.out.println("public f()"); }}
```

Dynamic Binding again

Does anyone ever do static binding?

- yes, C++ for ex. has static binding by default
- To get dynamic binding, you have to use an explicit keyword called **`virtual`**

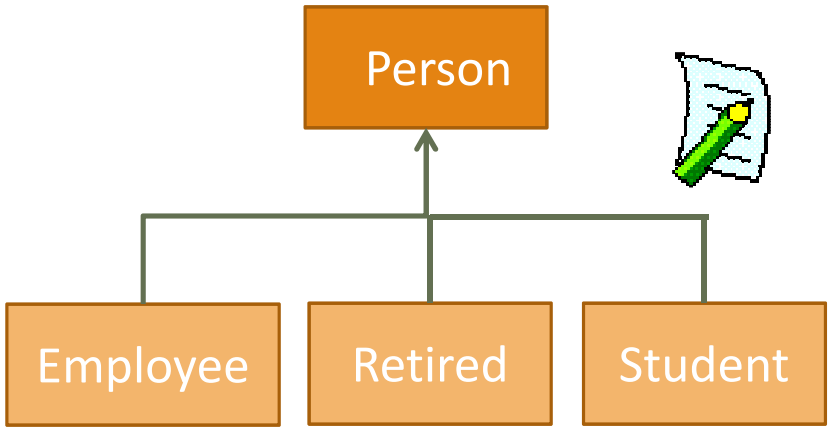
Does Java always use dynamic binding?

- Java always uses dynamic binding for methods except:
  - for **`private`**, **`final`** and **`static`** methods

Just Checking

Consider the following code where … are the required parameters for the constructors:

```
Person p = new Person(…);
int m1 = p.getMoney( );
```



Each have their own `getMoney()`

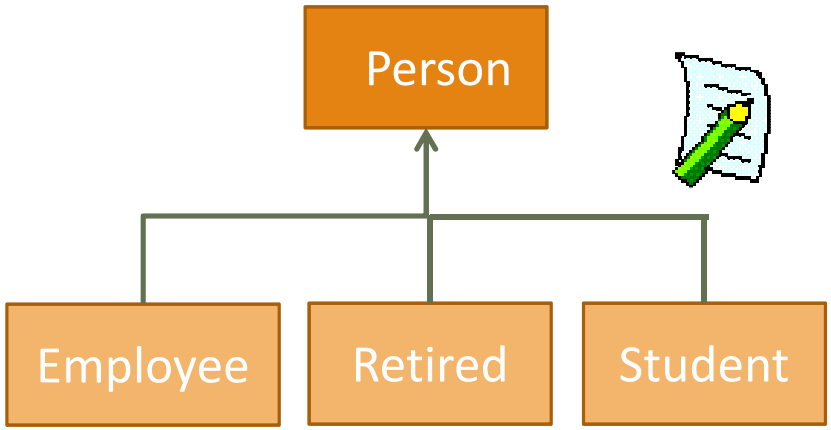
The reference to `getMoney( )` is to the class

A. Person
B. Student
C. Employee
D. Retired
E. none of the above, this cannot be determined by examining the code

Just Checking

Consider the following code where … are the required parameters for the constructors:

```
Person p = new Person(…);
int m1 = p.getMoney( );
p = new Student(…);
int m2 = p.getMoney( );
```



Each have their own `getMoney()`

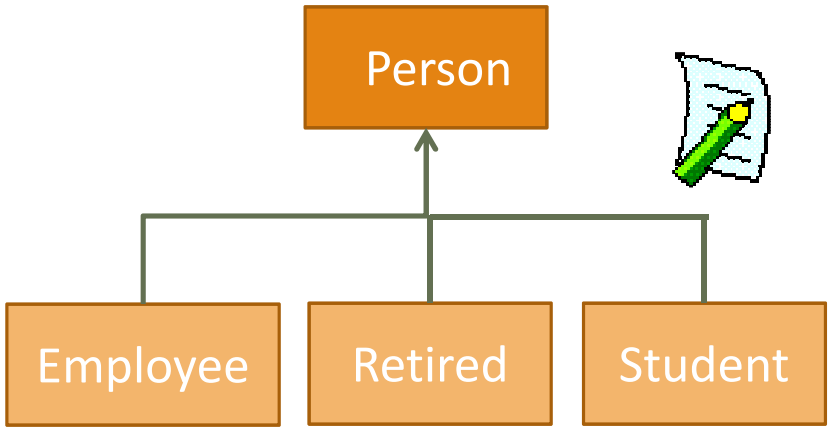
The reference to `getMoney( )` is to the class

A. Person
B. Student
C. Employee
D. Retired
E. none of the above, this cannot be determined by examining the code

Just Checking

Consider the following code where … are the required parameters for the constructors:

```
Person p = new Person(…);
int m1 = p.getMoney( );
p = new Student(…);
int m2 = p.getMoney( );
if (m2 < 100000)
      p = new   Employee(…);
else if (m1 > 50000)
    p = new Retired(…);
int m3 = p.getMoney( );
```



Each have their own getMoney()

The reference to `getMoney( )` is to the class

A. Person
B. Student
C. Employee
D. Retired
E. none of the above, this cannot be determined by examining the code

No Late Binding for some Methods

Java uses static binding with:

- private methods,
  - would serve no purpose, why?
- **final** methods
  - would serve no purpose, why?
- **static** methods
  - would serve a purpose, when?

No Late Binding for Static Methods

```
class Parent{
    public static void classMethod(){
        System.out.println("ClassMethod in Parent");
    }
    public void instanceMethod(){
        System.out.println("InstanceMethod in Parent");
    }
}
```

This is called "hiding" not "overriding".

Static method cannot be overridden

```
class Child extends Parent{
    public static void classMethod(){
        System.out.println("ClassMethod in Child");
    }
    public void instanceMethod(){
        System.out.println("InstanceMethod in Child");
    }
}
```

No Late Binding for Static Methods

```
public static void main(String[] arg) {
      Parent p1 = new Parent();
      Child c = new Child();
      c.instanceMethod();
      c.classMethod();
      p1.instanceMethod();
      p1.classMethod();

      Parent p2 = new Child();
      p2.instanceMethod();
      p2.classMethod();
}
```

Output

No Late Binding for Static Methods

```
public static void main(String[] arg) {
        Parent p1 = new Parent();
        Child c = new Child();
        c.instanceMethod();
        c.classMethod();
        p1.instanceMethod();
        p1.classMethod();

        Parent p2 = new Child();
        p2.instanceMethod();
        p2.classMethod();
}
```

Instance method: JVM uses the actual class of the instance p2 to determine which method to run.

Class method: The compiler will only look at the *declared type* of the reference, and use that declared type to determine, *at compile time*, which method to call.

Better use class name

No Late Binding for Static Methods

```
class Parent{
   public static void classMethod(){
        System.out.println("ClassMethod in Parent");
   }
   public void instanceMethod(){
        System.out.println("InstanceMethod in Parent");
   }
   public void hello(){
        System.out.println("Hello from parent call classMethod");
        classMethod();
   }
}
```

```
class Child extends Parent{
   public static void classMethod(){
        System.out.println("ClassMethod in Child");
   }
   public void instanceMethod(){
        System.out.println("InstanceMethod in Child");
   }
}
```

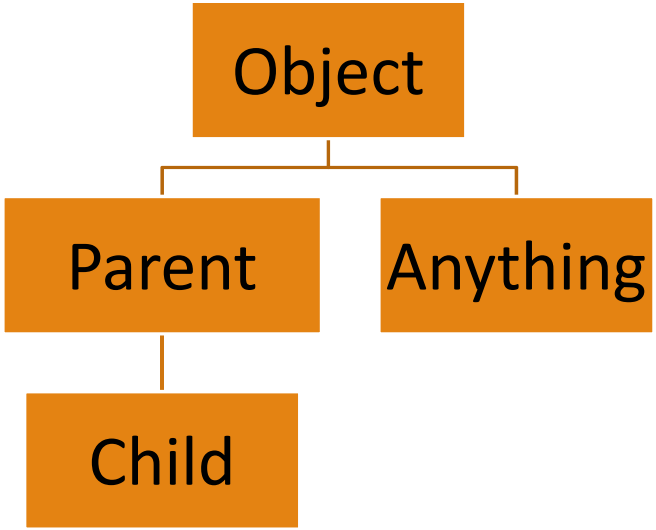
No Late Binding for Static Methods

```
public static void main(String[] arg) {
      Parent p1 = new Parent();
      Child c = new Child();
      c.hello();
      p1.hello();

      Parent p2 = new Child();
      p2.hello();
}
```

Output

Recap: Will it compile? Will it run?

Assume the following inheritance tree:



Assume the following declarations:

```
Parent p;
Child c;
Anything a1, a2;
```

Recap: case 1

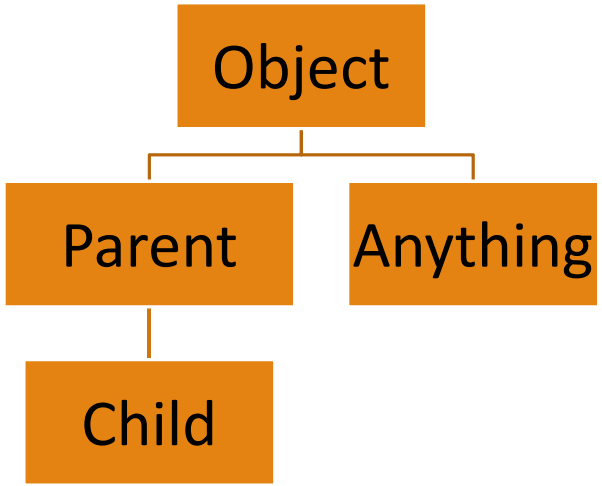
Case 1:

static type of RHS is identical to static type of LHS

```
c = c;
a1 = a2;
p = p;
```

→ *NO compiler error,*

→ *NO run-time error*



Recap: case 2

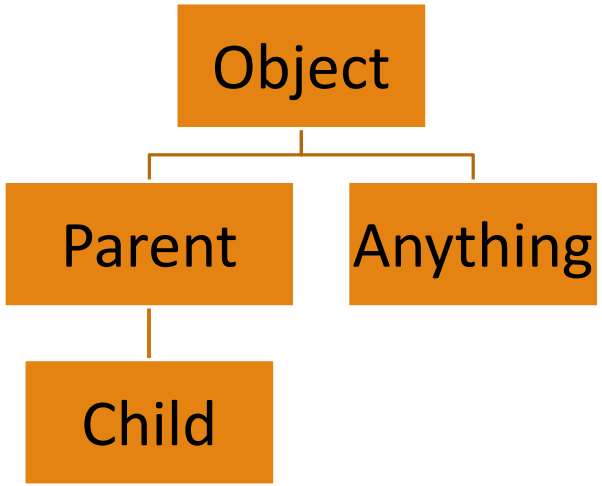
Case 2:

static type of RHS is a more specific than static type of LHS

```
p = c;
```

→ *NO compiler error,*

→ *NO run-time error*

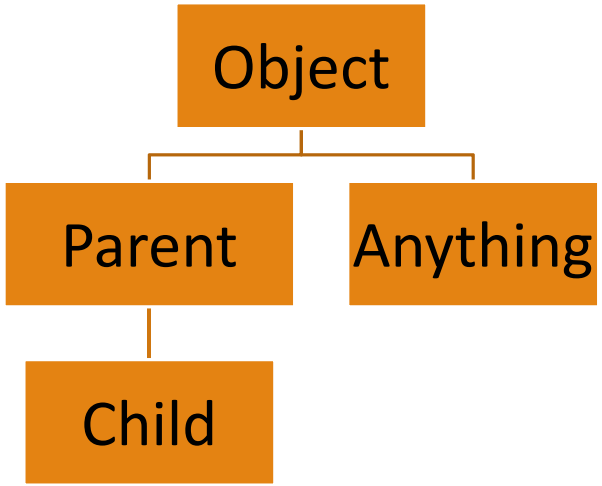


Recap: case 3

Case 3:
static type of RHS is in another branch as the static type of LHS

```
p = a1;
a1 = p;
```

→ *Compiler error*



Recap: case 4

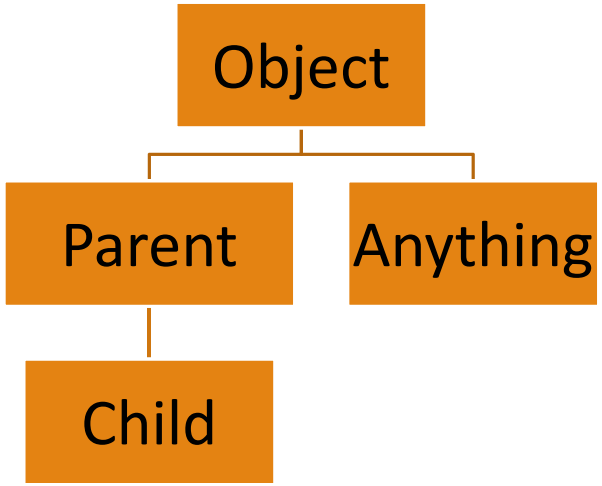
Case 4:
static type of RHS is a more general than static type of LHS

--> we need to analyze further…

- **case 4.1:**
  no casting

```
c = p;
```

→ *Compiler error*



Recap: case 4

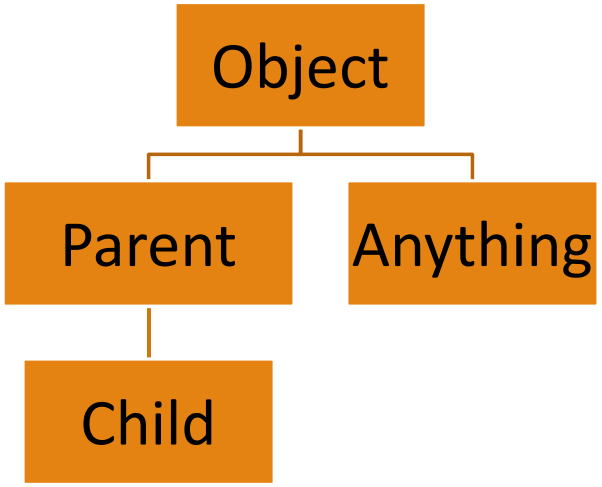
Case 4:
static type of RHS is a more general than static type of LHS

--> we need to analyze further…

- **case 4.2:**
  the RHS is casted into a type that is more general or is not in the same branch as the static type of the LHS

```
c = (Parent)p;
c = (Object)p;
c = (Anything)p;
```

→ compiler error



Recap: case 4 (con't)

Case 4:

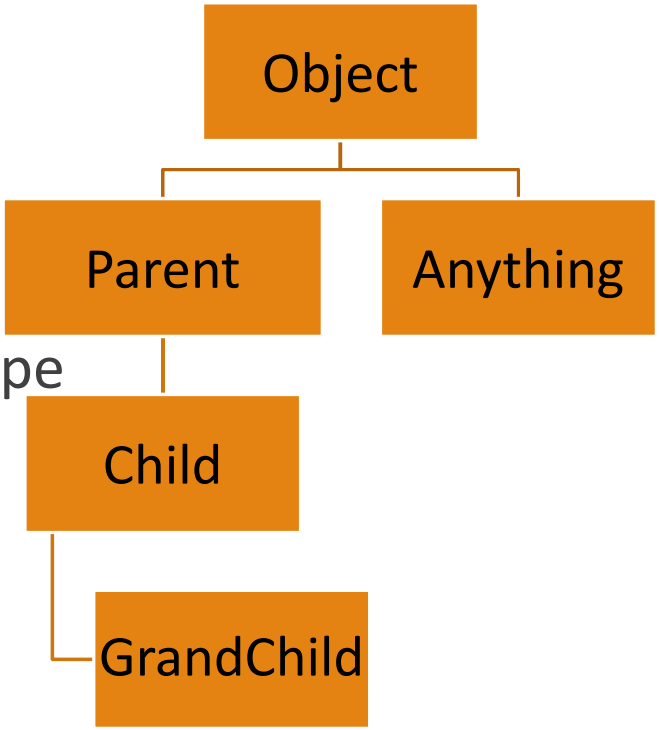
static type of RHS is a more general than static type of LHS

--> we need to analyze further…

- **case 4.3:**
  the RHS is casted into a type that is identical or is a subtype of the static type of the LHS

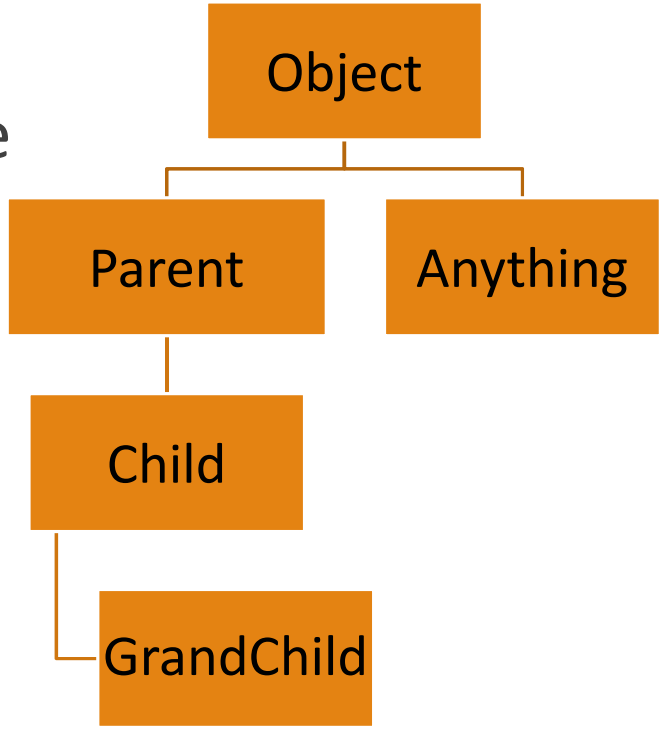
```
c = (Child) p;
c = (GrandChild) p;
```

→ NO compiler error,
we need to analyze further to determine if run-time error



Recap: case 4 (con't)

- **case 4.3.1:**
  the dynamic type of RHS is identical or more specific than the static type of the LHS




```
p = new Child();
c = (Child) p;
p = new GrandChild();
c = (GrandChild) p;
```

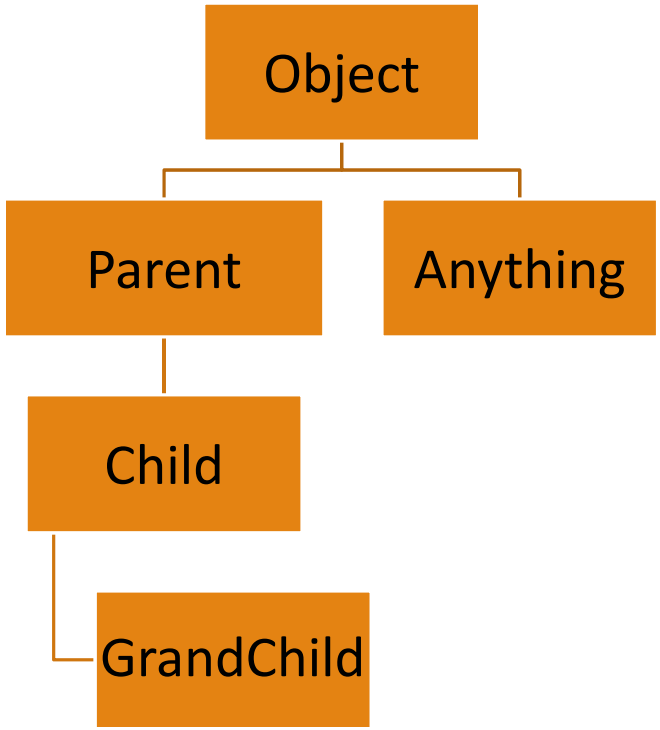
→ NO compiler error, NO run-time error.

Recap: case 4 (con't)

- **case 4.3.2:**
  the dynamic type of RHS is more general than the static type of the LHS

```
p = new Parent();
c = (Child) p;
```

→ NO compiler error,
Run-time error.



Abstract Classes

Often, you only want to/can only specify a portion of the parent

- ex. cannot write the code of a method… but you know that this method should be there

abstract class:

- only defines a generalized form that will be shared by all its subclasses, leaving the implementation details to its subclasses

concrete class:

- has concrete methods, including implementations of any abstract methods inherited from its superclasses.

Any class with an abstract method should be declared abstract.

Example: Abstract Class

```
abstract class Shape {                  // abstract class
   abstract public double area();       // abstract method
   public void display () {…}           // concrete method
}
class Square extends Shape {
  public double area() {…}
}
class Circle extends Shape {
  public double area() {…}
}
```

Example 2: Farm.java

```
abstract class Animal {
   abstract public void talk();  // all animals talk, but differently
}
//-----------------------------------------------------
class Pig extends Animal {
   public void talk() {
        System.out.print("oink oink");
  }
//-----------------------------------------------------
}
class Cow extends Animal {
   public void talk() {
        System.out.print("moo moo");
  }
}
```

Abstract Classes

An abstract class:

- cannot be instantiated
  - why not?
- often contains abstract methods, but it doesn't have to
- cannot be defined as **final** *(see next slides)*

If the class contains at least one abstract method, then it is automatically considered abstract

The use of abstract classes is a design decision; it helps us establish common elements in a class that is too general to instantiate

Abstract Methods

an abstract method

- is really a placeholder in a class hierarchy that represents a generic concept
- should be defined as high as possible in the inheritance tree
- cannot be defined as:
  - **`final`**… why not?
  - **`private`** **…** why not?
  - **`static`** (because it has no definition yet)
- cannot be a constructor

Abstract Methods

Child classes should provide actual method bodies

Any subclass of an abstract class must either implement all the abstract methods in the superclass or be itself declared abstract.

A concrete method can be overriden to become abstract.

```
abstract class A {
 short m2() { return (short) 420; }
}
abstract class B extends A {
 abstract short m2();
}
```

A First Look at the `clone` Method

Every object inherits the method `clone` from the class `Object`

- `clone` has no parameters
- It should return a deep copy of the calling object

However, the inherited version cannot do that

- each class is expected to override it with a more appropriate version

A First Look at the **`clone`** Method

If a class has a copy constructor, the **`clone`** method can use the *copy constructor* to create the copy returned by the **`clone`** method

```
public Animal clone() {
   return new Animal(this);
}
```

and another example:

```
public Dog clone() {
   return new Dog(this);
}
```

`clone` Returning an `Object`

Prior to version 5.0, Java did not allow covariant return types

So, the `clone` method for all classes returned an `Object`

```
public Object clone() {
     return new Animal(this);
}
```

So, the result had to be casted

```
Animal copy = (Animal)original.clone();
```

Limitations of Copy Constructors

The copy constructor and **`clone`** method may appear to do the same thing

But there are cases where only a **`clone`** will work

Limitations of Copy Constructors

Ex: given a method in the class **`Animal`** that copies an array of animals

If the array contains objects of type **`Dog`**

- If we use the copy constructor:
  - then the copy will be a plain Animal, not a true copy

```
b[i] = new Animal(a[i]);
//plain Animal object
```

Limitations of Copy Constructors …

- If we use the **`clone`** method, a true copy is made

```
b[i] = (a[i].clone()); //Dog object
```

  - because the method **`clone`** has the same name in all classes, and polymorphism works with method names

  - The copy constructors named **`Animal`** and **`Dog`** have different names, and polymorphism doesn't work with methods of different names