Lecture 2: Classes and objects. Inheritances

Outline

- Object oriented programming
- Defining Classes
- Using Classes
- References vs Values
- Inheritances

Object oriented programming

- Represent the real world

Baby

Object oriented programming

- Represent the real world

Baby

Name

Sex

Weight

Decibels

poops so far

Object Oriented Programming

- Objects group together
  - Primitives (int, double, char, etc..)
  - Objects (String, etc…)

Baby

```
String name
boolean isMale
double weight
double decibels
int numPoops
```

Why use classes?

- Why not just primitives?

```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
```

Why use classes?

- Why not just primitives?

```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
// little baby david
String nameDavid2;
double weightDavid2;
```

David2?
Terrible ☹

Why use classes?

- Why not just primitives?

```
// little baby alex
String nameAlex;
double weightAlex;
// little baby david
String nameDavid;
double weightDavid;
// little baby david
String nameDavid2;
double weightDavid2;
```

David2?
Terrible ☹

500 Babies? That Sucks!

Why use classes?

Baby1

Why use classes?

Baby1

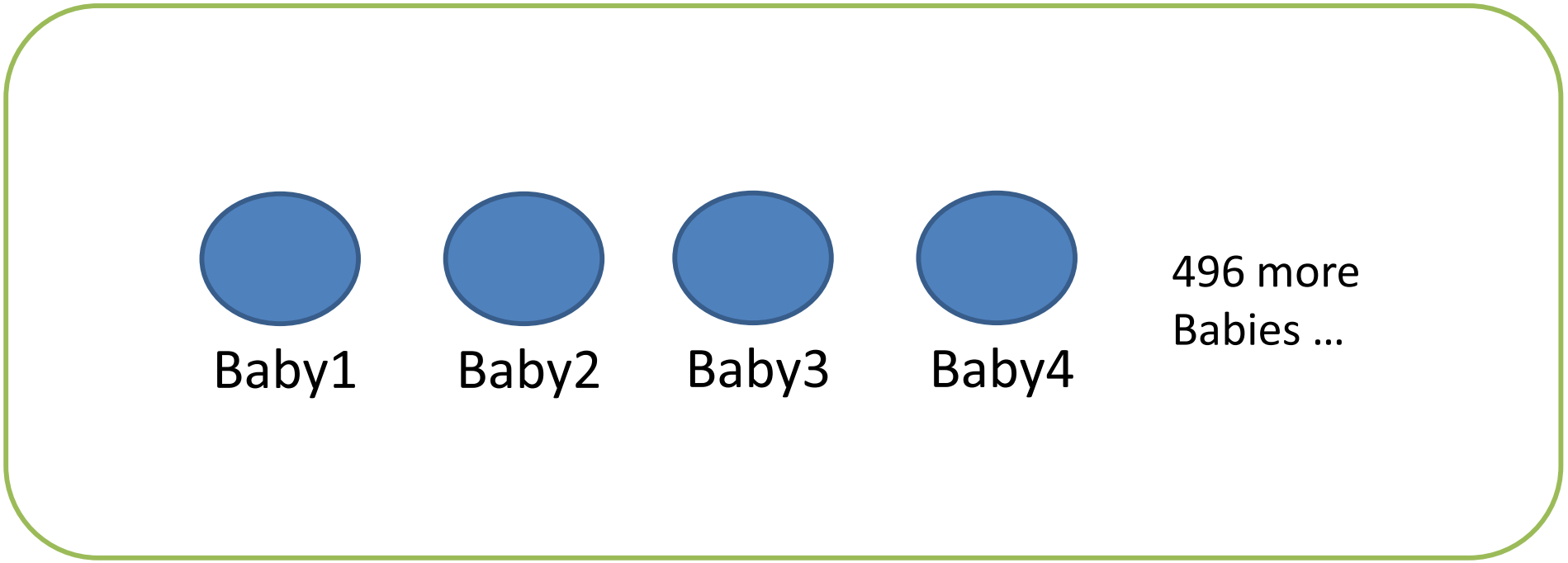
Baby2

Baby3

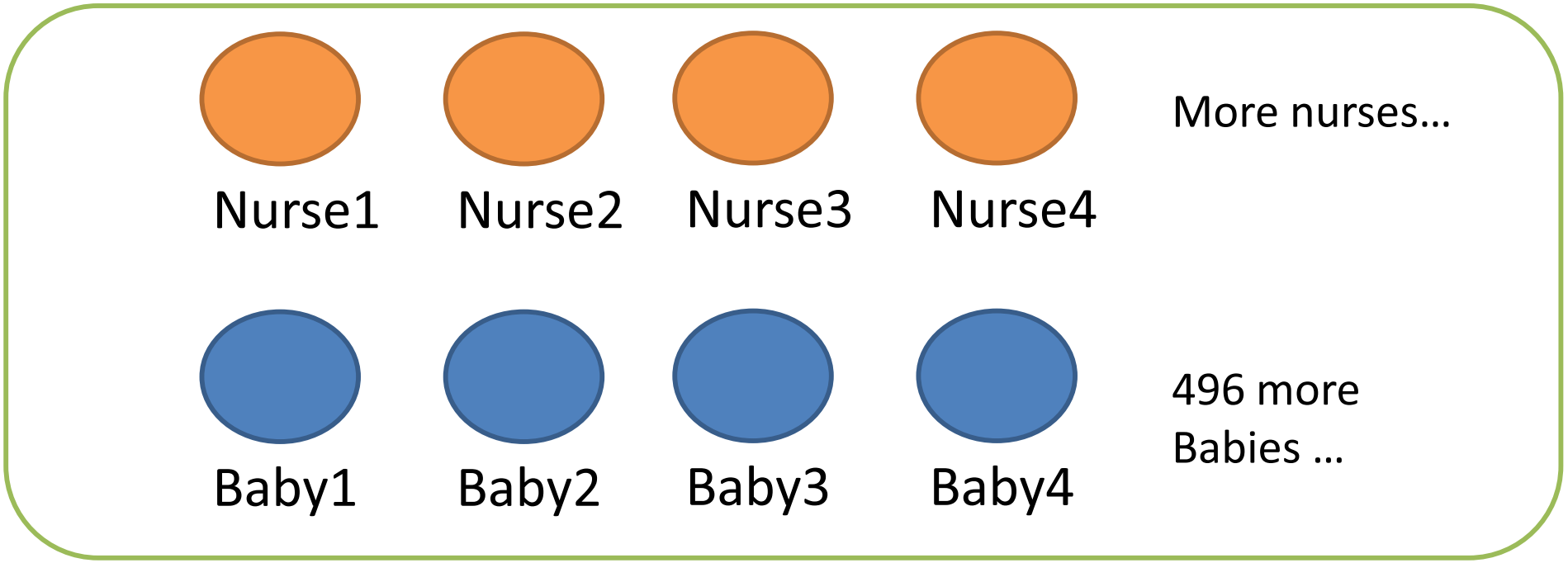
Baby4

496 more Babies …

Why use classes?

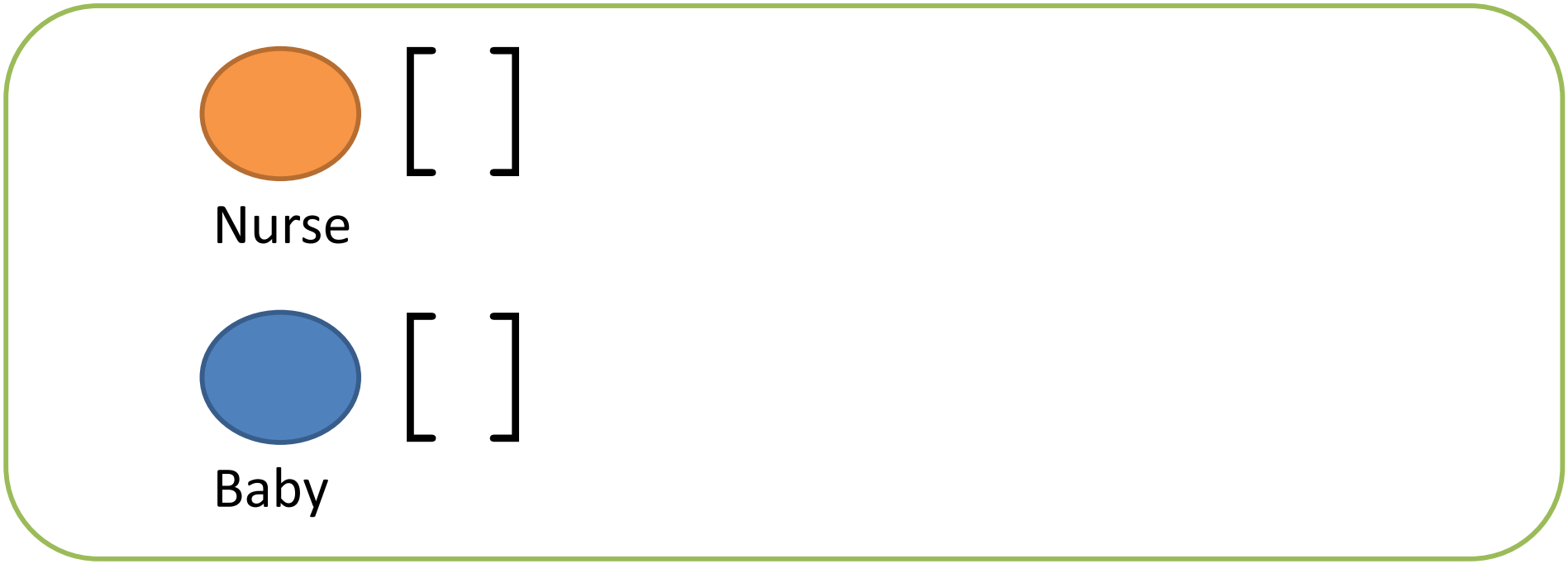


Why use classes?

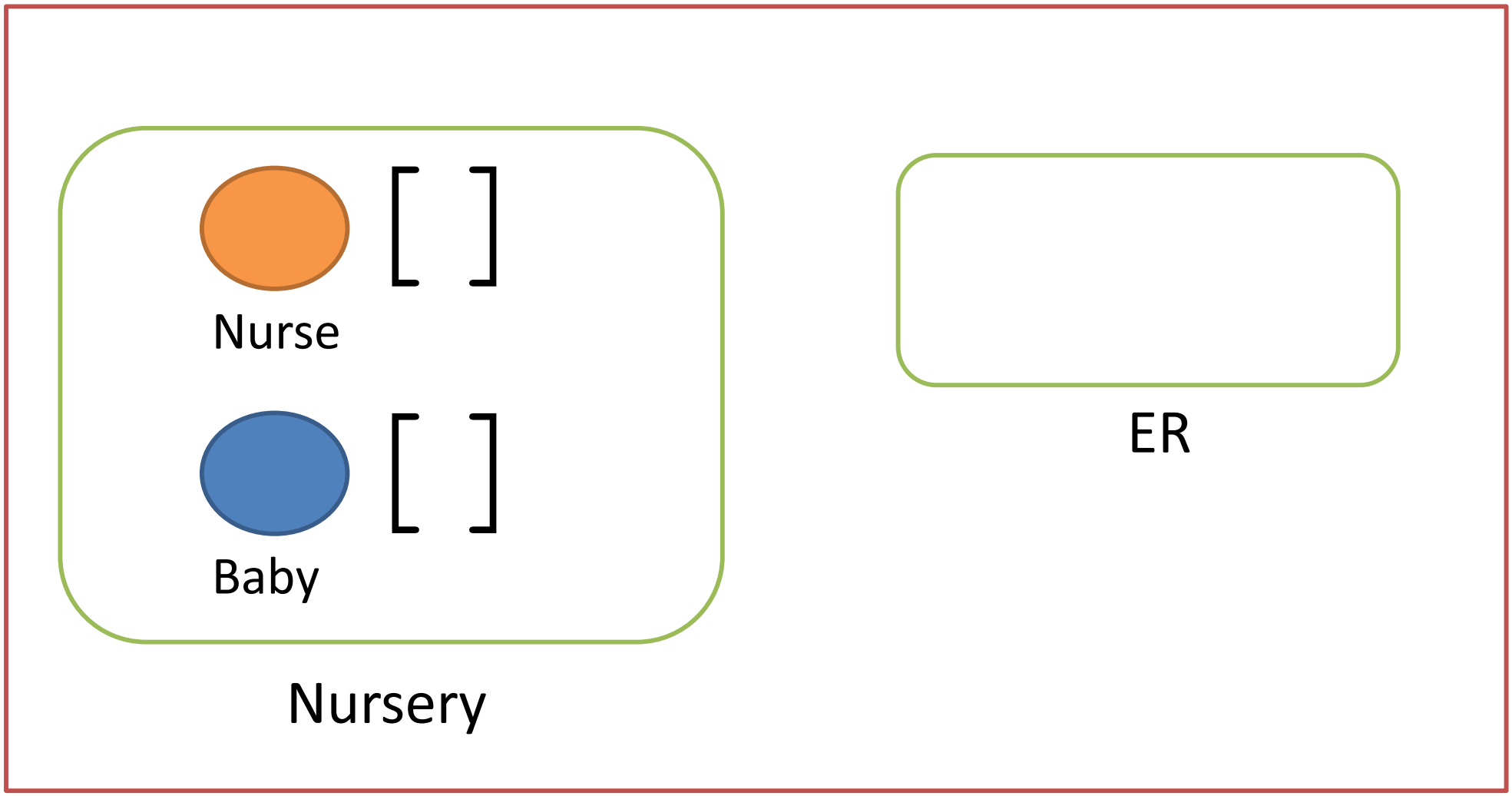


Nursery

Why use classes?



Why use classes?



Outline

- Object oriented programming
- Defining Classes
- Using Classes
- References vs Values
- Inheritances

Class - overview

```
public class Baby {
    String name;
    boolean isMale;
    double weight;
    double decibels;
    int numPoops = 0;

    void poop() {
      numPoops += 1;
      System.out.println("Dear mother, "+
      "I have pooped.  Ready the diaper.");
    }
}
```

Class Definition

Class - overview

```
Baby myBaby = new Baby();
```

Class

Instance

Let’s declare a baby!

```
public class Baby {

}
```

Let’s declare a baby!

```
public class Baby {

}
```

fields

methods

Note

- Class names are Capitalized
- 1 Class = 1 file
- Having a `main` method means the class can be run

Baby fields

```
public class Baby {

    TYPE var_name;
    TYPE var_name = some_value;



}
```

Baby fields

```
public class Baby {
    String name;
    double weight = 5.0;
    boolean isMale;
    int numPoops = 0;


}
```

Baby Siblings?

```
public class Baby {
   String name;
   double weight = 5.0;
   boolean isMale;
   int numPoops = 0;
   XXXXX  YYYYY;


}
```

Baby Siblings?

```
public class Baby {
    String name;
    double weight = 5.0;
    boolean isMale;
    int numPoops = 0;
    Baby[] siblings;

}
```

Ok, let's make this baby!

```
Baby ourBaby = new Baby();
```

But what about it's name? it's sex?

Constructors

```
public class CLASSNAME{
    CLASSNAME ( ) {
    }

    CLASSNAME ([ARGUMENTS]) {
    }
}
```

```
CLASSNAME obj1 = new CLASSNAME();
CLASSNAME obj2 = new CLASSNAME([ARGUMENTS])
```

Constructors

- Constructor name == the class name
- No return type – never returns anything
- Usually initialize fields
- All classes need at least one constructor
  - If you don't write one, defaults to

```
CLASSNAME () {
}
```

Baby constructor

```
public class Baby {
    String name;
    boolean isMale;
    Baby(String myname, boolean maleBaby){
          name = myname;
          isMale = maleBaby;
    }
}
```

Baby methods

```
public class Baby {
    String name = "Slim Shady";
    ...
    void sayHi() {
        System.out.println(
        "Hi, my name is.. " + name);
    }
}
```

Baby methods

```
public class Baby {
    String weight = 5.0;

    void eat(double foodWeight) {
        if (foodWeight >= 0 &&
            foodWeight < weight) {
             weight = weight + foodWeight;
        }
    }

}
```

Baby class

```
public class Baby {
    String name;
    double weight = 5.0;
    boolean isMale;
    int numPoops = 0;
    Baby[] siblings;

    void sayHi() {…}
    void eat(double foodWeight) {…}
}
```

Outline

- Object oriented programming
- Defining Classes
- Using Classes
- References vs Values
- Inheritances

Classes and Instances

```
// class Definition
public class Baby {…}


// class Instances
Baby shiloh = new Baby("Shiloh Jolie-Pitt", true);
Baby knox   = new Baby("Knox Jolie-Pitt",   true);
```

Accessing fields

- Object.FIELDNAME

```
Baby shiloh = new Baby("Shiloh Jolie-Pitt",
                       true)
System.out.println(shiloh.name);
System.out.println(shiloh.numPoops);
```

Calling Methods

- Object.METHODNAME([ARGUMENTS])

```
Baby shiloh = new Baby("Shiloh Jolie-Pitt",
                       true)
shiloh.sayHi();     // "Hi, my name is ..."
shiloh.eat(1);
```

Outline

- Object oriented programming
- Defining Classes
- Using Classes
- References vs Values
- Inheritances

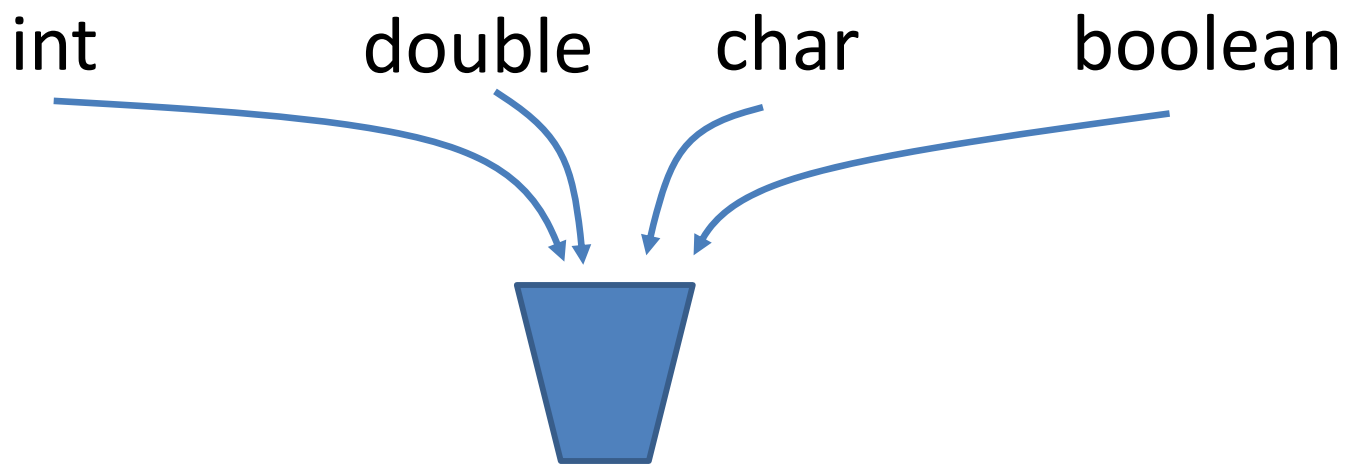
Primitives vs References

- Primitive types are basic java types
  - int, long, double, boolean, char, short, byte, float
  - The actual values are stored in the variable

- Reference types are arrays and objects
  - String, int[], Baby, …

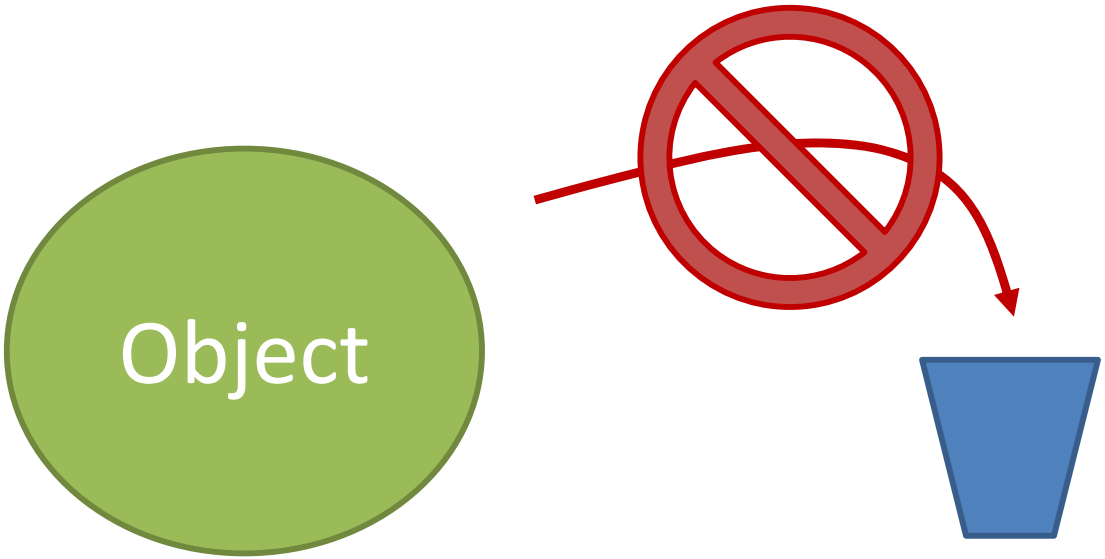
How java stores primitives

- Variables are like fixed size cups
- Primitives are small enough that they just fit into the cup



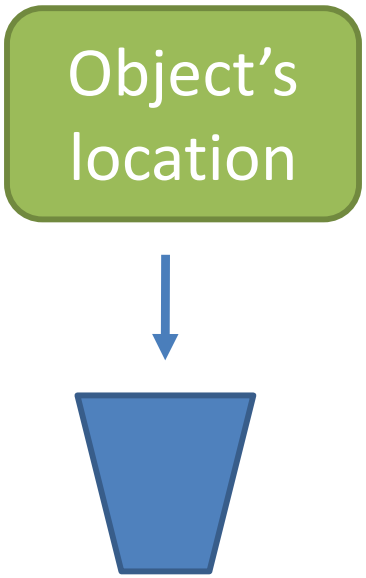
How java stores objects

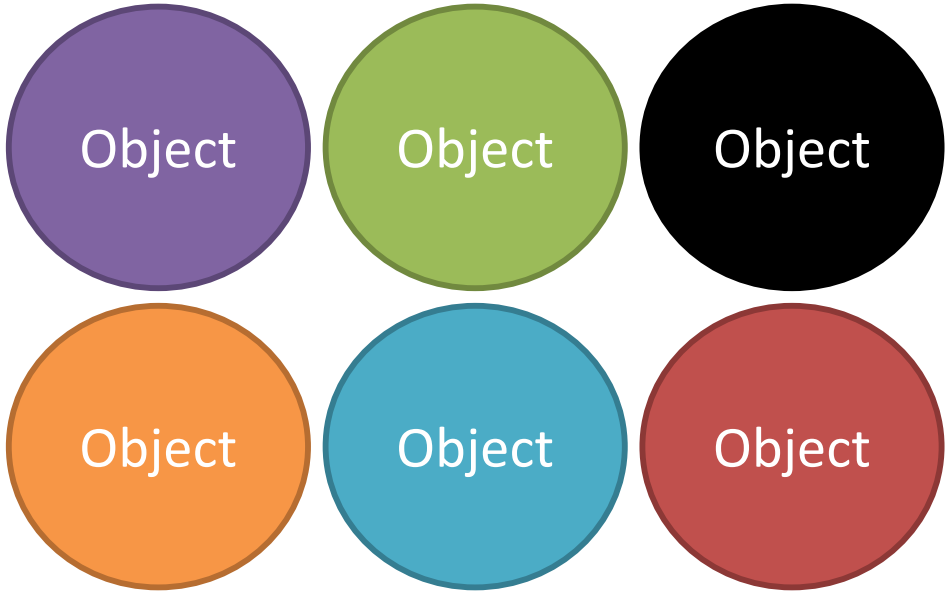
- Objects are too big to fit in a variable
  - Stored somewhere else
  - Variable stores a number that locates the object



How java stores objects

- Objects are too big to fit in a variable
  - Stored somewhere else
  - Variable stores a number that locates the object





References

- The object's location is called a reference
- == compares the references

```
Baby shiloh1 = new Baby("shiloh");
Baby shiloh2 = new Baby("shiloh");
```

Does `shiloh1 == shiloh2`?

References

- The object's location is called a reference
- == compares the references

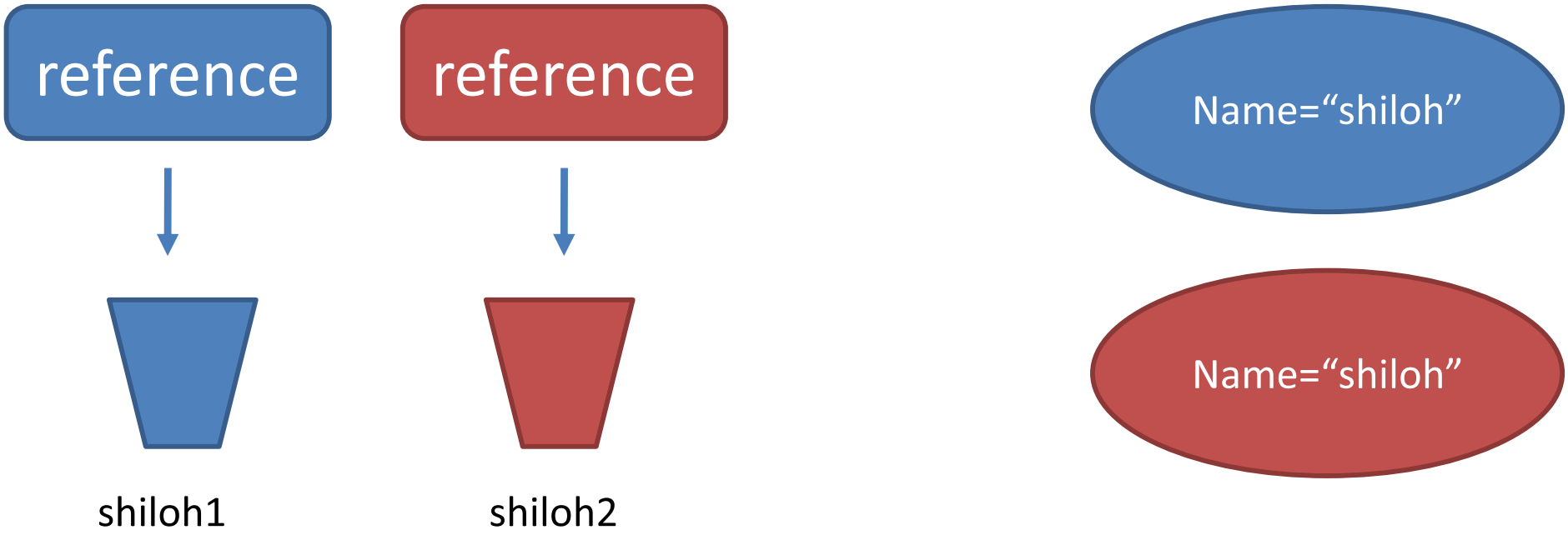
```
Baby shiloh1 = new Baby("shiloh");
Baby shiloh2 = new Baby("shiloh");
```

Does `shiloh1 == shiloh2`?

no

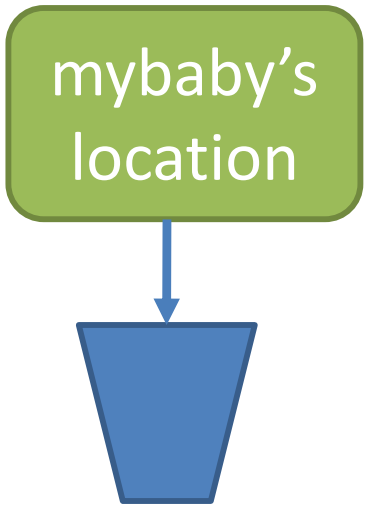
References

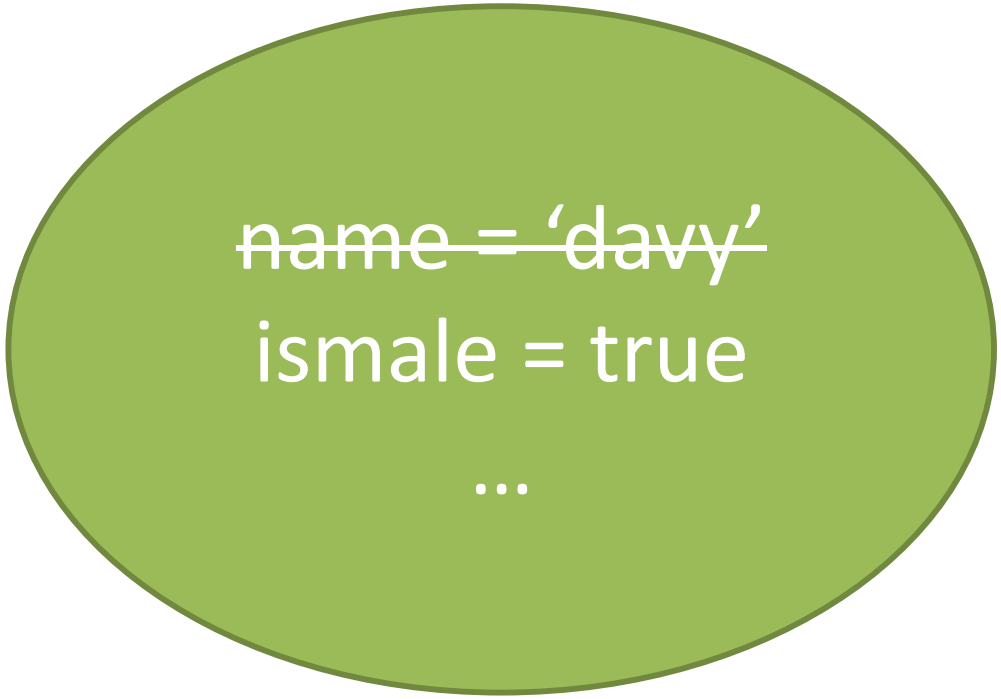
```
Baby shiloh1 = new Baby("shiloh");
Baby shiloh2 = new Baby("shiloh");
```



References

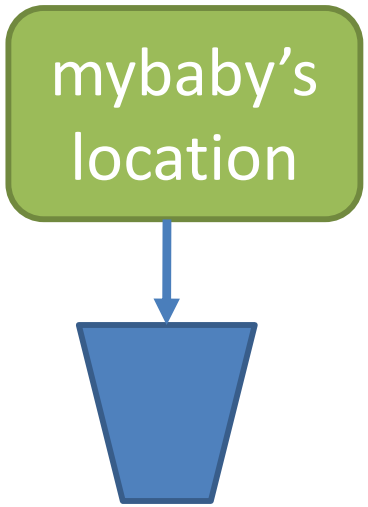
```
Baby mybaby = new Baby("davy", true)
mybaby.name = "david"
```

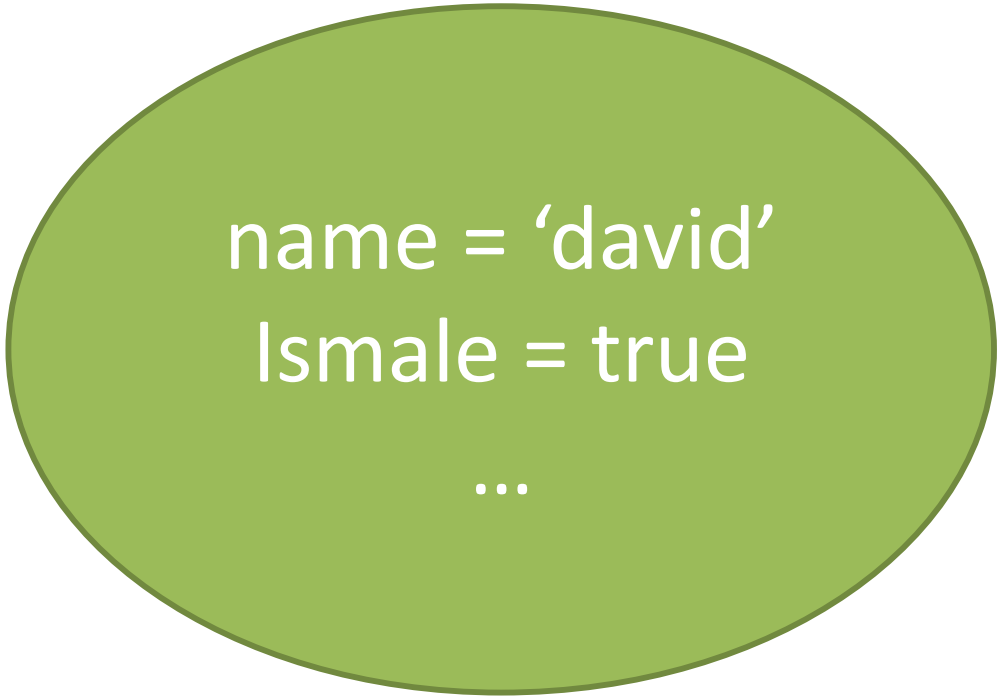




References

```
Baby mybaby = new Baby('davy', true)
mybaby.name = 'david'
```

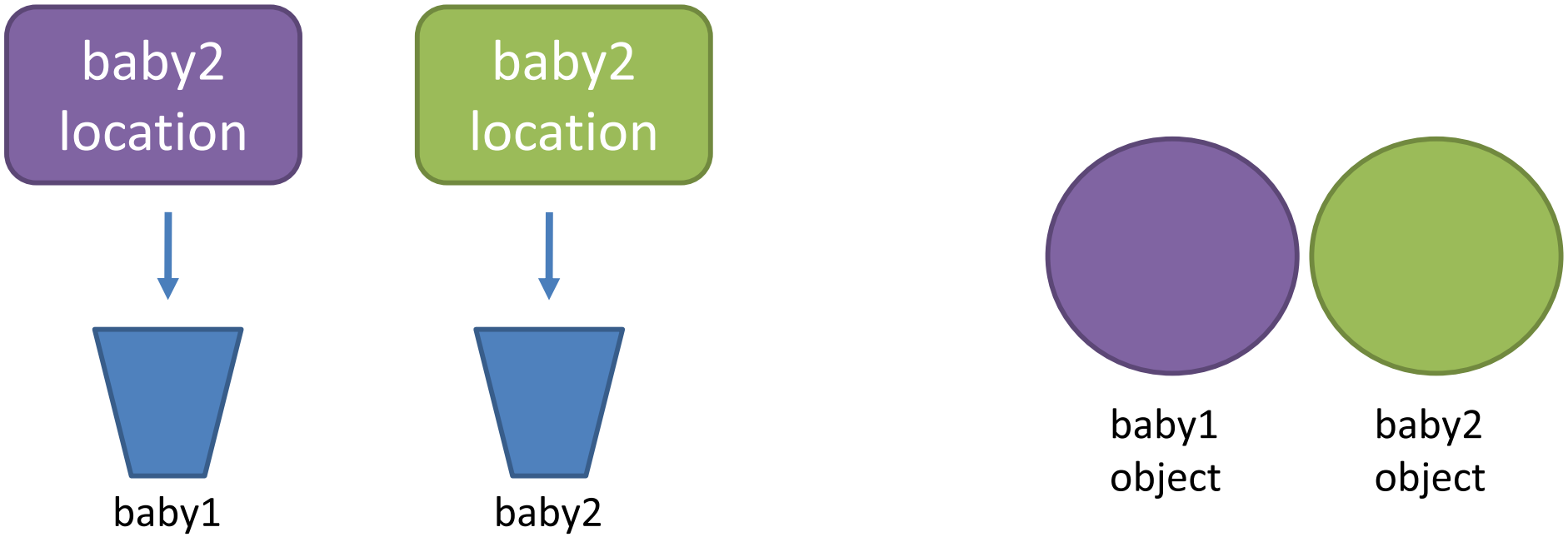




References

- Using = updates the reference.

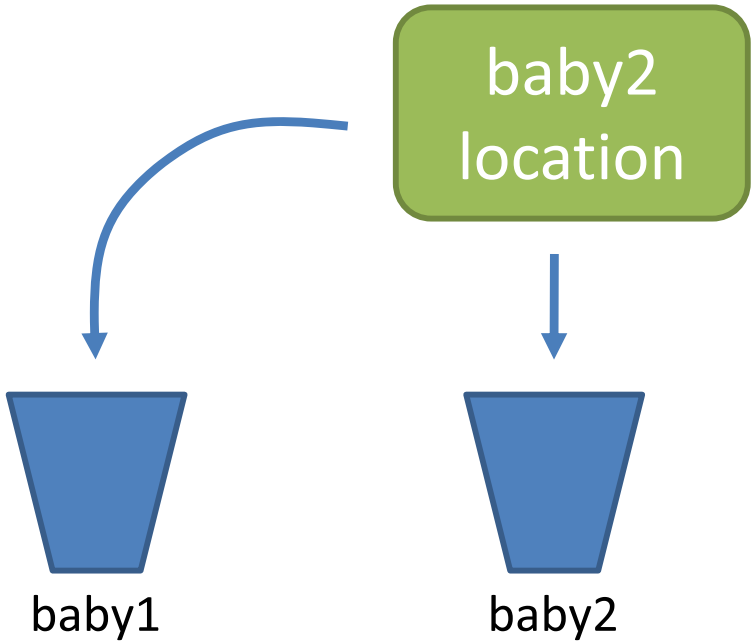
```
baby1 = baby2
```

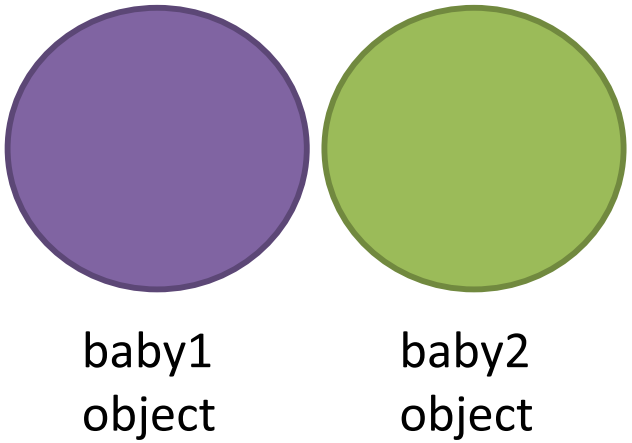


References

- Using = updates the reference.

```
baby1 = baby2
```





References

- using [ ] or •
  - Follows the reference to the object
  - May modify the object, but never the reference
- Imagine
  - Following directions to a house
  - Moving the furniture around
- Analogous to
  - Following the reference to an object
  - Changing fields in the object

Methods and references

```
void doSomething(int x, int[] ys, Baby b) {
    x = 99;
    ys[0] = 99;
    b.name = "99";
}
...

int i = 0;
int[] j = {0};
Baby k = new Baby("50", true);
doSomething(i, j, k);
```

i=? j=? k=?

Outline

- Object oriented programming
- Defining Classes
- Using Classes
- References vs Values
- Inheritances

Very *Very* Basic Inheritance

- Making a Game

```
public class Dude {
    public String name;
    public int hp = 100
    public int mp = 0;

    public void sayName() {
          System.out.println(name);
    }
    public void punchFace(Dude target) {
          target.hp -= 10;
    }
}
```

Inheritance..

- Now create a Wizard…

```
public class Wizard {
    // ugh, gotta copy and paste
    // Dude's stuff
}
```

Inheritance?

- Now create a Wizard…

But Wait!

A Wizard does and has everything a Dude does and has!

Inheritance?

- Now create a Wizard…

Don’t Act Now!

You don’t have to Copy & Paste!

Buy Inheritance!

- Wizard is a **subclass** of Dude

```
public class Wizard extends Dude {
}
```

Buy Inheritance!

- Wizard can use everything* the Dude has!

```
wizard1.hp += 1;
```

- Wizard can do everything* Dude can do!

```
wizard1.punchFace(dude1);
```

- You can use a Wizard like a Dude too!

```
dude1.punchface(wizard1);
```

*except for private fields and methods

Buy Inheritance!

- Now augment a Wizard

```
public class Wizard extends Dude {
    ArrayList<Spell> spells;
    public class cast(String spell) {
        // cool stuff here
        ...
        mp -= 10;
    }
}
```

Inheriting from inherited classes

- What about a Grand Wizard?

```
public class GrandWizard extends Wizard {
    public void sayName() {
      System.out.println("Grand wizard" + name)
    }
}

grandWizard1.name = "Flash"
grandWizard1.sayName();
((Dude)grandWizard1).sayName();
```

How does Java do that?

- What Java does when it sees

```
grandWizard1.punchFace(dude1)
```

1. Look for `punchFace()` in the GrandWizard class
2. It's not there! Does GrandWizard have a parent?
3. Look for `punchFace()` in Wizard class
4. It's not there! Does Wizard have a parent?
5. Look for `punchFace()` in Dude class
6. Found it! Call `punchFace()`
7. Deduct hp from dude1

How does Java do that? pt2

- What Java does when it sees

```
((Dude)grandWizard1).sayName()
```

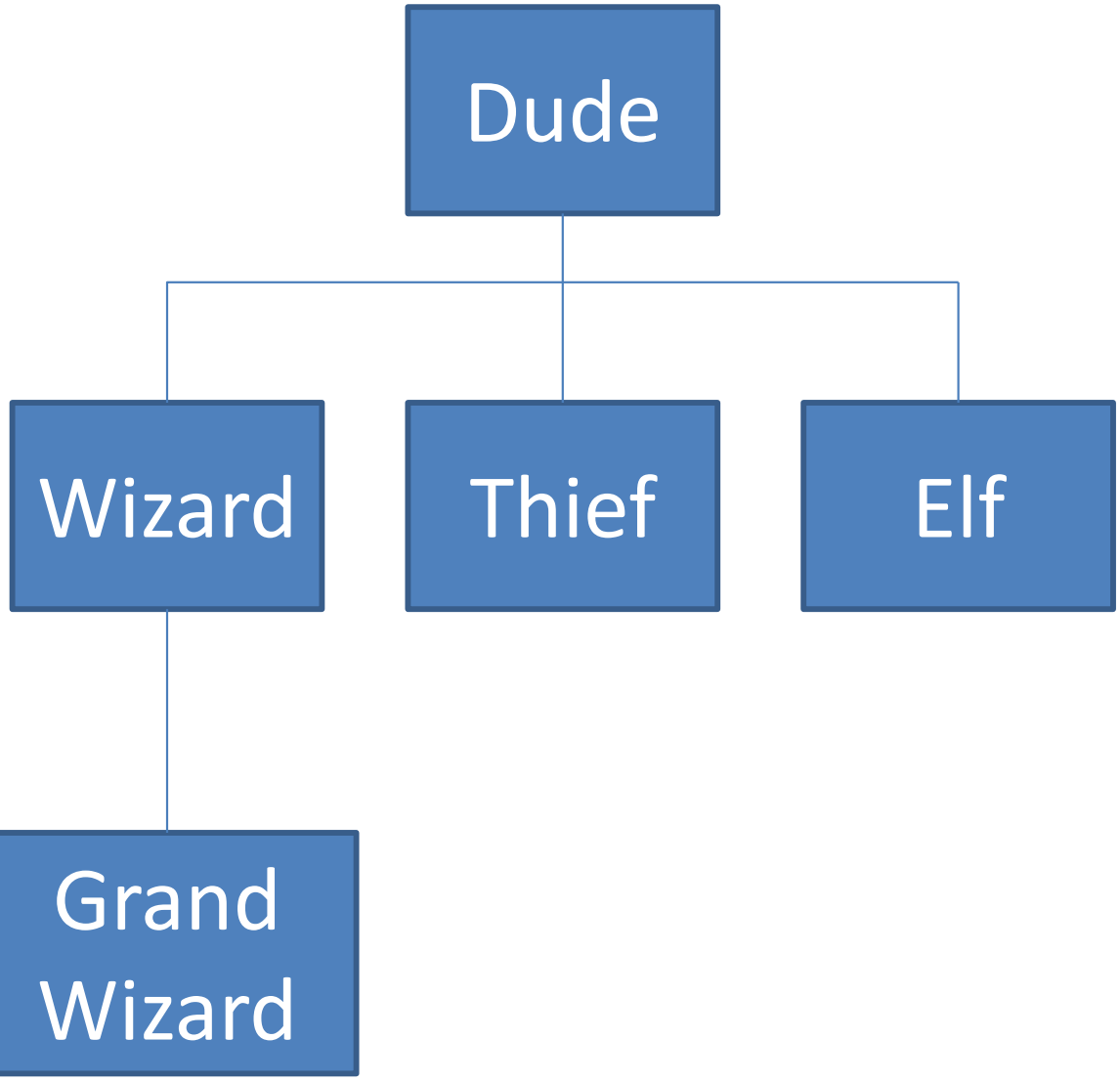
1. Cast to `Dude` tells Java to start looking in `Dude`
2. Look for `sayName()` in Dude class
3. Found it! Call `sayName()`

What's going on?

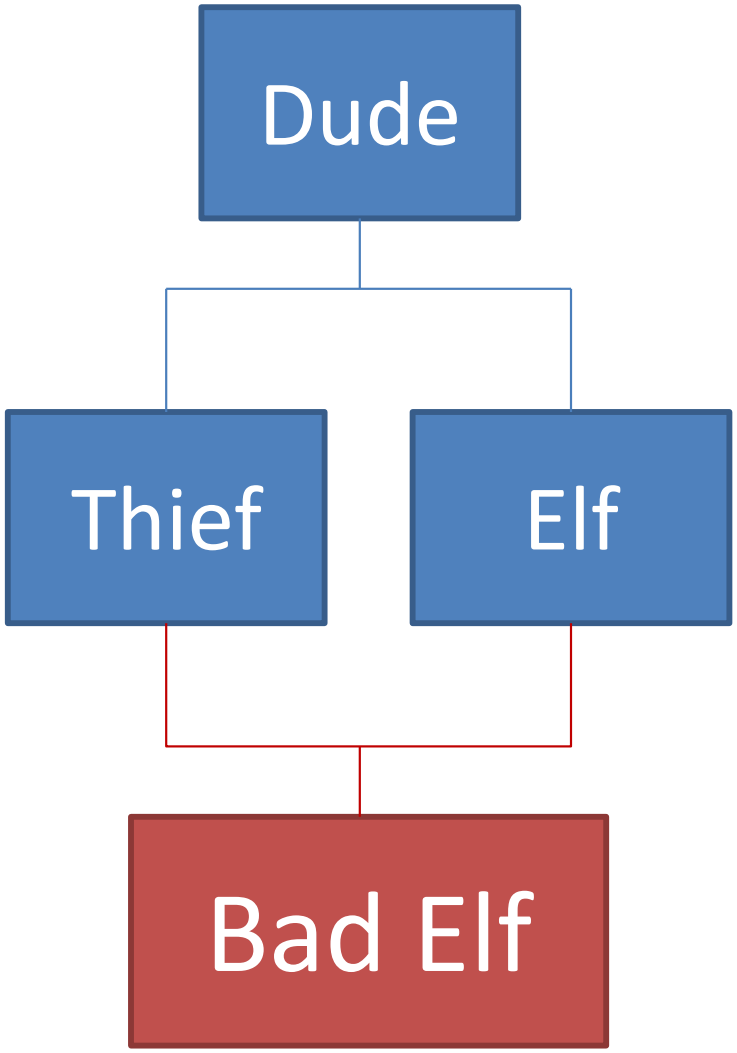
Parent of Wizard, Elf..

Subclass of Dude

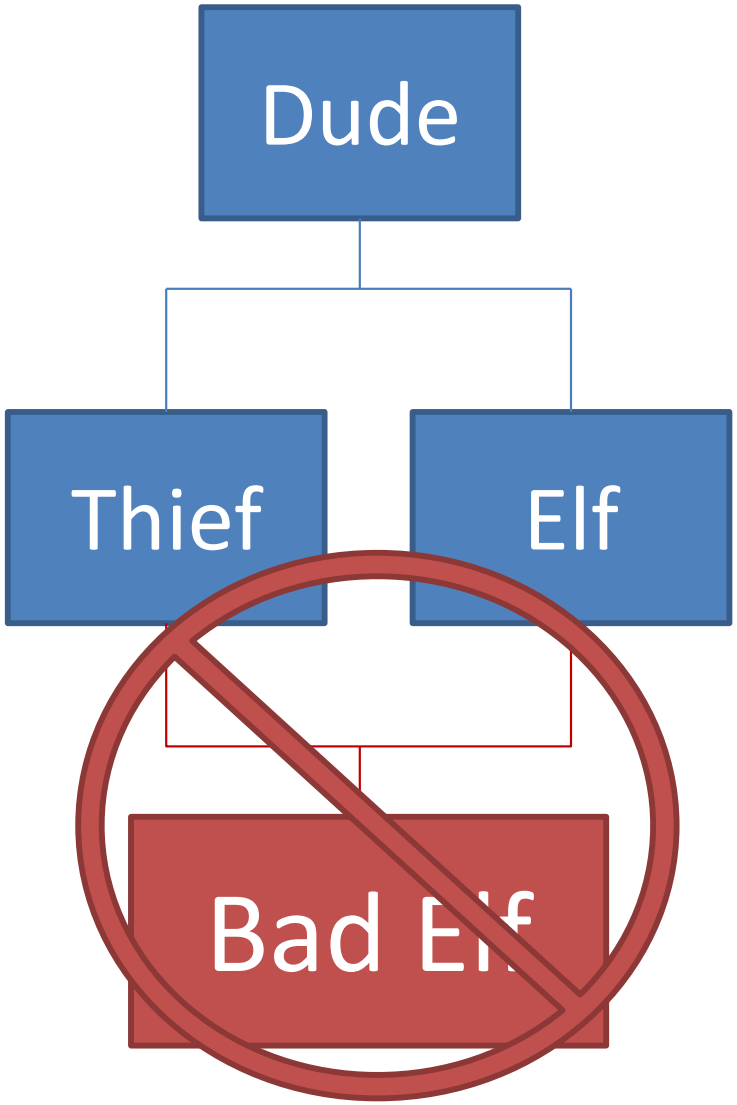
Subclass of Wizard



You can only inherit from one class



You can only inherit from one class



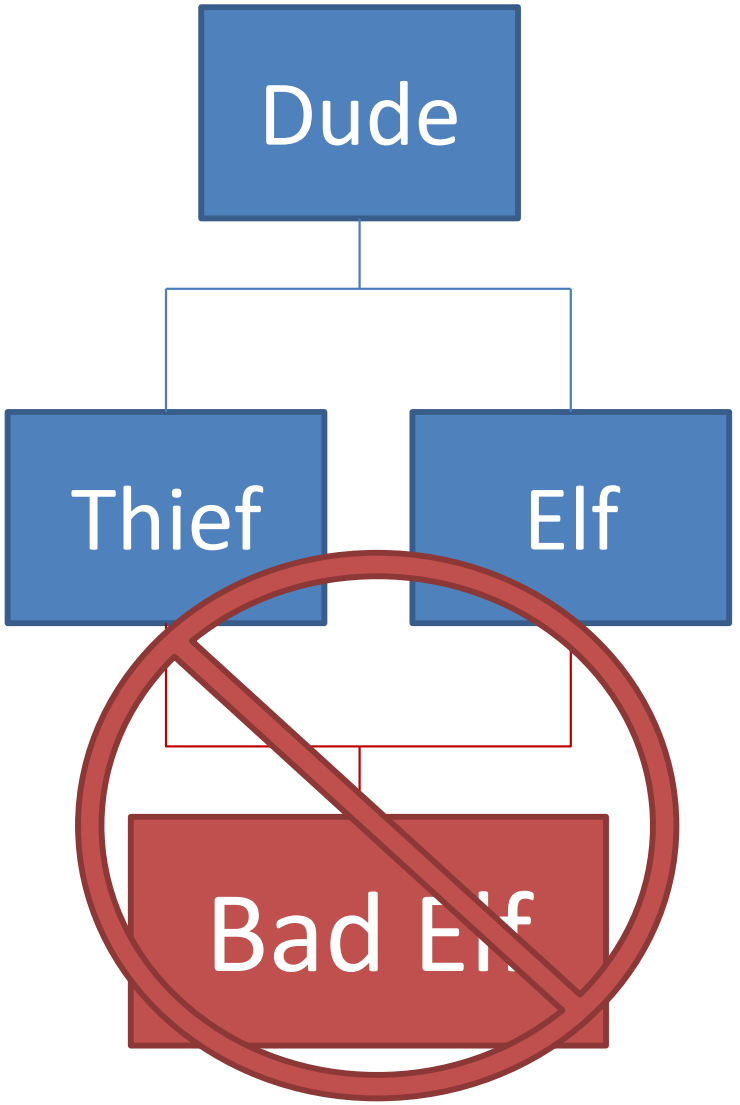
You can only inherit from one class

What if Thief and Elf both implement

```
public void sneakUp()
```

If they implemented differently, which `sneakUp()` does BadElf call?

Java Doesn't Know!!



Inheritance Summary

- class A extends B {} == A is a subclass of B
- A has all the fields and methods that B has
- A can add it's own fields and methods
- A can only have 1 parent
- A can replace a parent's method by re-implementing it
- If A doesn't implement something Java searches ancestors