Today’s Topics

Object oriented programming

Defining Classes

Using Classes

References vs Values

Static types and methods

Why use classes?

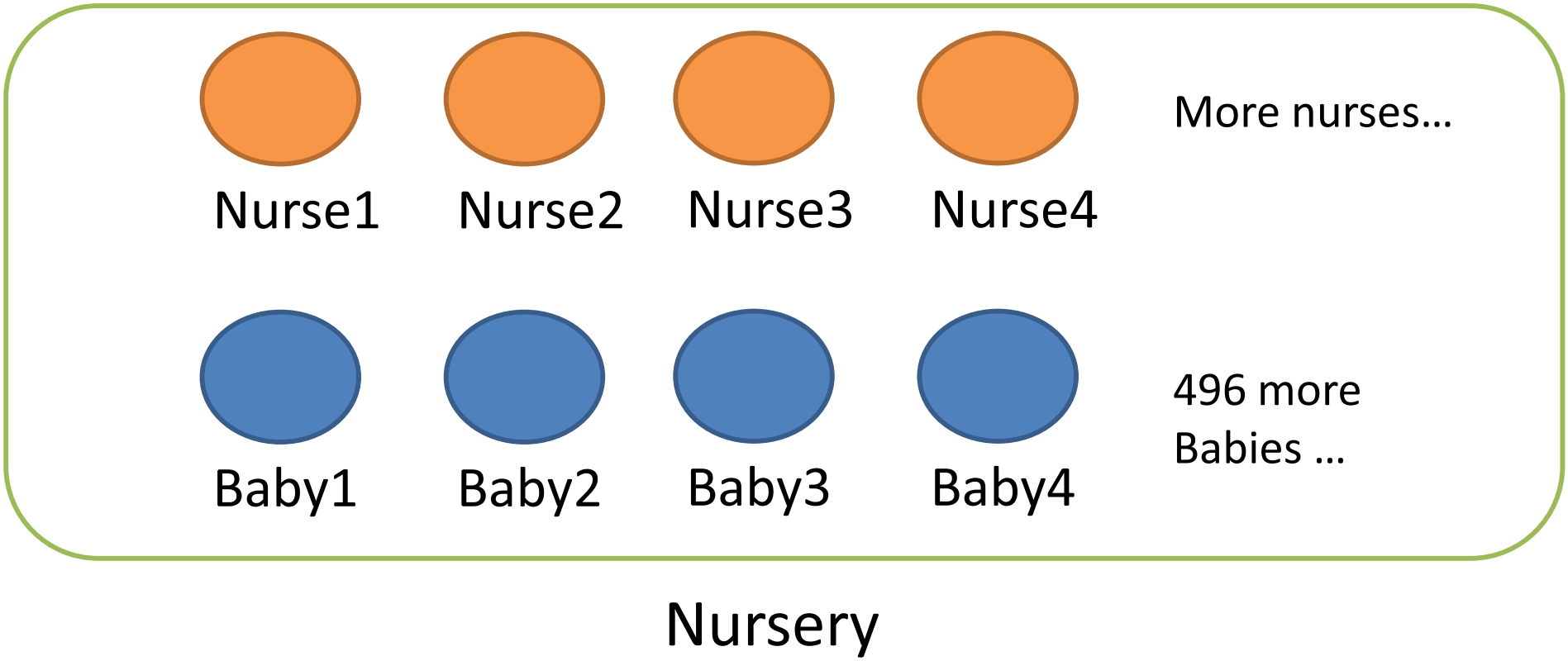
- Why not just primitives?

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

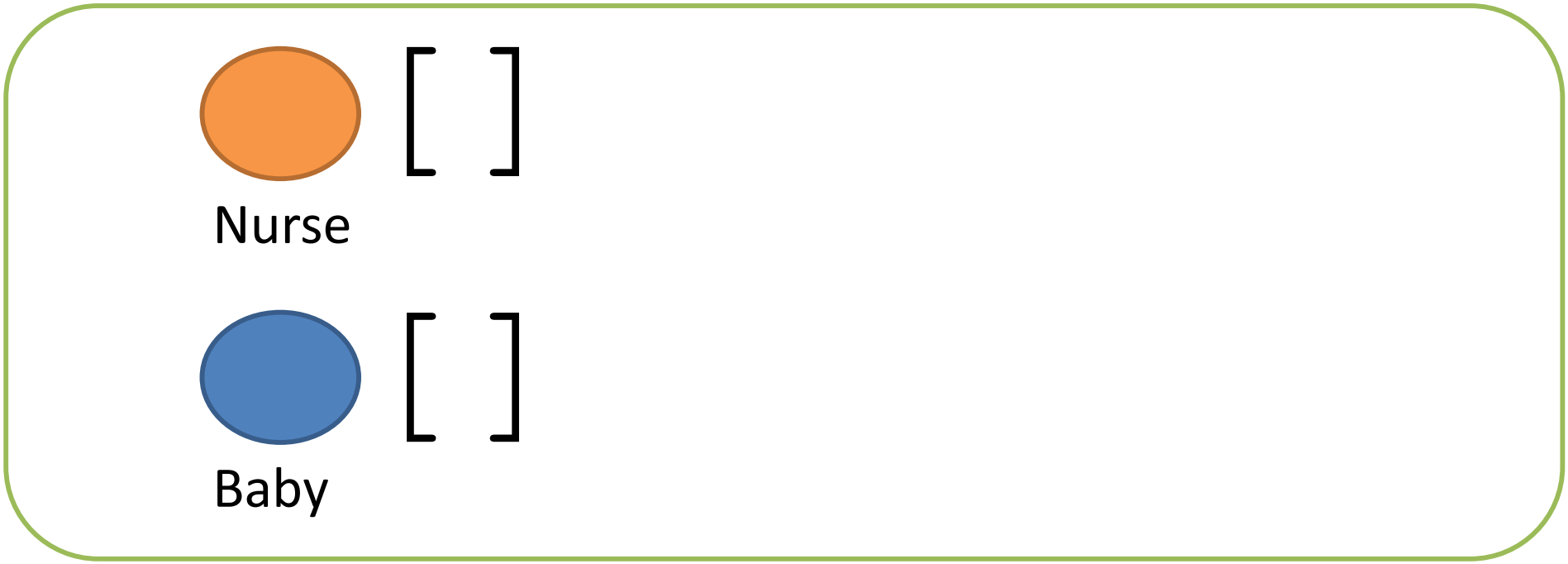
David2?
Terrible ☹

500 Babies? That Sucks!

Why use classes?



Why use classes?



Nursery

Note

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

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;
    }
}
```

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);
```

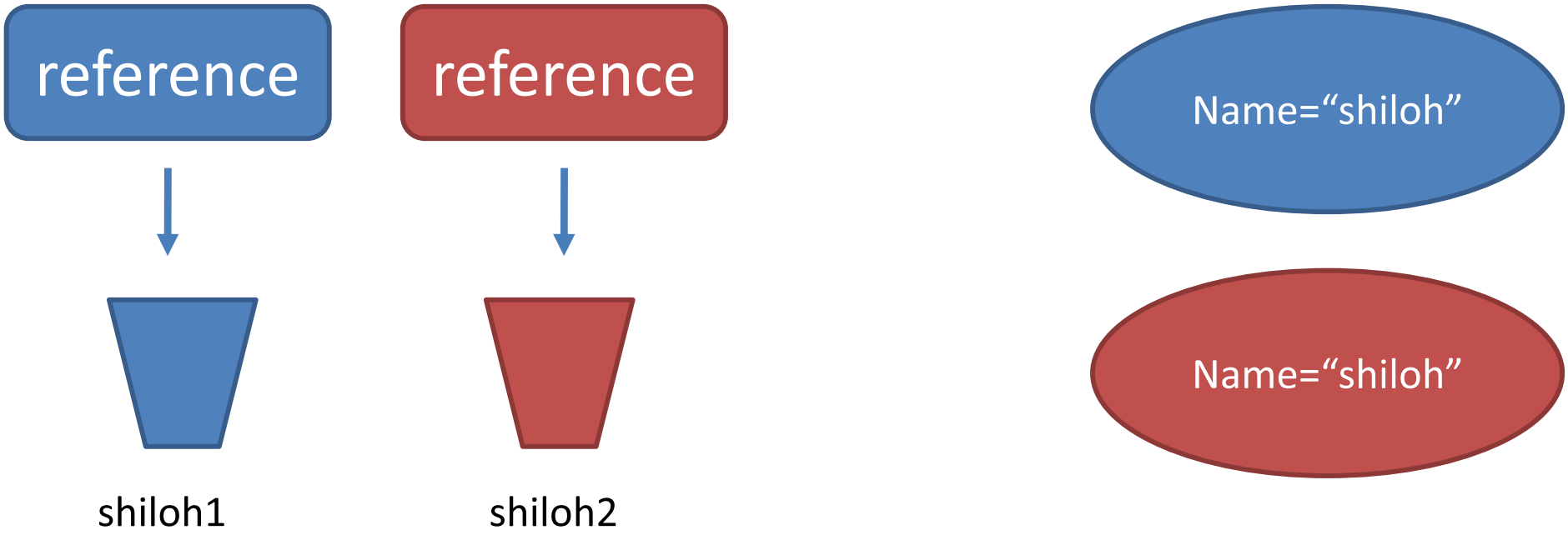
Primitives vs References

- 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, …

References

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



Public vs. Private

- Public: others can use this
- Private: only the class can use this

public/private applies to any
field or **method**

Mr. MeanGuy

```
public class Malicious {
   public static void main(String[] args) {
      maliciousMethod(new CreditCard());
   }
   static void maliciousMethod(CreditCard card)
   {
      card.expenses = 0;
      System.out.println(card.cardNumber);
   }
}
```

Access Control

```
public class CreditCard {
  String cardNumber;
  double expenses;
  void charge(double amount) {
    expenses = expenses + amount;
  }
  String getCardNumber(String password) {
    if (password.equals("SECRET!3*!")) {
      return cardNumber;
    }
      return "jerkface";
  }
}
```

Access Control DONE RIGHT

```
public class CreditCard {
  private String cardNumber;
  private double expenses;
  public void charge(double amount) {
    expenses = expenses + amount;
  }
  public String getCardNumber(String password)
  {
    if (password.equals("SECRET!3*!")) {
      return cardNumber;
    }
    return "jerkface";
  }
}
```

Why Access Control

- Protect private information (sorta)
- Clarify how others should use your class
- Keep implementation separate from interface

Inheritance

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()
```

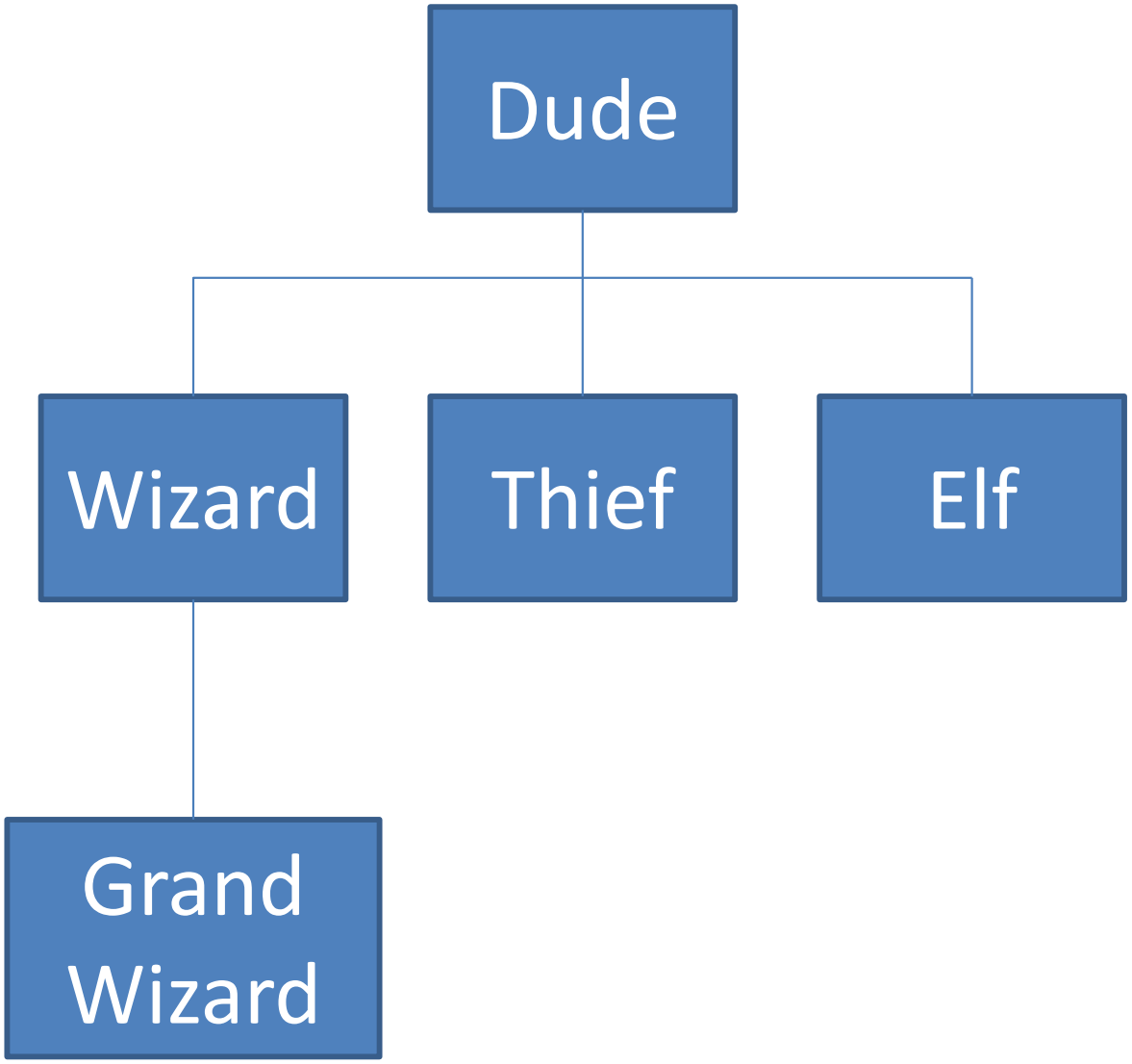
1. Cast to `Dude` tells Java to start looking in `Dude`
2. Look for `sayName()` in Dude class
3. Found it! Call `sayName()`

What's going on?

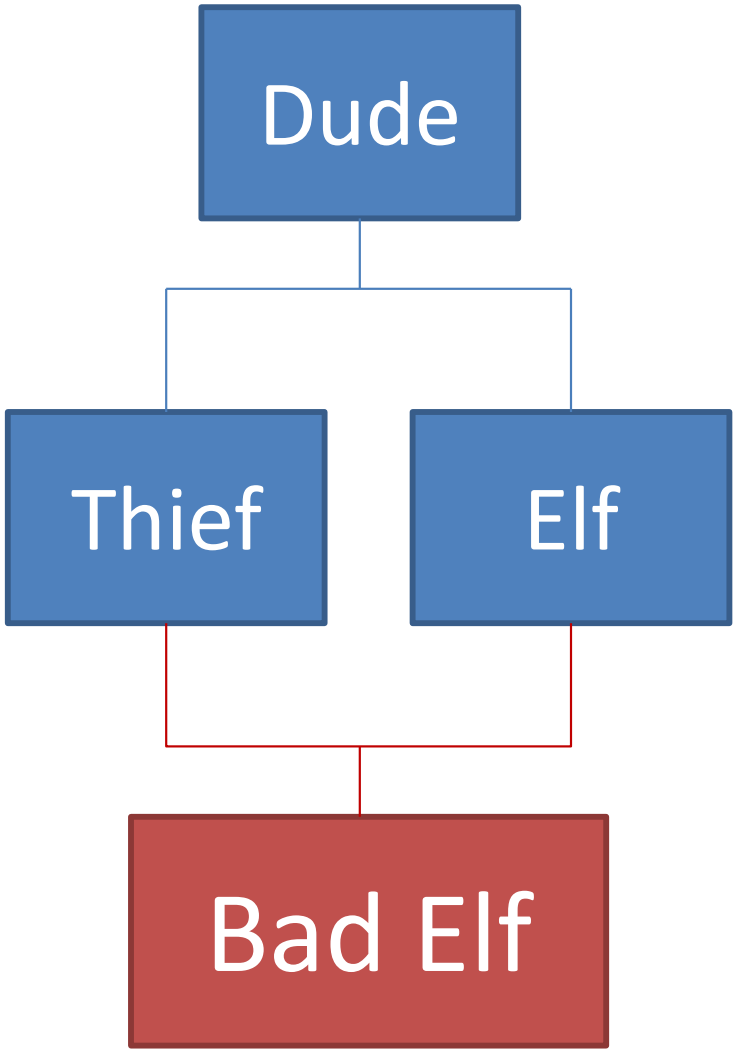
Parent of Wizard, Elf..

Subclass of Dude

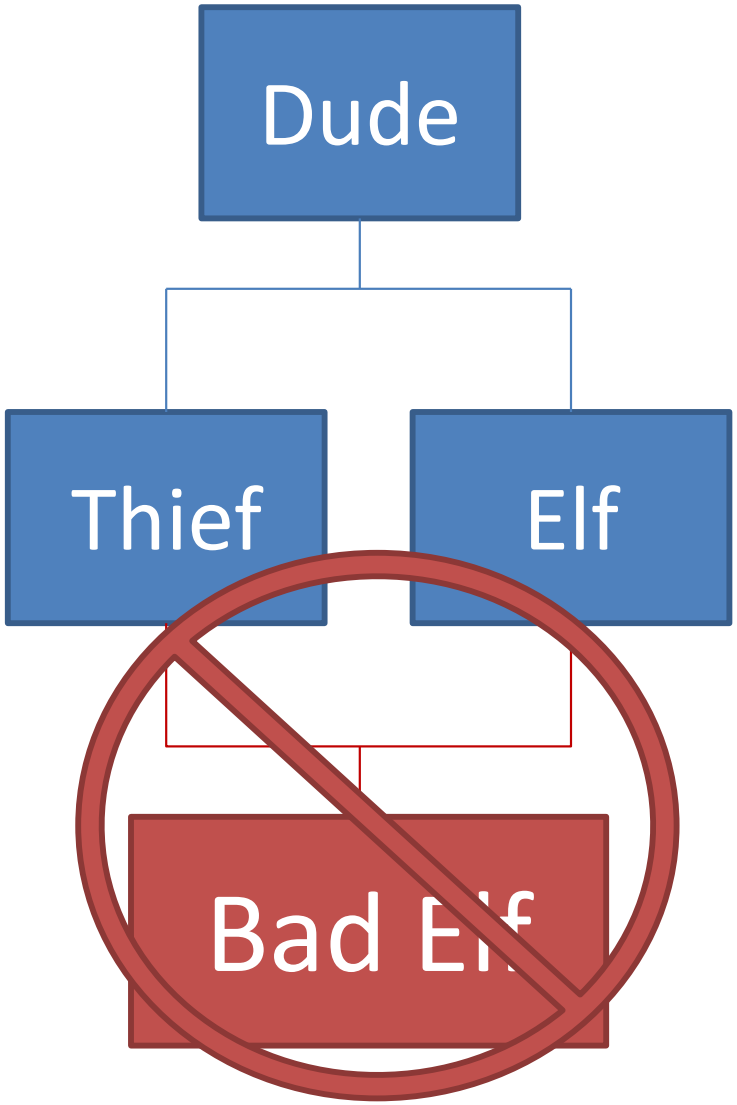
Subclass of Wizard



You can only inherit from one class



You can only inherit from one class



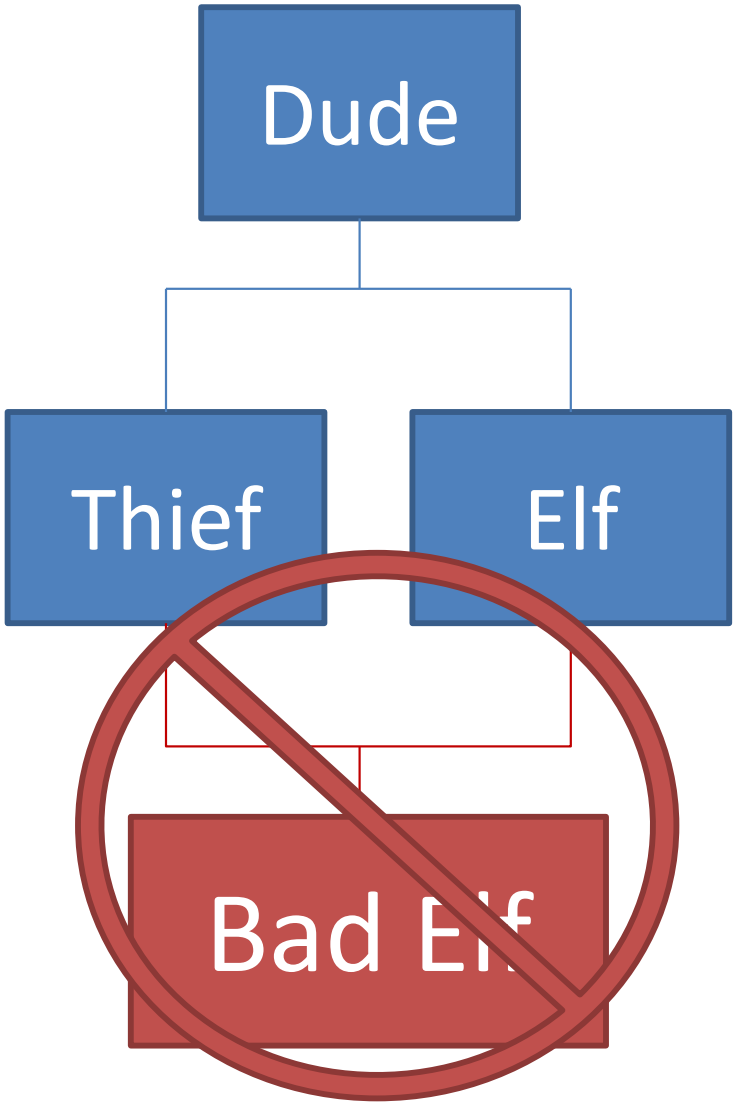
You can only inherit from one class

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

So much more to learn!

- http://java.sun.com/docs/books/tutorial/java/IandI/subclasses.html
- http://home.cogeco.ca/~ve3ll/jatutor5.htm
- http://en.wikipedia.org/wiki/Inheritance_(computer_science)
- http://www.google.com

Arrays with items

Create the array bigger than you need

Track the next “available” slot

```
Book[] books = new Book[10];
int nextIndex = 0;

books[nextIndex] = b;
nextIndex = nextIndex + 1;
```

Arrays with items

Create the array bigger than you need

Track the next “available” slot

```
Book[] books = new Book[10];
int nextIndex = 0;

books[nextIndex] = b;
nextIndex = nextIndex + 1;
```

What if the library expands?

ArrayList

Modifiable list

Internally implemented with arrays

Features

- Get/put items by index
- Add items
- Delete items
- Loop over all items

Array → ArrayList

```
Book[] books =
     new Book[10];
int nextIndex = 0;

books[nextIndex] = b;
nextIndex += 1;
```

→

```
ArrayList<Book> books
= new ArrayList<Book>();

books.add(b);
```

```java
import java.util.ArrayList;
class ArrayListExample {
    public static void main(String[] arguments) {
        ArrayList<String> strings = new ArrayList<String>();
        strings.add("Evan");
        strings.add("Eugene");
        strings.add("Adam");

        System.out.println(strings.size());
        System.out.println(strings.get(0));
        System.out.println(strings.get(1));

        strings.set(0, "Goodbye");
        strings.remove(1);
        for (int i = 0; i < strings.size(); i++) {
             System.out.println(strings.get(i));
         }
        for (String s : strings) {
            System.out.println(s);
        }
    }
}
```