Polymorphism

Polymorphism

Definition

Ability for different types of objects to feature their own version of a method – i.e. name + return type + parameters – which each behave differently

Replaces a switch on the object's type → Pseudo Code

```
switch(o.type){
  case Employee:     o.work();      break;
  case Lawyer:       o.sue();       break;
  case Secretary:    o.report();    break;
}
```

Rules to work w/ polymorphism

- A variable of type *T* can hold an object of any subclass of *T*.
  - E.g `Employee ed = new Lawyer();`
  - ed → Lawyer
  - ed → LegalSecretary

- You can call any methods from the `Employee` class on `ed`.

- When you do, it uses the overridden version, if any, in order to behaves as the subclass; e.g. `Lawyer`

```
System.out.println(ed.getSalary());          // 50000.0
System.out.println(ed.getVacationForm()); // pink
```

Example – Polymorphism and arrays

Arrays of superclass types can store any subtype as elements.

```
public static void main(String[] args) {
    Employee[] e = {new Lawyer(),   new Secretary(),
                    new Marketer(), new LegalSecretary()};

    for (int i = 0; i < e.length; i++) {
      System.out.println("pay  : " + e[i].getSalary());
      System.out.println("vdays: " + e[i].getVacationDays());
      System.out.println();
    }
}
```

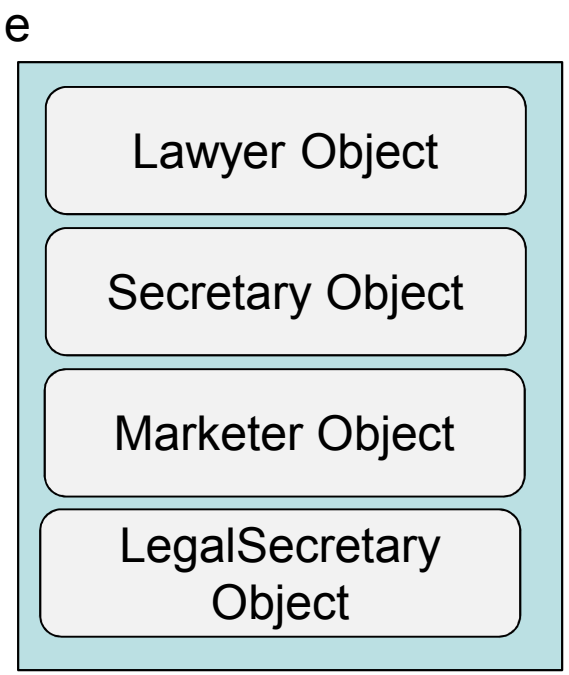
Output:

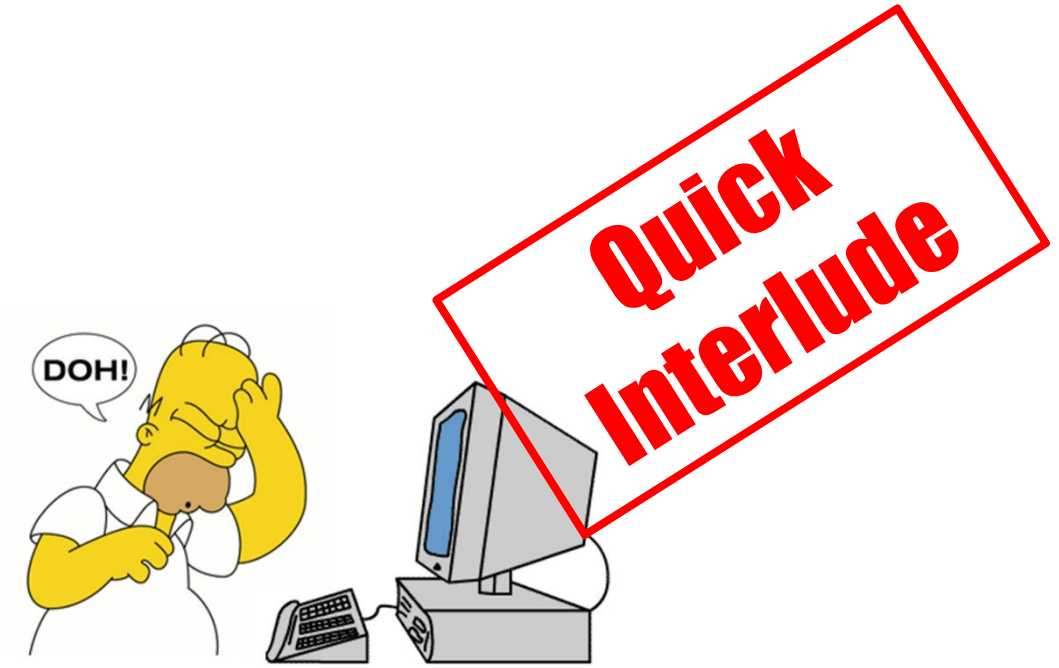
```
pay  : 50000.0
vdays: 15

pay  : 50000.0
vdays: 10

pay  : 60000.0
vdays: 10

pay  : 55000.0
vdays: 10
```

Side note: What does this look like in memory?

Memory Diagram of Previous Example

Everyone comfortable with this representation?

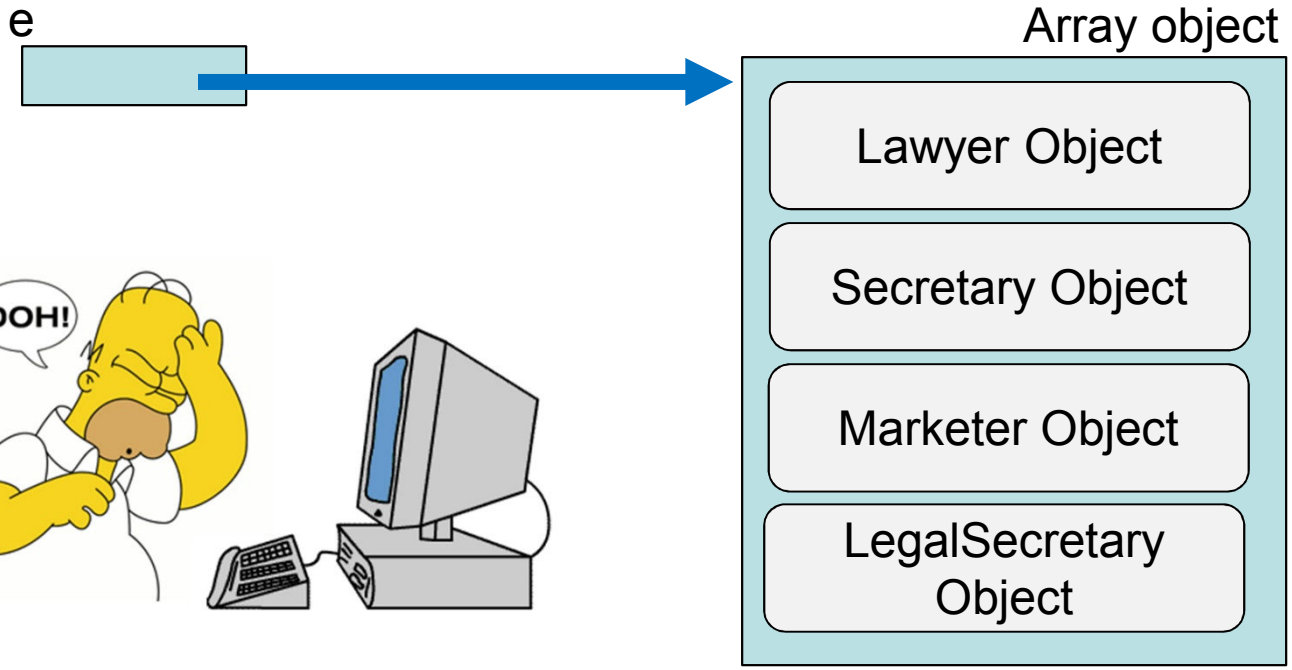





```
Employee[] e = {      new Lawyer(),    new Secretary(),
                      new Marketer(), new LegalSecretary()};
```

Memory Diagram of Previous Example

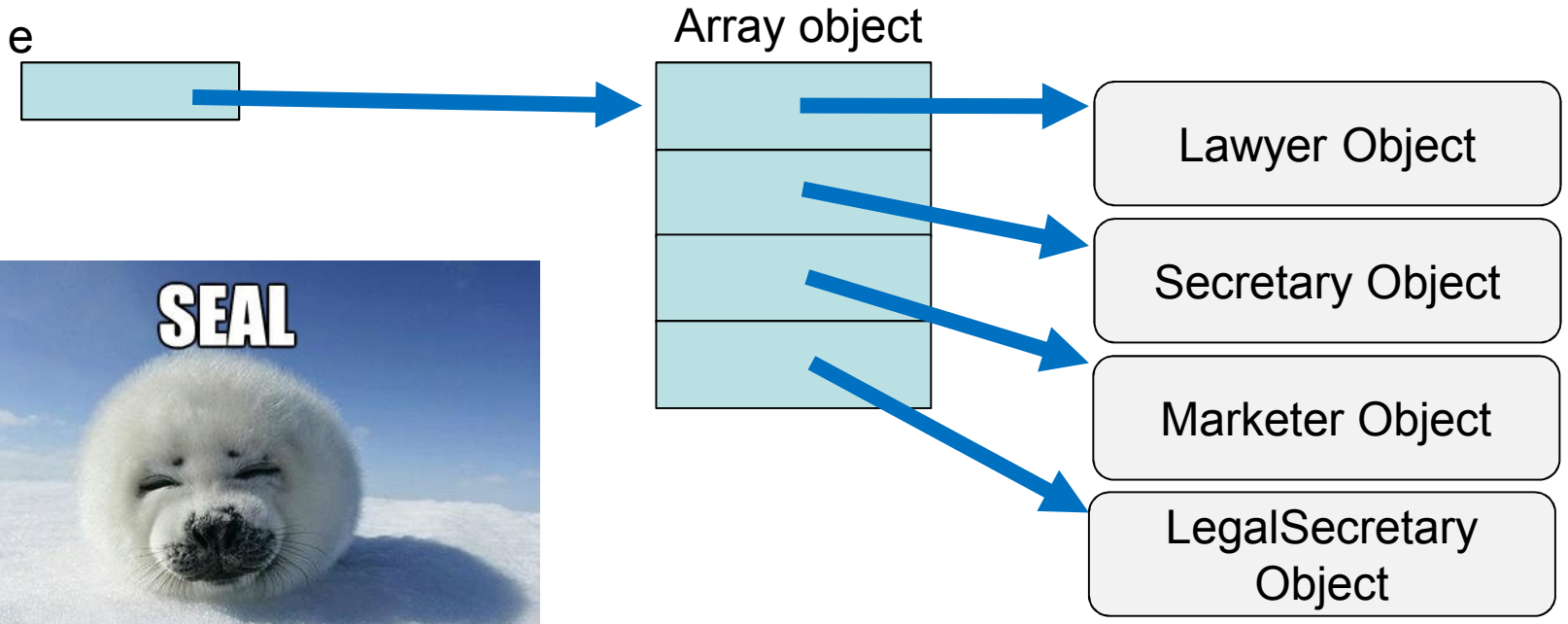
Everyone comfortable with this representation?




```
Employee[] e = {       new Lawyer(),    new Secretary(),
                       new Marketer(), new LegalSecretary()};
```

Memory Diagram of Previous Example

Everyone comfortable with this representation?




```
Employee[] e = {      new Lawyer(),    new Secretary(),
                      new Marketer(), new LegalSecretary()};
```

Now… SO FAR we used only Superclass methods on ed…

A variable can only call that type's methods, not a subtype's.

```
Employee ed = new Lawyer();
int hours = ed.getHours();  // ok; in Employee
ed.sue();                   // compiler error
```

Java's reasoning

= variable `ed` could store any kind of employee

not all kinds know how to `sue`

How do we use the subclass methods?

A variable can only call that type's methods, not a subtype's.

```
Employee ed = new Lawyer();
int hours = ed.getHours();   // ok; in Employee
ed.sue();                    // compiler error
```

Java's reasoning

= variable `ed` could store any kind of employee

not all kinds know how to `sue`

To use `Lawyer` methods on `ed`, we can type-cast it.

```
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue();                  // ok

((Lawyer) ed).sue();              // shorter version
```

Things to be careful about…

- The code crashes if you cast an object too far down the tree.

```
Employee eric = new Secretary();
((Secretary) eric).takeDictation("hi");     // ok
((LegalSecretary) eric).fileLegalBriefs(); // error
//   (Secretary doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```
Lawyer linda = new Lawyer();
((Secretary) linda).takeDictation("hi");    // error
```

- Casting doesn't actually change the object's behavior. It just gets the code to compile/run.

```
((Employee) linda).getVacationForm()        // pink
```

PopQuiz – Let's get "Creative"

```
import java.util.*;

class A {
   public String f(){return "A";}
}

public class B extends A {

   public String f(){return "B";}

   public static void main(String[] args){
      B b = new B();
      A a = (A) b;
      System.out.println(a.f());
   }
}
```

What's displayed?

B

http://stackoverflow.com/questions/22874640/overriding-methods-in-java-and-then-casting-object-to-parent-class-behavior