Remember our class hierarchy:

```
Employee
    Marketer
    Secretary
    LegalSecretary
    Lawyer
```

We can write:

```java
Employee e = new Lawyer();
```

Why? Because a Lawyer is an Employee. We can even write:

```java
e.getSalary();
e.getVacationForm();
```

because all Employees have getSalary() and getVacationForm() methods. The nice thing about polymorphism is that at runtime, Java will figure out that e is actually a Lawyer, and will call the appropriate Lawyer code.

We can’t write:

```java
e.takeDictation();
```
Because `e` is an `Employee`, and while certain `Employee`es (*i.e.*, those of type `Secretary`) can `takeDictation()`, not all `Employee`s can.

For that matter, we can’t even write:

```java
    e.sue();  // compiler error
```

even though the underlying type of the object that `e` points to is in fact a `Lawyer`. Again, because `e` is of type `Employee`, we can only call methods common to *all* `Employee`s.

If we know for sure that the underlying object that `e` points to is a `Lawyer`, we can do a typecast.

```java
    Lawyer vinny = (Lawyer)e;
    vinny.sue();
```

We could also have written:

```java
    Employee e = new Lawyer();
```

```java
    ...
    ((Lawyer)e).sue();
```

Remember that we did casts earlier in the semester for things like:

```java
    int x = 5;
    int y = 2;

    double d = x/(double)y;
```

to convert `y` into a double to force floating-point division.

We can also write something like this:

```java
    Employee team[] = new Employee[3];

    team[0] = new Marketer();
```
team[1] = new Lawyer();
team[2] = new LegalSecretary();

for (int i=0; i<team.length; i++)
    System.out.println(team[i].getSalary());

Again, this works because all Employees have a getSalary(). The compiler makes sure that every element of team is of type Employee (so that we know that they can all getSalary()), and at runtime, the version of getSalary() appropriate for the object type is called, e.g., Marketer’s getSalary() for team[0].