Inheritance and Polymorphism

(with Pokemon)

OOP

- Object-oriented programming is another paradigm that makes objects its central players, not functions.
- Objects are pieces of data and the associated behavior.
- Classes define an object, and can inherit methods and instance variables from each other.

Inheritance

Occasionally, we find that many abstract data types are related.

For example, there are many different kinds of people, but all of them have similar methods of eating and sleeping.

Inheritance

We would like to have different kinds of Pokémon, which differ (among other things) in the amount of points lost by its opponent during an attack.

The only method that changes is `attack`. All the other methods *remain the same*. Can we avoid *duplicating code* for each of the different kinds?

Inheritance

Key OOP Idea: Classes can inherit methods and instance variables from other classes

```
public class WaterPokemon extends Pokemon
{
    …
    void attack(Pokemon other)
    {
        other.decrease_hp(75);
    }
}
```


Inheritance

Key OOP Idea: Classes can inherit methods and instance variables from other classes

```
public class WaterPokemon extends Pokemon
{
    …
    void attack(Pokemon other)
    {
        other.decrease_hp(75);
    }
}
```

The Pokemon class is the ***superclass*** of the WaterPokemon class.

The WaterPokemon class is the ***subclass*** of the Pokemon class.

Inheritance

Key OOP Idea: Classes can inherit methods and instance variables from other classes

```
public class WaterPokemon extends Pokemon
{
    …
    void attack(Pokemon other)
    {
        other.decrease_hp(75);
    }
}
```

The attack method from the Pokemon class is ***overridden*** by the attack method from the WaterPokemon class.

7

Inheritance

```
WaterPokemon ashs_squirtle = new
      WaterPokemon("Squirtle","Ash", 314);
Pokemon mistys_togepi = new Pokemon("Togepi",
      "Misty", 245);
mistys_togepi.attack(ashs_squirtle);
System.out.println(ashs_squirtle.getHitPts());
264
ashs_squirtle.attack(mistys_togepi);
System.out.println(mistys_togepi.getHitPts());
170
```

8

Inheritance

```
WaterPokemon ashs_squirtle = new
      WaterPokemon("Squirtle","Ash", 314);
Pokemon mistys_togepi = new Pokemon("Togepi",
      "Misty", 245);
mistys_togepi.attack(ashs_squirtle);
System.out.println(ashs_squirtle.getHit
264
ashs_squirtle.attack(mistys_togepi);
System.out.println(mistys_togepi.getHitPts());
170
```

mistys_togepi uses the attack method from the Pokemon class.

9

Inheritance

```
WaterPokemon ashs_squirtle = new
      WaterPokemon("Squirtle","Ash", 314);
Pokemon mistys_togepi = new Pokemon("Togepi",
      "Misty", 245);
mistys_togepi.attack(ashs_squirtle);
System.out.println(ashs_squirtle.getHitPts());
264
ashs_squirtle.attack(mistys_togepi);
System.out.println(mistys_togepi.getHi
170
```

ashs_squirtle uses the attack method from the WaterPokemon class.

10

Inheritance

```
WaterPokemon ashs_squirtle = new
      WaterPokemon("Squirtle","Ash", 314);
Pokemon mistys_togepi = new Pokemon("Togepi",
      "Misty", 245);
mistys_togepi.attack(ashs_squirtle);
System.out.println(ashs_squirtle.getHitPts());
264
ashs_squirtle.attack(mistys_togepi);
System.out.println(mistys_togepi.getHit
170
```

The WaterPokemon class does not have a getHitPts method, so it uses the method from its superclass.

11

Review: Inheritance

If the class of an object has the method or attribute of interest, that particular method or attribute is used.

Otherwise, the method or attribute of its parent is used.

Inheritance can be many levels deep.

If the parent class does not have the method or attribute, we check the parent of the parent class, and so on.

12

Review: Inheritance

We can also both *override* a parent's method or attribute *and* use the original parent's method.

Say that we want to modify the attacks of Electric Pokémon: when they attack another Pokémon, the other Pokémon loses the *original* 50 HP, but the Electric Pokémon gets an increase of 10 HP.

13

Review: Inheritance

```
public class ElectricPokemon extends Pokemon
{
    …
    void attack(Pokemon other)
    {
        super.attack(other);
        increase_hp(10);
    }
}
```

We use the original attack method from the parent.

14

Polymorphism

Write the method `attackAll` for the Pokemon class that takes an array of Pokemon objects as its argument. When called on a Pokemon object, that Pokémon will attack each of the Pokémon in the provided array. (Ignore the output printed by the `attack` method.)

```
>>> ashs_pikachu = ElectricPokemon('Pikachu', 'Ash', 300);
>>> ashs_squirtle = WaterPokemon('Squirtle', 'Ash', 314);
>>> mistys_togepi = Pokemon('Togepi', 'Misty', 245);
>>> mistys_togepi.attackAll([ashs_pikachu, ashs_squirtle]);
>>> ashs_pikachu.getHitPts();
250
>>> ashs_squirtle.getHitPts();
264
```

15

Polymorphism

```
public class Pokemon
{
    …
    void attackAll(Pokemon[] others)
    {
        for(Pokemon other : others)
            attack(other);
    }
}
```

16

Polymorphism

```
for(Pokemon other : others)
    attack(other);
```

other can be an object of many different data types: `ElectricPokemon`, `WaterPokemon`, and `Pokemon`, for example.

17

Polymorphism

```
for(Pokemon other : others)
    attack(other);
```

`attack` can work on objects of many *different* data types, without having to consider each data type separately.

`attack` is *polymorphic*.

poly = Many
morph = Forms

18

Polymorphism

Write the method `attackedBy` for the `Pokemon` class that takes an array of `Pokemon` objects as its argument. When called on a `Pokemon` object, that Pokémon will be attacked by each of the Pokémon in the provided array.

```
>>> ashs_pikachu = ElectricPokemon('Pikachu', 'Ash', 300);
>>> ashs_squirtle = WaterPokemon('Squirtle', 'Ash', 314);
>>> mistys_togepi = Pokemon('Togepi', 'Misty', 245);
>>> ashs_squirtle.attackedBy([ashs_pikachu, mistys_togepi]);
>>> ashs_squirtle.getHitPts();
204
```

19

Polymorphism

```
public class Pokemon
{
    …
    void attackedBy(Pokemon[] others)
    {
        for(Pokemon other : others)
            other.attack(this);
    }
}
```

20

Polymorphism

```
for(Pokemon other : others)
    other.attack(this);
```

other can be an object of many different data types: `ElectricPokemon`, `WaterPokemon`, and `Pokemon`, for example.

It can also be an object of *any other class* that has an `attack` method.

21

Polymorphism

Key OOP idea: The same method can work on data of *different types*.

We have seen a polymorphic function before:

```
3 + 4
→7
'hello' + ' world'
→'hello world'
```

The + operator, or the add function, is *polymorphic*. It can work with both numbers and strings.

22

Everything is an Object

Everything (except primitives) in Java is an object.

In particular, every class is a subclass of a built-in `Object` class.

This is *not* the term 'object', but a class whose *name* is `Object`.

23

Everything is an Object

The `Object` class provides extra methods that enable polymorphic methods like `equals` or `toString` to work on many different forms of data.

24

Everything is an Object

Implement the method `equals` for the Pokemon class that will allow us to check if one Pokémon is "equal" to another, which means that it has the same name as another.

```
ashs_pikachu = ElectricPokemon('Pikachu', 'Ash', 300);
brocks_pikachu = ElectricPokemon('Pikachu', 'Brock', 300);
ashs_pikachu.equals(brocks_pikachu);
→true
```

25

Everything is an Object

Implement the method `equals` for the Pokemon class that will allow us to check if one Pokémon is "equal" to another, which means that it has the same name as another.

```
public class Pokemon
{
    …
    boolean equals(Pokemon p)
    {
        return _____________________________________;
    }
}
```

26

Everything is an Object

Implement the method `equals` for the Pokemon class that will allow us to check if one Pokémon is "equal" to another, which means that it has the same name as another.

```
public class Pokemon
{
    …
    boolean equals(Pokemon p)
    {
        return this.name.equals(p.name);
    }
}
```

27

Everything is an Object

The `Object` class provides a default `equals` method, but the `equals` method we defined for the Pokemon class *overrides* the method provided by the `object` class, as we would expect from inheritance.

Bottom line:

Inheritance and polymorphism in OOP allow us to *override standard methods*!

28

Everything is an Object

Side-note:

Every class inherits from the `object` class, including the Pokemon class.

`public class Pokemon`

is shorthand for

`public class Pokemon extends Object`

29

Conclusion

- Inheritance and polymorphism are two key ideas in OOP.
  - Inheritance allows us to establishes relationships (and reuse code) between two similar data types.
  - Polymorphism allows functions to work on many types of data/objects.
- (Almost) Everything is an object.
- OOP allows us to override standard methods.

30