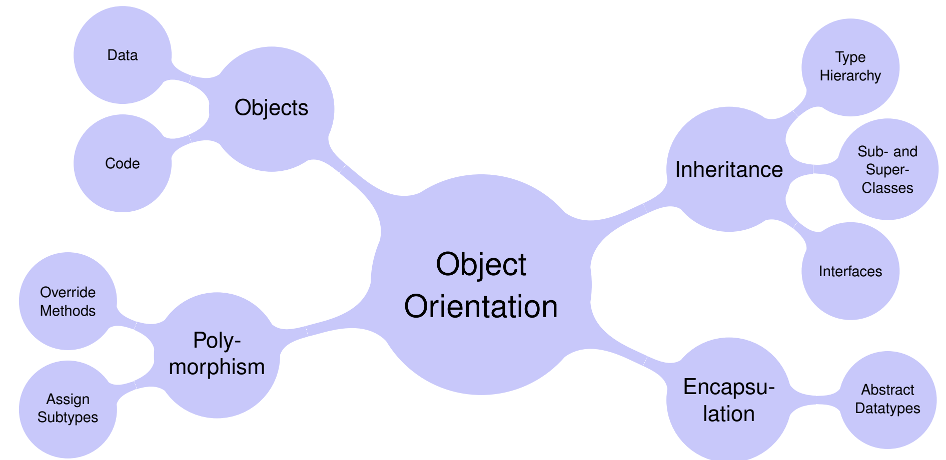
15. Java Object Orientation II

Polymorphism

Object Orientation: Different Aspects



Polymorphism

- **Override Methods:** Inherited methods from a superclass can be overriden: Same signature, new code.
- **Variable Assignment:** Objects of a given type can be assigned to variables of any supertype.

Overriding Methods

Inherited methods of a supertype can get a new implementation. Same *signature*, new *code*.

We remember the method `alarm()` for last time. This abstract method was defined in class `Wind` as follows

```
class Wind extends Measurement {
    int speed;
    ...
    boolean alarm(){    // implements abstract method alarm()
        return this.speed > 80;
    }
}
```

Overriding Methods

We define a new subclass `WindWithGusts`, that also tracks *gusts* in addition to windspeed and direction.

```
/*
 * Fancy windsensor data that also tracks gusts. Requires special hardware.
 */
class WindWithGusts extends Wind {
    int gusts;

    @Override
    boolean alarm(){    // replaces implementation of supertype
        return this.speed > 80 || this.gusts > 20;
    }
}
```

Inherited from `Wind` (↑ this.speed)

Access to Overriden Method: `super` Keyword

A subclass doesn't have to repeat the code that is being overridden.

⇒ Call of the overriden implementation using keyword `super`, *but only within the overriding implementation*

```
class WindWithGusts extends Wind {
    int gusts;

    @Override
    boolean alarm(){    // replaces implementation of supertype
        return super.alarm() || this.gusts > 20;
    }
}
```

(↑ super.alarm()) Executes: `this.speed > 80 ;`

Access to Constructors of Superclass

Setting: Creation of a measurement always requires a coordinate.
→ Constructor in class `Measurement`

```
class Measurement {
    Coordinate position;

    Measurement(float lat, float lon){
        this.position = new Coordinate(lat, lon);
    }
    ...
}
```


Access to Constructors of Superclass

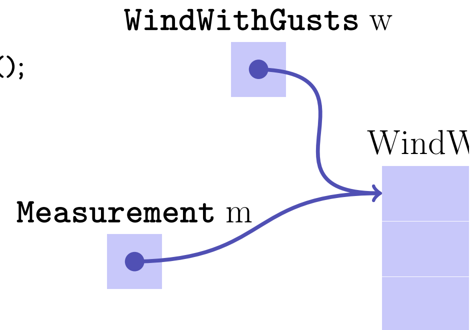
- Using keyword `super`, a constructer of a superclass can be called.
- The amount and types of the arguments determines *which* constructer will be called
- Calling `super(...)` *must* be the first instruction!

```
class Wind {
    ...
    Wind(float lat, float lon, int speed, int direction ){
        super(lat, lon);
        this.speed = speed;
        this.direction = direction;
    }
    ...
}
```

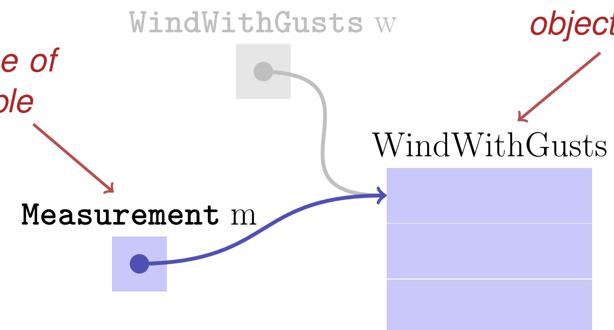
Polymorphic References

Variables of a declared type can reference objects of a subtype.

```
WindWithGusts w = new WindWithGusts();

Measurement m;

m = w; // polymorphic reference!


// But this doesn't compile: w = m;
```



Static vs. Dynamic Type

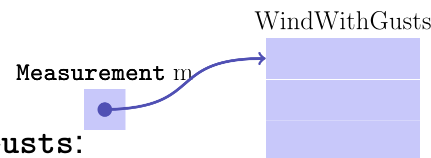


Dynamic Methodbinding

When calling a method, the implementation of the *dynamic type* is executed!

Call:

```
m.alarm();
```



⇒ Executed code from class `WindWithGusts`:

```
@Override
boolean alarm(){
    return super.alarm() || this.gusts > 20;
}
```


Usages for dynamic binding

Given: A list of different kinds of measurements (*Temperatures, Wind, ...*

Wanted: A list of measurements that cause an alarm.

```
void filterByAlarm(Measurement[] measurements){
    for (int i = 0; i < measurements.length; ++i){
        if (measurements[i].alarm()){      //dynamic method binding!
            measurements[i]=null;          //remove from array
        }
    }
}
```