15. Java Object Orientation II

**Polymorphism**

- **Override Methods**: Inherited methods from a superclass can be overridden: 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:

```java
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.

```java
/*
 * 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;
    }
}
```

**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

```java
class WindWithGusts extends Wind {
    int gusts;

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

**Access to Constructors of Superclass**

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

```java
class Measurement {
    Coordinate position;

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

**Using keyword super, a constructor of a superclass can be called.**

- The amount and types of the arguments determines which constructor will be called
- Calling `super(...)` *must* be the first instruction!
Polymorphic References

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

```java
WindWithGusts w = new WindWithGusts();
Measurement m;
m = w; // polymorphic reference!
```

// But this doesn’t compile: w = m;

---

Static vs. Dynamic Type

- **WindWithGusts**
- **Measurement**

**Static type of the variable**

**Dynamic type: type of the referenced object**

---

Dynamic Methodbinding

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

- Call: `m.alarm();`

⇒ Executed code from class WindWithGusts:

```java
@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.

```java
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
    }
}
}
```