**OOP Theory. Study notes. Version Two**

In OOP we create **objects** from a template class – A class has **properties (fields)** and **method**s.

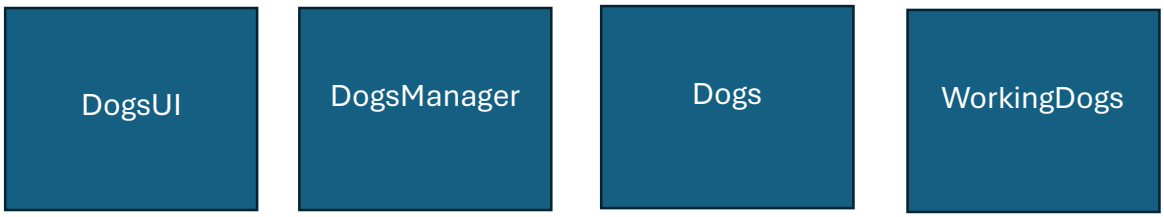
Both properties and methods are kept "safe" from abuse by the **access modifiers**

- **public** – directly accessible in any class in the same packate
- **private** - directly accessible only in the class in which they are defined
- **protected** - directly accessible in the class in which they are defined as well as any class which inherits from it and by any class in the same package.

**Properties** are variables that can be changed/edited/updates unless the property is a constant; in Java we call this FINAL.

**Four classes –**

1. A **UI** class with the main method. This is the point of entry into the program. It creates the manager object.
2. A **manager class** with all the methods, especially the **constructor**[1] method. We will use the constructor method to find and read in the text file into an array of dog objects
3. A **Dog class** from which to instantiate Dog objects. The Dog class has properties and methods.
4. A **WorkingDog class** from which to instantiate WorkingDog objects. The WorkingDog class inherits all the properties and methods of the Dog class but adds some additional properties and methods of its own.




**Static and non-static properties and methods.**

Most properties and methods belong to the created object. If however, the property or method belongs to the class it is said to be **static**.

- Properties (and methods) can be **static** in which case they can be accessed in the following way **<ClassName.variable>** - this is because they belong to the class.
- Properties (and methods) can be **non-static** in which case they can only be called via a created object – they belong to the object. **<ObjectName.variable>** but this seldom works because the objects variable are usually private and we have to use getter and setter methods instead (accessor and mutator methods)

[1] **The constructor has the same name as the class. It has no return type. It runs automatically. It is always public. Its job is to instantiate new objects and to initialize the starting values of the various properties.**

**Examples:**

| | |
|---|---|
| `String theName = dogArray[10].getName;` | **Non static method.** Here we are getting the name of dog number 10 in the array. |
| `dogArray[15].setAge(6);` | **Non static method**. Here we are changing the age of dog number 15 to 6 years. |
| `Dogs.maxAge;` | **Static field**. The field maxAge which all created dog objects have access to. |
| `WorkingDogs. minAge;` | **Static field**. The field minAge which all the created workingDog objects have access to. |

**Example 2:**

The for loop below tests all dogs' ages against the min and max age values. The max and min age of **individual** dogs is being tested against the static property of min and max ages that apply to all dogs.

```
public String testAge()
{
    boolean flag1 = false, flag2 = false;
    String flag_x2 = "";

    for(int x = 0; x < counter; x++)
    {
        if(dogArray[x].getAge() < Dogs.maxAge)
            flag1 = true;
        if(dogArray[x].getAge() > WorkingDogs.minAge)
            flag2 = true;
        flag_x2 = flag_x2 + flag1 + " " + flag2 + "\n";

       // reset the boolean flags
       flag1 = false;
       flag2 = false;
    } // end for loop
    return flag_x2;
} // end testAge
```

**Methods** inside a class achieve a unique task.

- They can accept a parameter, or not.
- They can return a value, or not.
- They can accept a parameter and return a value.

Methods that return a value are **typed** – they must end with a "return" statement. Methods that return nothing are **void**.

Consider the **class diagrams that follow (UML diagram)**

| **StudentManager** | |
|---|---|
| **Properties**<br>- studentArray : Student [ ] // private non static property<br>- size : integer // private non static property<br>+ DISCOUNT1 = 0.05 : real // public static final property<br>+ DISCOUNT2 = 0.1 : real // public static final property | |
| **Methods**<br>+ Constructor ( )<br>+ displayAll( ) : string<br>+ generateQRCode(name) : image // a static method to generate a QR code for a student | |
| **NOTES on the UML diagram above.** | |
| StudenManager sm = new StudentManager( ) | Creates a **single default** StudentManager object. |
| sm.displayAll( ) | The **non static** method "displayAll" belongs to the created object and is therefore called using the objects name. |
| feeReduction = StudentManager.DISCOUNT1 * 2025_fees; | The **static property** DISCOUNT1 can be called directly anywhere because it is public and belongs to the class. |

| **Student** |
|---|
| **Properties:**<br>- name : string<br>- dob : date<br>- gender : char<br>- grade : integer<br>- bursary : boolean |
| **Methods:**<br>+ Constructor(n string, dob date, ge char, gr integer, b boolean) //**overloaded constructor**<br>+ Constructor(n string, dob date, ge char) // **overloaded constructor**<br><br>+ getName( ) : string // **accessor and mutator methods**<br>+ setname(name : string)<br>+ getDOB( ) : date<br>+ setDOB(dob :date)<br>+ getGender( ) : character<br>+ setGender(gender : char)<br>+ getGrade( ) : integer<br>+ setGrade(grade : integer)<br>+ getBursary( ) : boolean<br>+ setBursary(bursary : boolean)<br><br>+ calculateFees( ) : real<br>- determineDiscount( ) : real // **private helper method** for calculateFees<br><br>+ toString ( ) : string |

| **NOTES on the UML diagram above** | |
|---|---|
| `studentArray [x] = new Student(name, dob, gender, grade, bursary) ;` | Creates a unique **parameterized** Student object and places it into an array of Student objects at the position of the **index** "x". Uses one of the overloaded constructor methods. You will find this code snippet inside a loop. |
| `studentArray [x] = new Student(name, dob, gender) ;` | Creates a unique **parameterized** Student object and places it into an array of Student objects at the position of the **index** "x". Uses the other overloaded constructor method. You will find this code snippet inside a loop. |
| `Student theStudent = new Student(name, dob, grender, grade, bursary);`<br><br>`studentArray[x] = theStudent;` | This creates a **single** unique instance of Student which is then copied into the array of Students at index "x" |

| | |
|---|---|
| - name : string<br>- dob : date<br>- gender : char<br>- grade : integer<br>- bursary : Boolean | Because all the properties are private, you need accessor and mutator methods to work with their values (getter and setter methods) and they need to be public so that they can be called. |
| Never deviate from the class diagram i.e. the exact order, the same names and the same datatypes. | |

**Constructors**

Constructors are a special type of method. They are always public and have no return type. They also have the **same name** as the class where they are found. They are the only method that starts with a capital letter.

1. Are used to **instantiate** objects.
2. Are used to **initialize** the starting values of the properties
    a. Are used to assign **default** values to fields
    b. Are used to assign **specific** values to fields

**Default constructors** have no parameters. Therefore, the properties are set to default values (often zero, "" for the empty string, false for boolean or special predetermined values.

E.g. Googa bug = new Gogga( ) ; // the call to the default constructor

In grade 9 CoRo this creates a new Googa object that is red, it faces north, and it is placed into the middle of the playing grid – the default values.

E.g. Dog theDog = new Dog ( );

This could perhaps create a dog with no name, no age and a pedigree of false.

**Parameterized constructors** have parameters and initialize the starting values of the properties to a unique combination of values making the object "unique"

E.g. Googe bug2 = new Gogga( 5, 7, LEFT, yellow); // the call to the 4 parameter constructor

This creates a unique Googa object that is yellow, it faces west, and its placed at the intersection of 5 and 7 on the playing grid.

In the examples above two different Googa objects were created; this is an example of **polymorphism**.

**Methods**

- **Constuctor** method is always public and has no return type.
- **Accessor** methods are always typed
- **Mutator** methods are generally void
- **Helper** methods are private and assist another method in the same class achieved its task.
- **toString** method – this returns the selected properties as a single string. The toString method is always typed. The toString method is highly customizable, giving some or all of the properties in a **unique format** as asked for.

**Method headings** are generally made up of 4 parts. We can recognize a method because it always has **round brackets**

1. The accessor modifier
2. The return type (which can also be void)
3. The name of the method – always a small letter
4. The parameter list (also called the arguments)

E.g. public void calculateDividend(36) { } // no return type.

**Method headings** can have 5 parts if they are static

1. The accessor modifier
2. static
3. The return type (which can also be void)
4. The name of the method – always a small letter
5. The parameter list (also called the arguments)

E.g. public static double calculateDividend(36) { }

**Overloading** is when different methods have the same name but different parameters; this is often used when multiple constructors are needed

1. Different numbers of parameters
2. Different data types
3. Different order of the data types

**Overriding** is when the same method exists in two different classes i.e. the same name, signature and parameter list. In the example below the child case overrides the toString method of its parent class

```
@Override
public String toString( ) {    . . . . return theString }
```

**The four pillars of OOP**

| Encapsulation | Abstraction | Inheritance | Polymorphism |
|---|---|---|---|

**Encapsulation** – When properties and methods are "bound" together into one secure object. Access modifiers are used to keep the values of the object secure. Encapsulation is an example of "**information hiding**" – keeping properties and methods hidden (private) – see abstraction below.

**Abstraction** – the programmer hides all but the relevant data about an object in order to reduce complexity and increase efficiency. Focus on what is relevant in the current context and ignore (hide) that which is unimportant (in the current context)

**Inheritance** - This is when a child class inherits the properties and methods of the parent class; also called the super class and the subclass. This saved RAM, adds to the **reusability of code**, saves development time, saves money and makes **updating** the whole application easier and less error prone.

**Polymorphism** – a variable or object that can take on many forms. This is achieved by having multiple constructors. **Method overloading is an example of polymorphism.**

More on inheritance

The java reserved words for inheritance are

| | |
|---|---|
| **extends** | `public class WorkingDog extends Dog  {  }` |
| **super** | `public WorkingDog( String n, int a, boolean p, String t )`<br>`{`<br>`  super(n, a, p)`<br>`  this.trainer = t;`<br>`}`<br><br>`@Override`<br>`public String toString( )`<br>`{`<br>`  return super.toString( ) + "\t" + trainer;`<br>`}` |
| **instanceof** (note spelling and case) | `for(int x = 0; x < countAll; x++)`<br>`{`<br>`  if (dogArr [x] . instanceof (Dog)`<br>`  countDog = countDog + 1;`<br>`  else`<br>`  countWorkingDog = countWorkingDog + 1;`<br>`}` |
| | |