What is an object?

Objects are the key ingredients to understand the object-oriented technology. We can find many examples of real objects around us: a video projector, a computer, a desk, etc. Real objects are characterize by their statuses and behaviors. For instance, a dog has statuses: name, coloration and behaviors: barking, fetching and tail wagging. A bike has also its statuses: current gear, current speed, size of wheel and behaviors: change of gear, breaking, change of speed. Identification of statuses and behaviors is the key to thinking in terms of the object-oriented programming.

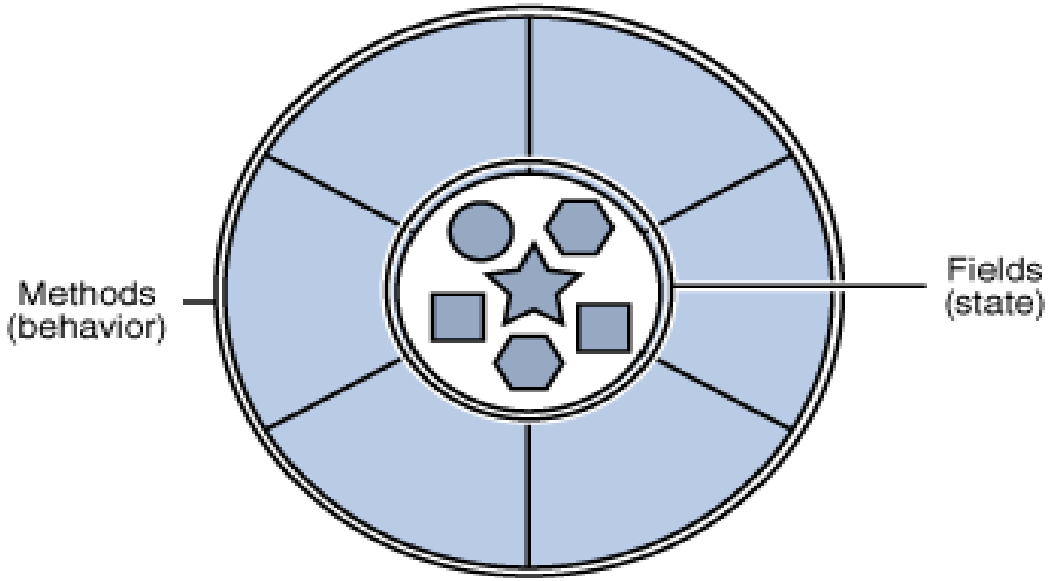
Exercise 1. Write down some real objects together with their statuses and behaviors. What did you notice?

What is an object?

During the exercise you noticed that objects differ in terms of their complexity. For instance, a desk lamp has only two statuses: on and off and two behaviors: turn on and turn off. On the other hand, a radio has more statuses: on, off, volume, frequency and behaviors: turn on, turn off, change volume, change frequency. We can also notice that some objects includes other objects or they are special types of another objects. This observation can be easily translated to the object-oriented technology. These are the main advantages of this approach.

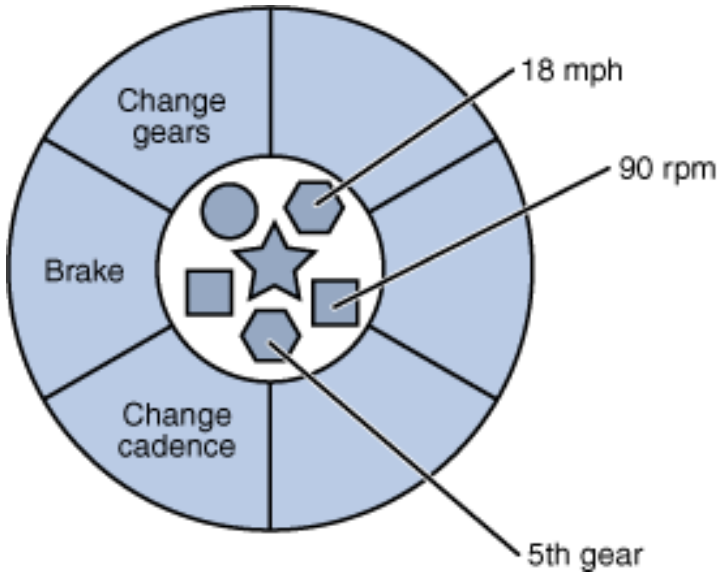
What is an object?

Objects represented in computer programs are very similar to real objects. They also have statuses and connected with them behaviors. Such objects stores their statuses in fields – variables and and they realize their behavior via methods – functions. Methods work on internal status of objects and they implements a fundamental mechanism of communication between objects. The strategy is that we hide statuses inside an object and we allow any interaction with the outside world only via methods. This mechanism is called *encapsulation*.



Scheme of an object represented in a computer programming language.

What is an object?



A model of a bike represented in a computer programming language.

An object in the object-oriented programming stays under our control thanks to its fields which store information about it and methods which allow us to change, in a controlled way, object's fields. So, fields and methods define how one can use an object. For instance, if an object which represents a bike has only six gears then methods should prevent attempts to set the gear status to a value lower than 1 and higher than 6.

Advantages of object-oriented programming

- **Modularity:** The source code written for each object can be stored separately – independent from the source code of other objects
- **Information hiding:** Via hiding object fields and only allowing interactions via methods the internal implementation remains hidden from the external world
- **Code reuse:** If an object already exists–maybe written by another programmer–we can reuse tits code. This allows specialists to implement a very specific code and ship it such that non-specialists can use it
- **Interchangeability:** When an object already exists in a program and it does not work properly, anymore, we can easily replace it by a different one without affecting another parts of our program

What is a class?

In the real world we can often identify classes of similar objects. For example, bikes of the same type. Each bike which belongs to the same class was built based on the same project and consists of the same components. We will say that a given object is an *instance* of a particular class. A class can be treated as a template from which one can create objects.

An example of the *Bicycle* in C++

```
class Bicycle {
private:
       int speed;
       int gear;

public:
       void changeGear(int newValue) {
            gear = newValue;
       }

       void speedUp(int increment) {
            speed = speed + increment;
       }

       void applyBrakes(int decrement) {
            speed = speed - decrement;
       }

       void printStates() {
         cout << "speed: " << speed << " gear:
              " << gear << endl;
       }
      Bicycle()
      {
           speed = 0;
           gear = 1;
      }
};
```

How to create an instance of the class *Bicycle*?

```
int main (int, char**)
{
// Create two different Bicycle objects
   Bicycle bike1;
   Bicycle bike2;
// Invoke methods on those objects
   bike1.speedUp(10);
   bike1.changeGear(2);
   bike1.printStates();

   bike2.speedUp(10);
   bike2.changeGear(2);
   bike2.speedUp(10);
   bike2.changeGear(3);
   bike2.printStates();
   // bike1.speed = 0;
return 0;
}
```

Program's output

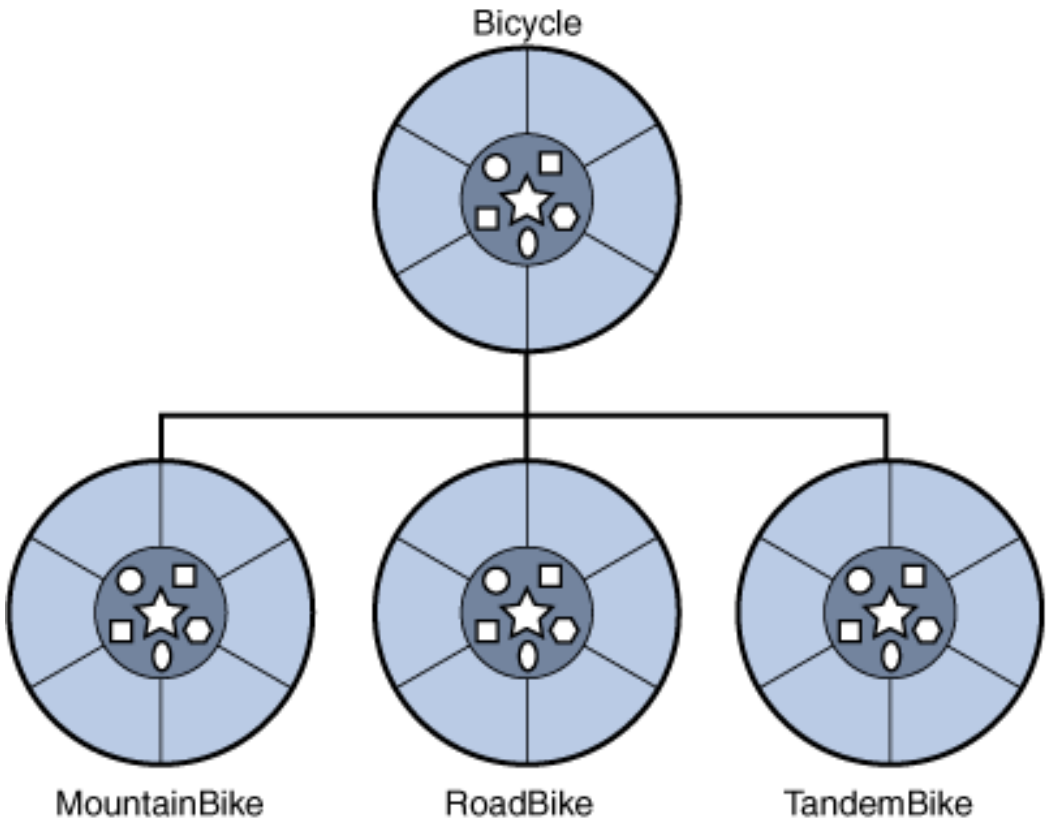
speed: 10 gear: 2

speed: 20 gear: 3

Inheriting

Different types of objects often have common elements. For example, a mountain bike, a city bike, a racing bike or a tandem. Each type has some similar elements: a current speed, a current gear. On the other hand, a tandem has two sits and two handlebars where a city bike has a basket and mountain bike often has more gears, etc.

The object-oriented programming allows to inherit the common elements. In the presented example, we can consider, the class *Bicylce* which is is the *superclass* for a class *MountainBike,* a *RacingBike* and a *TandemBike*. In C++ each class can have many *superclasses*–this is a bad practice–and each *superclass* can have many *subclasses*.



An example of the class inheriting.

Inheriting in C++

```
class MountainBike: public Bicycle {

     // new fields and methods defining a mountain bike would go here

}
```

In this example, the *MountainBike* has all the fields and all the methods of the class *Bicycle.* Nevertheless, we can add specific for mountain bikes elements to the class *MountainBike*. Thanks to that the superclass like the class *Bicycle* can be simple. But we need to remember about appropriate documentation of each subclass.

What is an interface? Pardon me an abstract class?

As you know already, objects define interactions with the outside world via methods and methods create an *interface* with the outside world like a remote controller creates an interface between you and a TV. An interface is a group of methods without their definitions. In C++ interface is realized by so-called abstract classes. Note that we cannot create objects derived from interfaces/abstract classes.

```
class Bicycle {

        void changeCadence(int newValue) = 0; // By setting zero to a
method we set a class to be a pure abstract.

        void changeGear(int newValue);

        void speedUp(int increment);

        void applyBrakes(int decrement);
}
```

Interface implementation

A class can implement an interface which means that this class has methods of the same declarations like an interface but defines them.

```
class ACMEBicycle : public Bicycle {

   // remainder of this class implemented as before
}
```

The main advantages of interfaces

Interfaces define interactions between classes and the outside world but the definition is realized in the sense of declarations and not definitions. In our example, the abstract class *Bicycle* defines methods which each bicycle should implements – define. Let's assume that *Bicycle* is realized by the class *ACMEBicycle* but at some point we have written another class which implements this interface better. Thanks to interfaces we can easily replace the class *ACMEBicycle* with a minimal modifications of a program.