

Polymorphism

Having learned the basics of inheritance, creating an inheritance hierarchy, and understanding that public inheritance essentially models an is-a relationship, it's time to move on to consuming this knowledge in learning the holy grail of object-oriented programming: polymorphism. Program statements code: This is the part of a program that performs actions and they are called methods.

In this lesson, you find out

- What polymorphism actually means
- What virtual functions do and how to use them
- What abstract base classes are and how to declare them
- What virtual inheritance means and where you need it

Basics of Polymorphism

“Poly” is Greek for many, and “morph” means form. Polymorphism is that feature of object-oriented languages that allows objects of different types to be treated similarly. This lesson focuses on polymorphic behavior that can be implemented in C++ via the inheritance hierarchy, also known as subtype polymorphism.

Need for Polymorphic Behavior

In Lesson, “Implementing Inheritance,” you found out how Tuna and Carp inherit public method Swim() from Fish as shown in example1 of Lesson “Implementing Inheritance”. It is, however, possible that both Tuna and Carp provide their own Tuna::Swim() and Carp::Swim() methods to make Tuna and Carp different swimmers. Yet, as each of them is also a Fish, if a user with an instance of Tuna uses the base class type to invoke Fish::Swim(), he ends up executing only the generic part Fish::Swim() and not Tuna::Swim(), even though that base class instance Fish is a part of a Tuna . This problem is demonstrated in Example 1.

Exp1: Invoking Methods Using an Instance of the Base Class Fish , That Belongs to a Tuna

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: public:
6: void Swim()
7: {
8:     cout << "Fish swims!" << endl;
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
9:  }
10: };
11:
12: class Tuna:public Fish
13: {
14:     public:
15:     // override Fish::Swim
16:     void Swim()
17:     {
18:         cout << "Tuna swims!" << endl;
19:     }
20: };
21:
22: void MakeFishSwim(Fish& InputFish)
23: {
24:     // calling Fish::Swim
25:     InputFish.Swim();
26: }
27:
28: int main()
29: {
30:     Tuna myDinner;
31:
32:     // calling Tuna::Swim
33:     myDinner.Swim();
34:
35:     // sending Tuna as Fish
36:     MakeFishSwim(myDinner);
37:
38:     return 0;
39: }
```

Output ▼

Tuna swims!

Fish swims!

Polymorphic Behavior Implemented Using Virtual Functions

You have access to an object of type Fish, via pointer Fish* or reference Fish&. This object could have been instantiated solely as a Fish, or be part of a Tuna or Carp that inherits from Fish. You don't know (and don't care). You invoke method Swim() using this pointer or reference, like this:

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
pFish->Swim();  
myFish.Swim();
```

What you expect is that the object Fish swims as a Tuna if it is part of a Tuna, as a Carp if it is part of a Carp, or an anonymous Fish if it wasn't instantiated as part of a specialized class such as Tuna or Carp. You can ensure this by declaring function Swim() in the base class Fish as a virtual function:

```
class Base  
{  
    virtual Return Type FunctionName (Parameter List);  
};  
class Derived  
{  
    Return Type FunctionName (Parameter List);  
};
```

Use of keyword *virtual* means that the compiler ensures that any overriding variant of the requested base class method is invoked. Thus, if Swim() is declared virtual, invoking *myFish.Swim()* (*myFish* being of type *Fish*&) results in *Tuna::Swim()* being executed as demonstrated by Example2.

Exp2: The Effect of Declaring Fish::Swim() as a virtual Method

```
0: #include <iostream>  
1: using namespace std;  
2:  
3: class Fish  
4: {  
5: public:  
6:     virtual void Swim()  
7:     {  
8:         cout << "Fish swims!" << endl;  
9:     }  
10: };  
11:  
12: class Tuna:public Fish  
13: {  
14:     public:  
15:         // override Fish::Swim  
16:         void Swim()  
17:         {  
18:             cout << "Tuna swims!" << endl;
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
19: }
20: };
21:
22: class Carp:public Fish
23: {
24:     public:
25:     // override Fish::Swim
26:     void Swim()
27:     {
28:         cout << "Carp swims!" << endl;
29:     }
30: };
31:
32: void MakeFishSwim(Fish& InputFish)
33: {
34:     // calling virtual method Swim()
35:     InputFish.Swim();
36: }
37:
38: int main()
39: {
40:     Tuna myDinner;
41:     Carp myLunch;
42:
43:     // sending Tuna as Fish
44:     MakeFishSwim(myDinner);
45:
46:     // sending Carp as Fish
47:     MakeFishSwim(myLunch);
48:
49:     return 0;
50: }
```

Output ▼

Tuna swims!
Carp swims!

Need for Virtual Destructors

There is a more sinister side to the feature demonstrated by Example1—unintentionally invoking base class functionality of an instance of type derived, when a specialization is available. What happens when a function calls operator *delete* using a pointer of type *Base** that actually points to an instance of type *Derived*?

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

Which destructor would be invoked? See Example 3.

Exp3: A Function That Invokes Operator delete on Base*

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5: public:
6:   Fish()
7:   {
8:     cout << "Constructed Fish" << endl;
9:   }
10:  ~Fish()
11:  {
12:    cout << "Destroyed Fish" << endl;
13:  }
14: };
15:
16: class Tuna:public Fish
17: {
18: public:
19:   Tuna()
20:   {
21:     cout << "Constructed Tuna" << endl;
22:   }
23:  ~Tuna()
24:  {
25:    cout << "Destroyed Tuna" << endl;
26:  }
27: };
28:
29: void DeleteFishMemory(Fish* pFish)
30: {
31:   delete pFish;
32: }
33:
34: int main()
35: {
36:   cout << "Allocating a Tuna on the free store:" << endl;
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
37: Tuna* pTuna = new Tuna;
38: cout << "Deleting the Tuna: " << endl;
39: DeleteFishMemory(pTuna);
40:
41: cout << "Instantiating a Tuna on the stack:" << endl;
42: Tuna myDinner;
43: cout << "Automatic destruction as it goes out of scope: " << endl;
44:
45: return 0;
46: }
```

Output ▼

```
Allocating a Tuna on the free store:
Constructed Fish
Constructed Tuna
Deleting the Tuna:
Destroyed Fish
Instantiating a Tuna on the stack:
Constructed Fish
Constructed Tuna
Automatic destruction as it goes out of scope:
Destroyed Tuna
Destroyed Fish
```

Note that while *Tuna* and *Fish* were both constructed on the free store due to *new*, the destructor of *Tuna* was not invoked on *delete*, rather only that of the *Fish*. This is in stark contrast to the construction and destruction of local member *myDinner* where all constructors and destructors are invoked. In the Lesson "Inheritance" we have demonstrated the correct order of construction and destruction of classes in an inheritance hierarchy, showing that all destructors need to be invoked, including *~Tuna()*. Clearly, something is amiss.

This flaw means that code in the destructor of a deriving class that has been instantiated on the free store using *new* would not be invoked if *delete* is called using a pointer of type *Base**. This can result in resources not being released, memory leaks, and so on and is a problem that is not to be taken lightly.

To avoid this problem, you use virtual destructors as seen in Example 4.

Exp4: Using virtual Destructors to Ensure That Destructors in Derived Classes Are Invoked When Deleting a Pointer of Type *Base**

```
0: #include <iostream>
1: using namespace std;
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
2:
3: class Fish
4: {
5:     public:
6:         Fish()
7:         {
8:             cout << "Constructed Fish" << endl;
9:         }
10:    virtual ~Fish() // virtual destructor!
11:    {
12:        cout << "Destroyed Fish" << endl;
13:    }
14: };
15:
16: class Tuna:public Fish
17: {
18:     public:
19:         Tuna()
20:         {
21:             cout << "Constructed Tuna" << endl;
22:         }
23:         ~Tuna()
24:         {
25:             cout << "Destroyed Tuna" << endl;
26:         }
27: };
28:
29: void DeleteFishMemory(Fish* pFish)
30: {
31:     delete pFish;
32: }
33:
34: int main()
35: {
36:     cout << "Allocating a Tuna on the free store:" << endl;
37:     Tuna* pTuna = new Tuna;
38:     cout << "Deleting the Tuna: " << endl;
39:     DeleteFishMemory(pTuna);
40:
41:     cout << "Instantiating a Tuna on the stack:" << endl;
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
42:   Tuna myDinner;
43:   cout << "Automatic destruction as it goes out of scope: " << endl;
44:
45:   return 0;
46: }
```

Output ▼

```
Allocating a Tuna on the free store:
Constructed Fish
Constructed Tuna
Deleting the Tuna:
Destroyed Tuna
Destroyed Fish
Instantiating a Tuna on the stack:
Constructed Fish
Constructed Tuna
Automatic destruction as it goes out of scope:
Destroyed Tuna
Destroyed Fish
```

NOTE: Always declare the base class destructor as *virtual*:

```
class Base
{
public:
    virtual ~Base() {}; // virtual destructor
};
```

This ensures that one with a pointer `Base*` cannot invoke *delete* in a way that the destructor of the deriving classes is not invoked.

Abstract Base Classes and Pure Virtual Functions

A base class that cannot be instantiated is called an abstract base class. Such a base class fulfills only one purpose, that of being derived from. C++ allows you to create an abstract base class using pure virtual functions.

A virtual method is said to be pure virtual when it has a declaration as shown in the following:

```
class AbstractBase
{
public:
    virtual void DoSomething() = 0; // pure virtual method
};
```


OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

This declaration essentially tells the compiler that *DoSomething()* needs to be implemented and by the class that derives from *AbstractBase*:

```
class Derived: public AbstractBase
{
    public:
    void DoSomething() // pure virtual method
    {
        cout << "Implemented virtual function" << endl;
    }
};
```

Thus, what class *AbstractBase* has done is that it has enforced class *Derived* to supply an implementation for virtual method *DoSomething()*. This functionality where a base class can enforce support of methods with a specified name and signature in classes that derive from it is that of an interface. Think of a *Fish* again. Imagine a *Tuna* that cannot swim fast because *Tuna* did not override *Fish::Swim()*. This is a failed implementation and a flaw. Making class *Fish* an abstract base class with *Swim* as a pure virtual function ensures that *Tuna* that derives from *Fish* implements *Tuna::Swim()* and swims like a *Tuna* and not like just any *Fish*. See Example 5.

Exp5: class Fish as an Abstract Base Class for Tuna and Carp

```
0: #include <iostream>
1: using namespace std;
2:
3: class Fish
4: {
5:     public:
6:     // define a pure virtual function Swim
7:     virtual void Swim() = 0;
8: };
9:
10: class Tuna:public Fish
11: {
12:     public:
13:     void Swim()
14:     {
15:         cout << "Tuna swims fast in the sea!" << endl;
16:     }
17: };
18:
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
19: class Carp:public Fish
20: {
21:     void Swim()
22:     {
23:         cout << "Carp swims slow in the lake!" << endl;
24:     }
25: };
26:
27: void MakeFishSwim(Fish& inputFish)
28: {
29:     inputFish.Swim();
30: }
31:
32: int main()
33: {
34:     // Fish myFish; // Fails, cannot instantiate an ABC
35:     Carp myLunch;
36:     Tuna myDinner;
37:
38:     MakeFishSwim(myLunch);
39:     MakeFishSwim(myDinner);
40:
41:     return 0;
42: }
```

Output ▼

Carp swims slow in the lake!
Tuna swims fast in the sea!

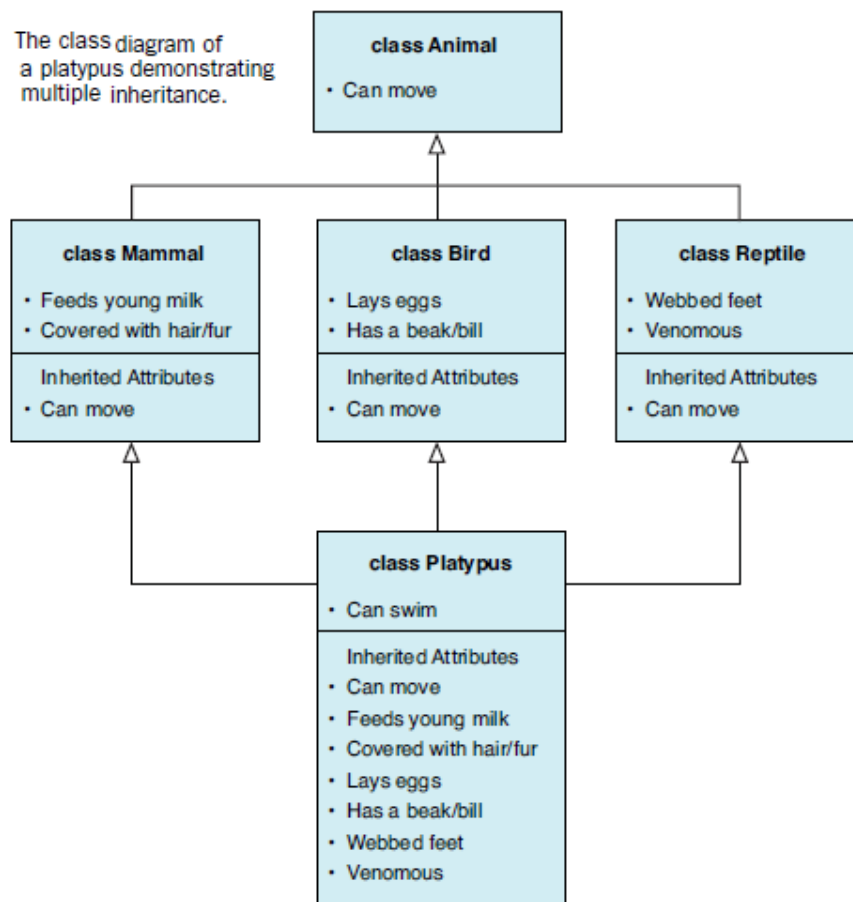
NOTE: Abstract Base Classes are often simply called ABCs. ABCs help enforce certain design constraints on your program.

Using virtual Inheritance to Solve the Diamond Problem

In Lesson "Inheritance" we saw the curious case of a duck-billed platypus that is part mammal, part bird, and part reptile. This is an example where a class Platypus needs to inherit from class Mammal, class Bird, and class Reptile. However, each of these in turn inherit from a more generic class Animal, as illustrated by the following Figure.

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN



So, what happens when you instantiate a Platypus? How many instances of class Animal are instantiated for one instance of Platypus? Example6 helps answer this question.

Exp6: Checking for the Number of Base Class Animal Instances for One Instance of Platypus

```
0: #include <iostream>
1: using namespace std;
2:
3: class Animal
4: {
5: public:
6:     Animal()
7:     {
8:         cout << "Animal constructor" << endl;
9:     }
10:
11: // sample method
12: int Age;
13: };
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
14:
15: class Mammal:public Animal
16: {
17: };
18:
19: class Bird:public Animal
20: {
21: };
22:
23: class Reptile:public Animal
24: {
25: };
26:
27: class Platypus:public Mammal, public Bird, public Reptile
28: {
29:     public:
30:     Platypus()
31:     {
32:         cout << "Platypus constructor" << endl;
33:     }
34: };
35:
36: int main()
37: {
38:     Platypus duckBilledP;
39:
40:     // uncomment next line to see compile failure
41:     // Age is ambiguous as there are three instances of base Animal
42:     // duckBilledP.Age = 25;
43:
44:     return 0;
45: }
```

Output ▼

```
Animal constructor
Animal constructor
Animal constructor
Platypus constructor
```

As the output demonstrates, due to multiple inheritance and all three base classes of Platypus inheriting in turn from class Animal, you have three instances of Animal created automatically for every instance of a Platypus, as shown in Line 38. This is

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

quite ridiculous as Platypus is still one animal that has inherited certain attributes from Mammal, Bird, and Reptile. The problem in the number of instances of base Animal is not limited to memory consumption alone. Animal has an integer member Animal::Age (which has been kept public for explanation purposes). When you want to access Animal::Age via an instance of Platypus, as shown in Line 42, you get a compilation error simply because the compiler doesn't know if you want to set

Mammal::Animal::Age or Bird::Animal::Age or Reptile::Animal::Age . It can get even more ridiculous—if you so wanted you could set all three:

```
duckBilledP.Mammal::Animal::Age = 25;
```

```
duckBilledP.Bird::Animal::Age = 25;
```

```
duckBilledP.Reptile::Animal::Age = 25;
```

Clearly, one duck-billed platypus should have only one Age. Yet, you want class Platypus to be a Mammal, Bird, and Reptile. The solution is in virtual inheritance. If you expect a derived class to be used as a base class, it possibly is a good idea to define its relationship to the base using the keyword virtual:

```
class Derived1: public virtual Base
```

```
{
```

```
    // ... members and functions
```

```
};
```

```
class Derived2: public virtual Base
```

```
{
```

```
    // ... members and functions
```

```
};
```

A better class Platypus (actually a better class Mammal, class Bird, and class Reptile) is in the Example 7.

| |
|--|
| Exp7: Demonstrating How virtual Keyword in Inheritance Hierarchy Helps Restrict the Number of Instances of Base Class Animal to One |
|--|

| |
|---|
| <pre>0: #include <iostream> 1: using namespace std; 2: 3: class Animal 4: { 5: public: 6: Animal() 7: { 8: cout << "Animal constructor" << endl; 9: } 10:</pre> |
|---|

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
11: // sample method
12: int Age;
13: };
14:
15: class Mammal:public virtual Animal
16: {
17: };
18:
19: class Bird:public virtual Animal
20: {
21: };
22:
23: class Reptile:public virtual Animal
24: {
25: };
26:
27: class Platypus:public Mammal, public Bird, public Reptile
28: {
29: public:
30: Platypus()
31: {
32:     cout << "Platypus constructor" << endl;
33: }
34: };
35:
36: int main()
37: {
38:     Platypus duckBilledP;
39:
40:     // no compile error as there is only one Animal::Age
41:     duckBilledP.Age = 25;
42:
43:     return 0;
44: }
```

Output ▼

Animal constructor

Platypus constructor

Do a quick comparison against the output of previous Example 7, and you see that the number of instances of class Animal constructed has fallen to one, which is finally reflective of the fact that only one Platypus has been constructed as well. This is

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

because of the keyword `virtual` used in the relationship between classes `Mammal`, `Bird`, and `Reptile` ensures that when these classes are grouped together under `Platypus` the common base `Animal` exists only in a single instance. This solves a lot of problems; one among them is Line 41 that now compiles without ambiguity resolution as shown in Example 6.

NOTE: Problems caused in an inheritance hierarchy containing two or more base classes that inherit from a common base, which results in the need for ambiguity resolution in the absence of virtual inheritance, is called the Diamond Problem.

The name “Diamond” is possibly inspired by the shape the class diagram takes (visualize above Figure with straight and slanted lines relating `Platypus` to `Animal` via `Mammal`, `Bird`, and `Reptile` to see a diamond).

Virtual Copy Constructors?

Well, the question mark at the end of the section title is justified. It is technically impossible in C++ to have virtual copy constructors. Yet, such a feature helps you create a collection (for example, a static array) of type `Base*`, each element being a specialization of that type:

```
// Tuna, Carp and Trout are classes that inherit public from base class Fish
```

```
Fish* pFishes[3];
```

```
Fishes[0] = new Tuna();
```

```
Fishes[1] = new Carp();
```

```
Fishes[2] = new Trout();
```

Then assigning it into another array of the same type, where the virtual copy constructor ensures a deep copy of the derived class objects as well, ensures that `Tuna`, `Carp`, and `Trout` are copied as `Tuna`, `Carp`, and `Trout` even though the copy constructor is operating on type `Fish*`.

Well, that’s a nice dream.

Virtual copy constructors are not possible because the `virtual` keyword in context of base class methods being overridden by implementations available in the derived class are about polymorphic behavior generated at runtime. Constructors, on the other hand, are not polymorphic in nature as they can construct only a fixed type, and hence C++ does not allow usage of the virtual copy constructors.

Having said that, there is a nice workaround in the form of defining your own clone function that allows you to do just that:

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
class Fish
{
    public:
        virtual Fish* Clone() const = 0; // pure virtual function
};

class Tuna:public Fish
{
    // ... other members
    public:
        Tuna * Clone() const // virtual clone function
    {
        return new Tuna(*this); // return new Tuna that is a copy of this
    }
};
```

Thus, virtual function Clone is a simulated virtual copy constructor that needs to be explicitly invoked, as shown in Example 8.

| Exp8: Tuna and Carp That Support a Clone Function as a Simulated Virtual Copy Constructor |
|--|
|--|

| |
|--|
| <pre>0: #include <iostream> 1: using namespace std; 2: 3: class Fish 4: { 5: public: 6: virtual Fish* Clone() = 0; 7: virtual void Swim() = 0; 8: }; 9: 10: class Tuna: public Fish 11: { 12: public: 13: Fish* Clone() 14: { 15: return new Tuna (*this); 16: } 17: 18: void Swim() 19: { 20: cout << "Tuna swims fast in the sea" << endl; 21: }</pre> |
|--|

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

```
22: };
23:
24: class Carp: public Fish
25: {
26:     Fish* Clone()
27:     {
28:         return new Carp(*this);
29:     }
30:     void Swim()
31:     {
32:         cout << "Carp swims slow in the lake" << endl;
33:     }
34: };
35:
36: int main()
37: {
38:     const int ARRAY_SIZE = 4;
39:
40:     Fish* myFishes[ARRAY_SIZE] = {NULL};
41:     myFishes[0] = new Tuna();
42:     myFishes[1] = new Carp();
43:     myFishes[2] = new Tuna();
44:     myFishes[3] = new Carp();
45:
46:     Fish* myNewFishes[ARRAY_SIZE];
47:     for (int Index = 0; Index < ARRAY_SIZE; ++Index)
48:         myNewFishes[Index] = myFishes[Index]->Clone();
49:
50:     // invoke a virtual method to check
51:     for (int Index = 0; Index < ARRAY_SIZE; ++Index)
52:         myNewFishes[Index]->Swim();
53:
54:     // memory cleanup
55:     for (int Index = 0; Index < ARRAY_SIZE; ++Index)
56:     {
57:         delete myFishes[Index];
58:         delete myNewFishes[Index];
59:     }
60:     return 0;
61: }
```

OBJECT ORIENTED PROGRAMMING

DR. AHMED A. HUSSEIN/COLLEGE OF SCIENCE FOR WOMEN

Output ▼

Tuna swims fast in the sea

Carp swims slow in the lake

Tuna swims fast in the sea

Carp swims slow in the lake