Abstraction in Header Files

The second strategy for implementing abstraction is creating header files. Find out more below!

WE'LL COVER THE FOLLOWING

In the last lesson, we created the `Circle` class which had two functions, `area()` and `perimeter()`. At that point, all the code was in a single file. Since our goal is to hide the unnecessary details from the users, we can divide the code into different files. This is where **header files** come into play.

Creating Header Files

Let's take a look at the `Circle` class we created in the previous lesson:

```
#include <iostream>
using namespace std;

class Circle{
  double radius;
  double pi;

  public:
  Circle (){
    radius = 0;
    pi = 3.142;
  }
  Circle(double r){
    radius = r;
    pi = 3.142;
  }

  double area(){
    return pi * radius * radius;
  }

  double perimeter(){
    return 2 * pi * radius;
  }
};

int main() {
```

```
  Circle c(5);
  cout << "Area: " << c.area() << endl;

  cout << "Perimeter: " << c.perimeter() << endl;
}
```

To hide our class, we will follow a few steps. The first step is to create a header file. This file will only contain the declaration of the class and its members. A header file always has the `.h` extension:

main.cpp

Circle.h

```
#include <iostream>
using namespace std;

class Circle{
  double radius;
  double pi;

  public:
  Circle (){
    radius = 0;
    pi = 3.142;
  }
  Circle(double r){
    radius = r;
    pi = 3.142;
  }

  double area(){
    return pi * radius * radius;
  }

  double perimeter(){
    return 2 * pi * radius;
  }
};

int main() {
  Circle c(5);
  cout << "Area: " << c.area() << endl;
  cout << "Perimeter: " << c.perimeter() << endl;
}
```

As you can see, the header file isn't very useful if the complete implementation is still visible in our `main` file. Therefore, the second step is to move all the implementation to a separate file. Let's call this `Circle.cpp`.

In this file, we must **include the header file**. The `include` command should already be familiar to you. We use it all the time to include libraries like

`iostream` or `vector`. We can include header files in the same way!

Since we're implementing all the methods of our `Circle` class in `Circle.cpp`, we must mention the name of the class along with the *scope resolution operator* (`::`). Let's do this now:

main.cpp

Circle.cpp

Circle.h

```
#include <iostream>
using namespace std;

class Circle{
  double radius;
  double pi;

  public:
  Circle (){
    radius = 0;
    pi = 3.142;
  }
  Circle(double r){
    radius = r;
    pi = 3.142;
  }

  double area(){
    return pi * radius * radius;
  }

  double perimeter(){
    return 2 * pi * radius;
  }
};

int main() {
  Circle c(5);
  cout << "Area: " << c.area() << endl;
  cout << "Perimeter: " << c.perimeter() << endl;
}
```

At this point, everything is in place. We can remove the `Circle` class from our `main.cpp`. All we have to do is include the header file and the compiler will handle the rest:

main.cpp

Circle.h

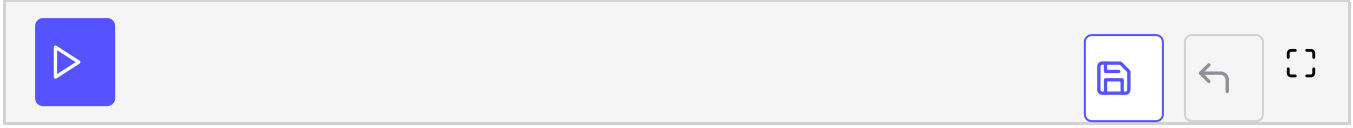
Circle.cpp

```
#include <iostream>
#include "./Circle.h"

using namespace std;


int main() {
  Circle c(5);
  cout << "Area: " << c.area() << endl;
  cout << "Perimeter: " << c.perimeter() << endl;
```

```
}
```



In the header file, we have two commands:

```
#ifndef CIRCLE_H
#define CIRCLE_H
```

These commands tell the compiler that this header file can be used in multiple places. The `#ifndef` command ends with `#endif`.

What we're seeing now is complete abstraction. None of the implementation is visible to users. If they need to know what methods are available in the `Circle` class, they can simply refer to the header file.

This ends our discussion on data hiding. Combining encapsulation and abstraction gives us a simple and reusable interface for our program. In the next section, we'll explore the concept of inheritance.