

# Introduction to OOP

Object Oriented Programming

# Introduction to OOP

- **What is OOP?**

- Object-oriented methodology is a way of viewing software components and their relationships.
- Object-oriented methodology relies on three characteristics that define object-oriented languages:
  1. Encapsulation
  2. Inheritance
  3. Polymorphism

# Introduction to OOP - Class

- **What is a Class?**

- A class is a blueprint of objects.
- A class is an object factory.
- A class is the template to create the object.
- A class is a user defined datatype

- **Object:**

- An object is an instance of a class.
- The property values of an object instance is different from the ones of other object instances of a same class
- Object instances of the same class share the same behavior (methods).

# Introduction to OOP – Object & Class

- *Class* reflects concepts.
- *Object* reflects instances that embody those concepts.

*class*



*object*



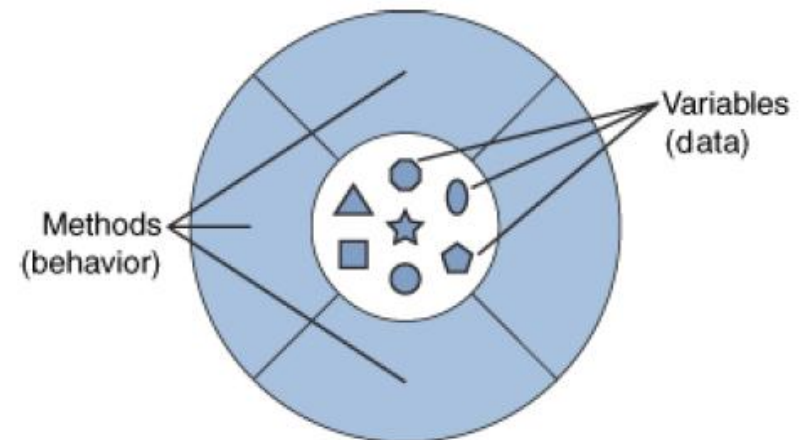
# Introduction to OOP – Object

- **What is an Object?**

- An object is a software bundle of variables and related methods.

- Object consists of:

- Data (object's Attributes)
  - Behavior (object's methods)



# Introduction to OOP – Object & Class

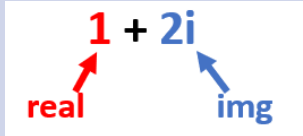
- Assume we have this object [`complexNumberObj`]
- How can we describe its Class?
  - First, it's a complex number.  
Let's give it that name `Complex`.
  - Second each complex number has a `real` part, and `imaginary` part.
  - Last thing it can `print` its values

`complexNumberObj`

`1 + 2i`  
↑      ↑  
`real` `img`

# Introduction to OOP – Object & Class

- So, we have two different concepts here are very important.

Class	Object
<p><u>Encapsulates</u> the <b>attributes</b>, and <b>behaviors</b> into a blue-print code to provide the design concept.</p>	<p>The living thing that interacts and actually runs.</p> 
<pre>Ex: class Complex {     real     img     print() }</pre>	<pre>Complex c1 = new Complex ( ); real = 1 img = 2</pre>

# How to create a class?

- To define a class, we write:

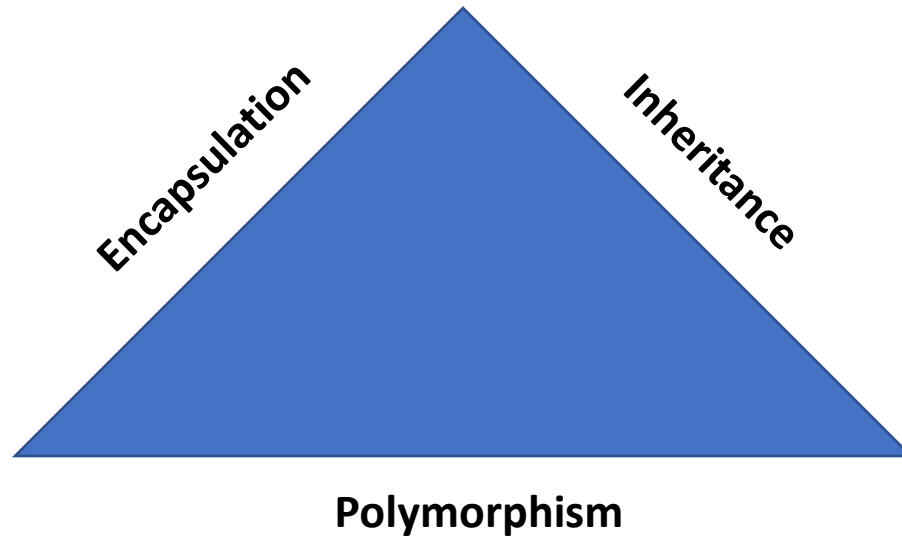
```
<access-modifier>* class <name>
{
    <attributeDeclaration>*
    <methodDeclaration>*
    <constructorDeclaration>*
}
```

- Example:

```
public class Complex {
    private int real;
    private int img;
    public void print() {
        System.out.println(real + " + " + img + "i");
    }
}
```

# OOP Principles

- Object Oriented has three principles:



# Encapsulation

- It is to **encapsulate the data and behaviors in one class**.
- Accompanied with **Data hiding** concept that leads to restricting access to some of the object's components.
- Data hiding is done in Java using the **public**, and **private** key words.

```
public class Complex {  
    private int real;  
    private int img;  
    public void print() {  
        System.out.println(real + " + " + img + "i");  
    }  
}
```

← private class members can only be accessed **from only inside the class**

↑ Public class members can be accessed from **inside or outside the class**

# Encapsulation cont'd

```
public class Complex {  
    private int real;  
    private int img;  
    public void print() {  
        System.out.println(real + " + " + img + "i");  
    }  
}
```

←← Accessible from inside the class

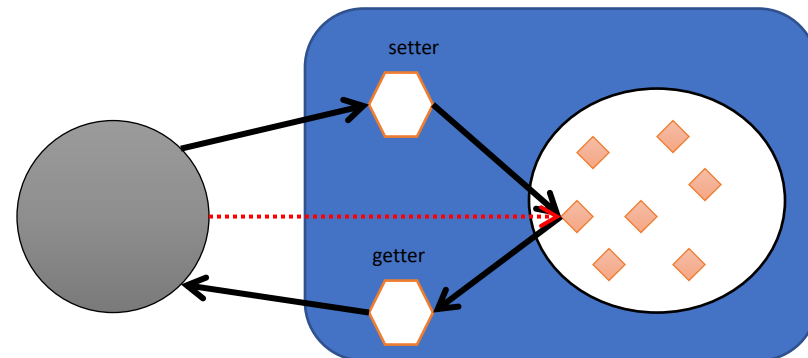
```
class Main{  
    public static void main(String[] args){  
        Complex c1 = new Complex();  
        c1.real = 1;  
        c1.img = 2;  
        c1.print();  
    }  
}
```

real has private access in Complex  
----  
(Alt-Enter shows hints)

←← Cannot be accessed outside the class

# Encapsulation cont'd

- As we can see in the previous example **private** class members are not accessible outside the class.
- These keywords are called **access modifiers**, and they can be added to attributes and methods.
- The point of **encapsulation** is to encapsulate data and behaviors in one class and to prevent the direct access to the data from the outside world and allow it through public accessor methods **protecting the data** from the misuse



# Encapsulation cont'd

- Exposing object attributes might lead to misuse
- Instead of exposing object attributes  
OOP controls the accessibility through the **setter**

```
public int getReal() {  
    return real;  
}
```

```
public void setReal(int r) {  
    real = r;  
}
```

```
public int getImg() {  
    return img;  
}
```

```
public void setImg(int i) {  
    img = i;  
}
```

# Encapsulation cont'd

- Now we can access the object attributes through the setter and getter methods like the following:

```
class Main{  
    public static void main(String[] args){  
        Complex c1 = new Complex();  
        c1.setReal(1);  
        c1.setImg(2);  
        c1.print();  
    }  
}
```

# Coding Guidelines

- Think of an appropriate name for your class.
  - Don't use XYZ or any random names.
- Class names starts with a CAPITAL letter.
  - not a requirement it is a convention

# Declaring Methods

- To declare a method:

```
<modifier>* <Return type> <name> ([<Param Type> <Param Name>]*)  
{  
    <Statement>*  
}
```

- Example:

```
class Complex {  
    private int real;  
    public int getReal(){ return real; }  
    public void setReal(int r){ real = r; }  
}
```

# Method characteristics:

- It can return one or no values
- It may accept as many parameters as needed or no parameters at all.
- After the method finishes execution, it goes back to the caller.
- Method name should start with a small letter.
- Method name should be a verb.

# Class and constructor

- To make use of this class we need to **construct** an object of this class.

```
Complex c1 = new Complex ( );
```

- Constructing an object is like building it and making it ready for action inside the program.
- To construct an object, we use a special type method called **Constructor**
- A **constructor** is simply a method that does not have a return type and its name matches the class name (case sensitive).

# Class and constructor cont'd

- The Constructor is a special type method that:
  1. Has the same name of the class case sensitive.
  2. Doesn't have a return type
  3. It is called automatically when creating an object to initialize the instance members of the class.

# Class and constructor cont'd

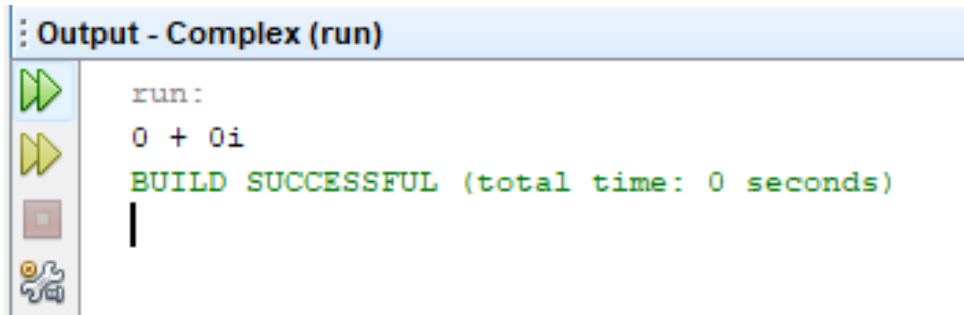
- If no constructor has been defined for the class, the environment provides us with one constructor that is called **default constructor**.
- The default constructor initializes the object members to the natural initial value, and it must be called before creating an object.
- If we wrote a custom constructor that takes arguments, then there will **NOT** be a default constructor.

# Class and constructor cont'd

- To construct an object of class we use the **new** keyword.

```
class Main{  
    public static void main(String[] args){  
        Complex c1 = new Complex();    The complex is created  
        c1.print();    Printing the complex values  
    }  
}
```

- The output of this program will be like



```
Output - Complex (run)  
run:  
0 + 0i  
BUILD SUCCESSFUL (total time: 0 seconds)  
|
```

# Class and constructor cont'd

- To add a constructor to our class we need to code extra method like - with the same name of the class (case sensitive).

```
public class Complex {  
    private int real;  
    private int img;  
  
    public Complex(int r, int i){  
        real = r;  
        img = i;  
    }  
    public void print(){  
        System.out.println(real + " + " + img + "i");  
    }  
}
```

# Class and constructor cont'd

- Now to construct an object we need to use the new constructor.

```
class Main{  
    public static void main(String[] args){  
        //Complex c1 = new Complex();  
        Complex c1 = new Complex(1,2);  
        c1.print();  
    }  
}
```

- Now using the default constructor is illegal because we have a custom constructor and the environment hasn't gifted us a default constructor.

# Members Categories

- Instance Members (Object Members)
- Static Members (Class Members)

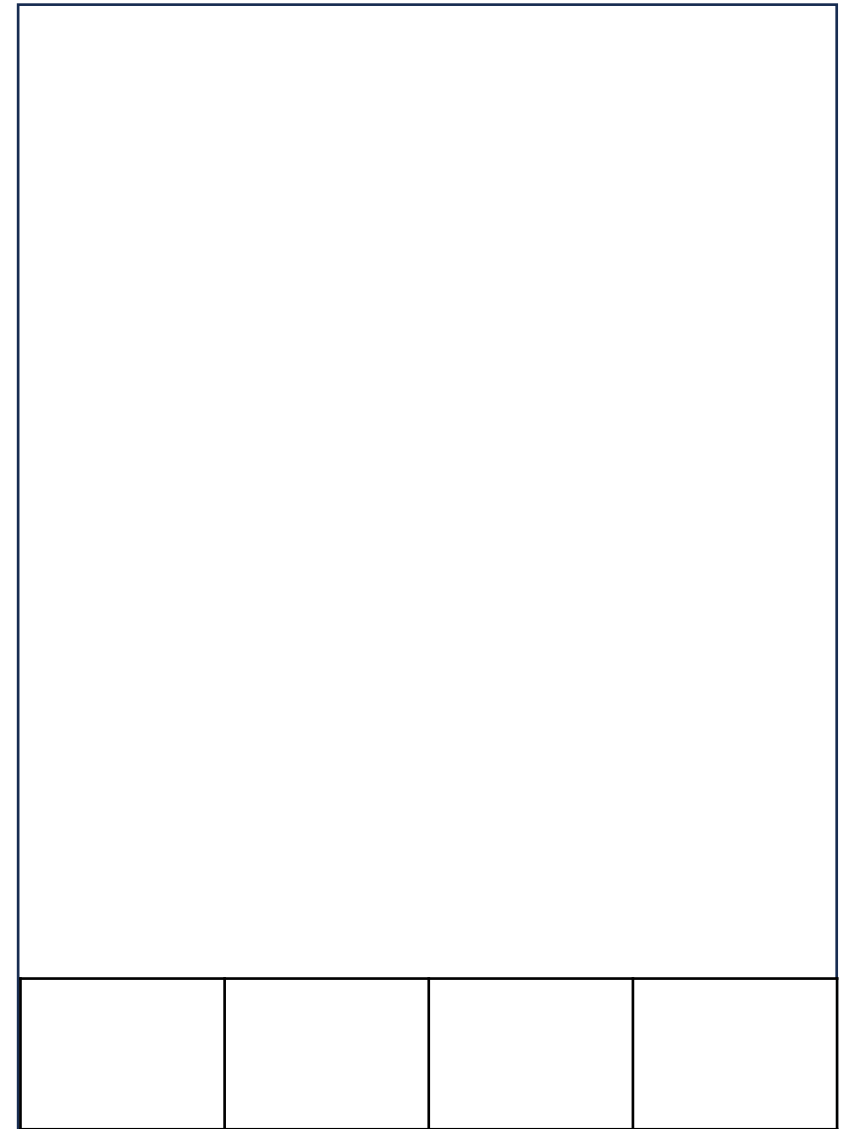
# Instance Members (Object Members)

- **Instance Member Characteristics:**

1. Describes the object
2. Loaded in the memory when an object created
3. Exists in the memory as many objects have been created (if they are variables) (but once if they are methods)
4. Can only be accessed via an object

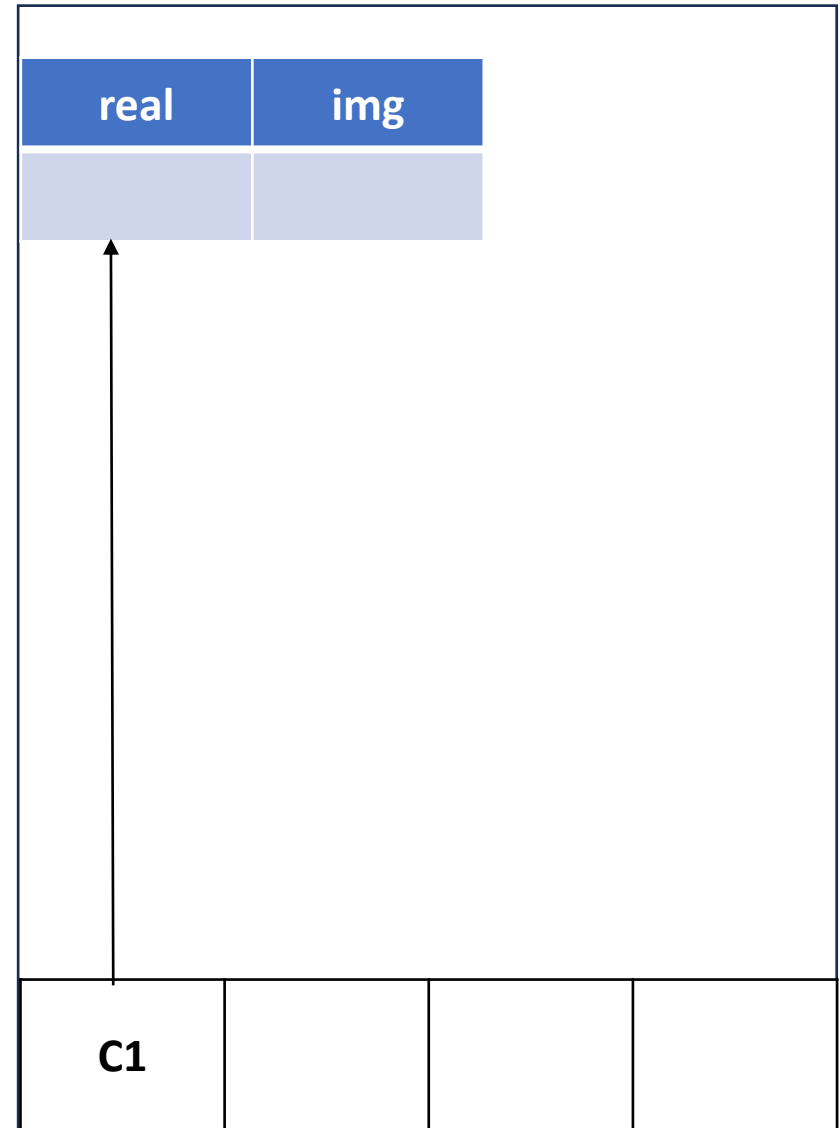
# Instance Members Example

- `Complex c1 = new Complex( )`



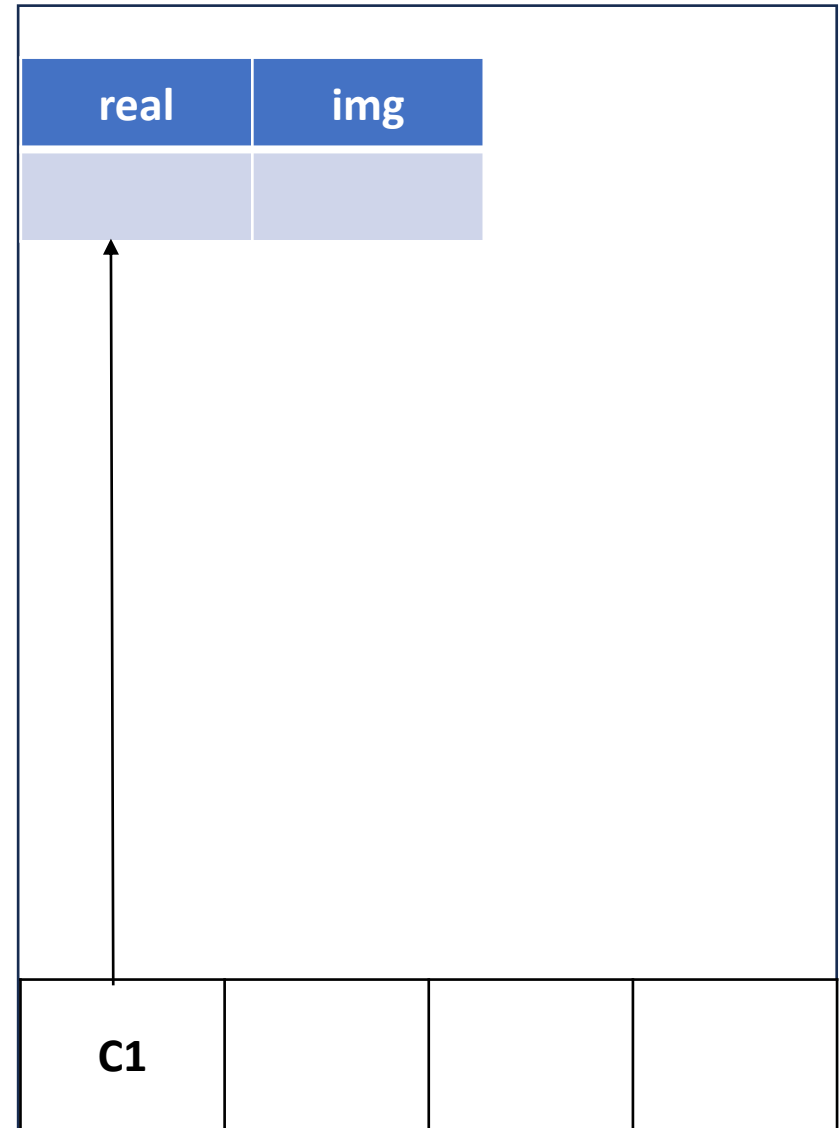
# Instance Members Example

- `Complex c1 = new Complex( )`



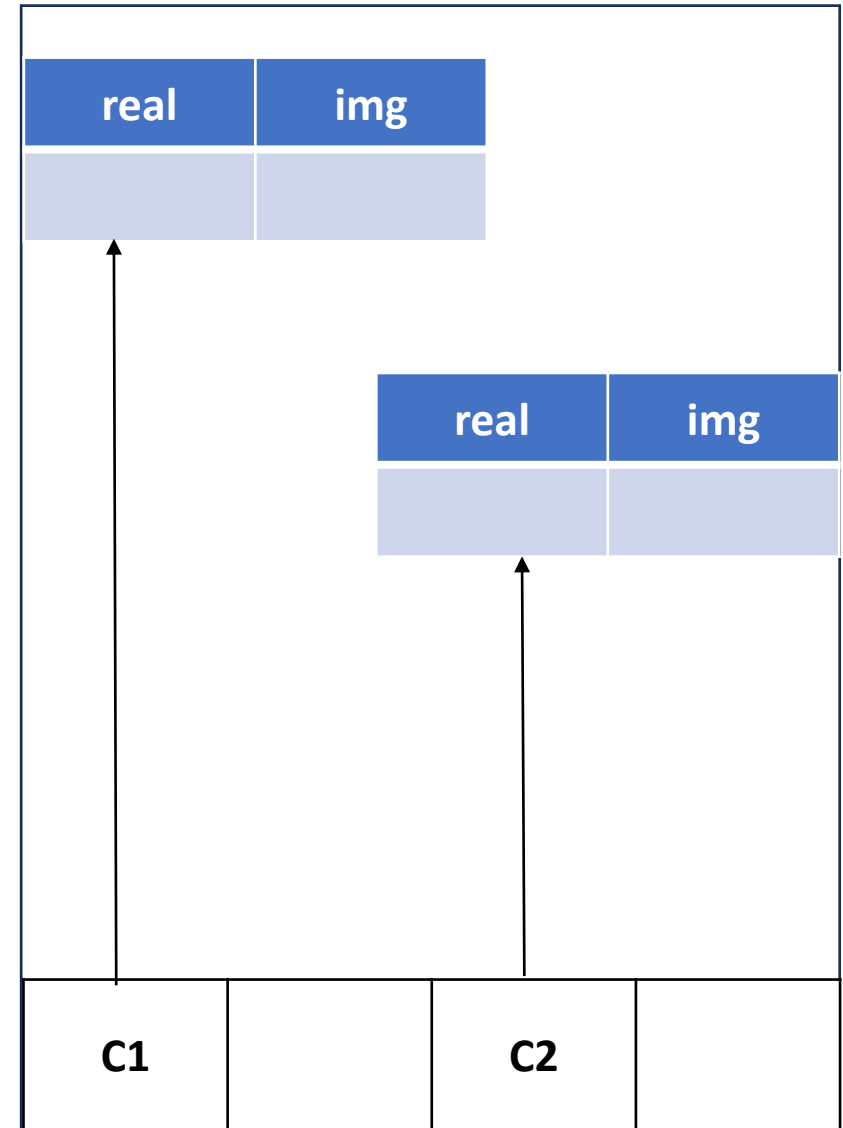
# Instance Members Example

- Complex c1 = new Complex( )
- Complex c2 = new Complex( )



# Instance Members Example

- Complex c1 = new Complex( )
- Complex c2 = new Complex( )



# Instance Members Example

- Complex c1 = new Complex( )

- Complex c2 = new Complex( )

- c1.setReal(1);

```
public void setReal(int r){  
    real = r;  
}
```

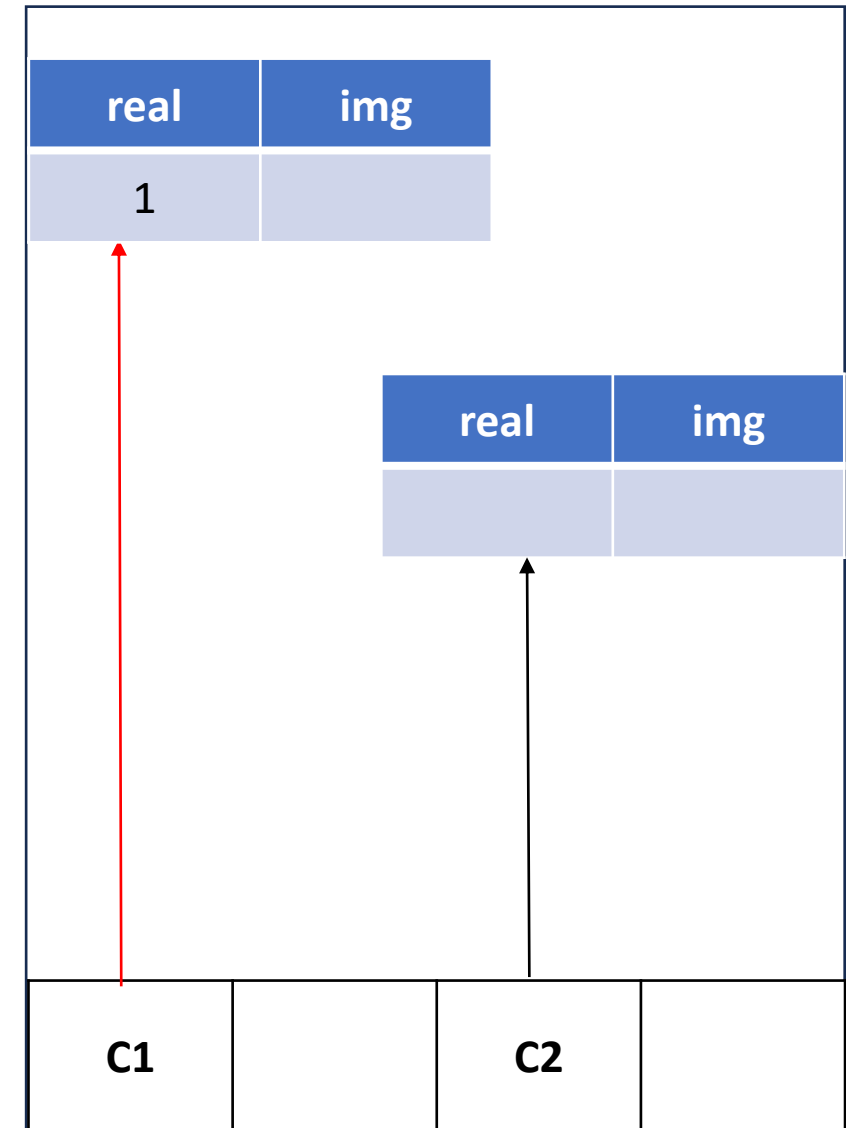
- c1.setReal(**c1**, 1);

**c1**      1

```
public void setReal(Complex this, int r){
```

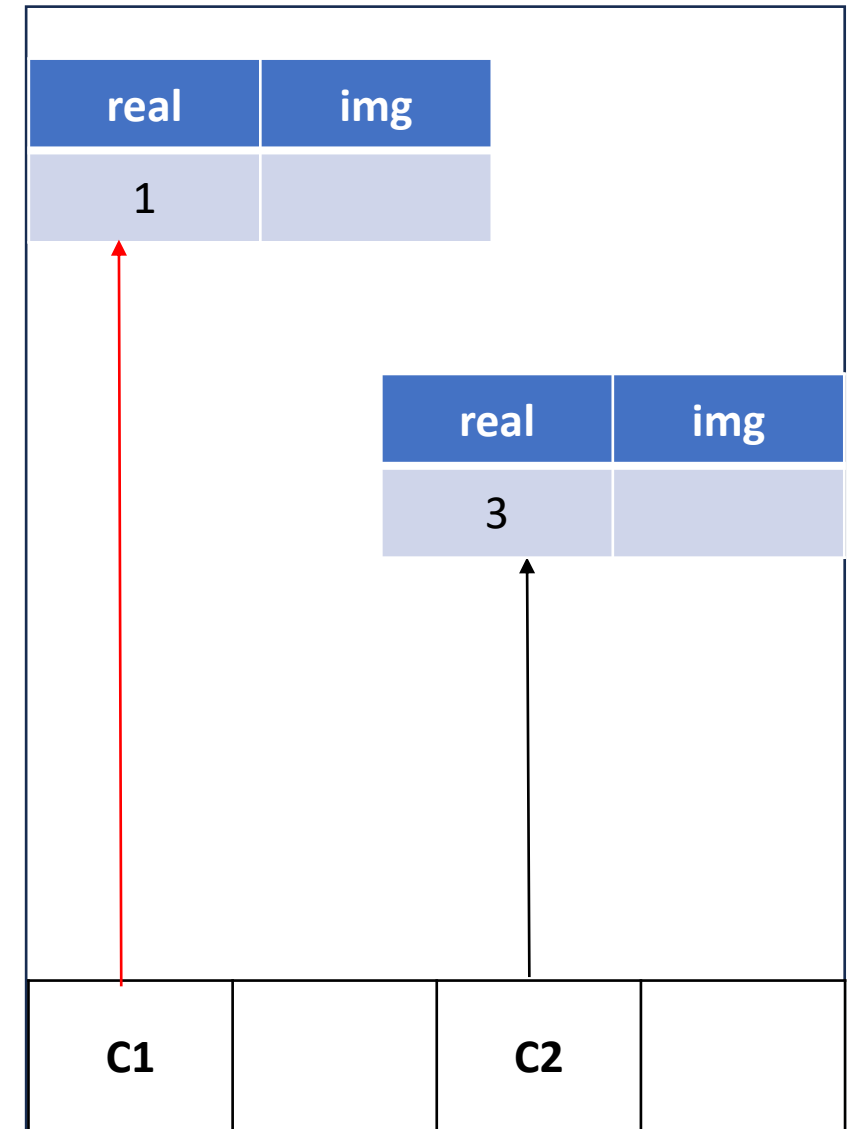
```
    this.real = r;
```

```
}
```



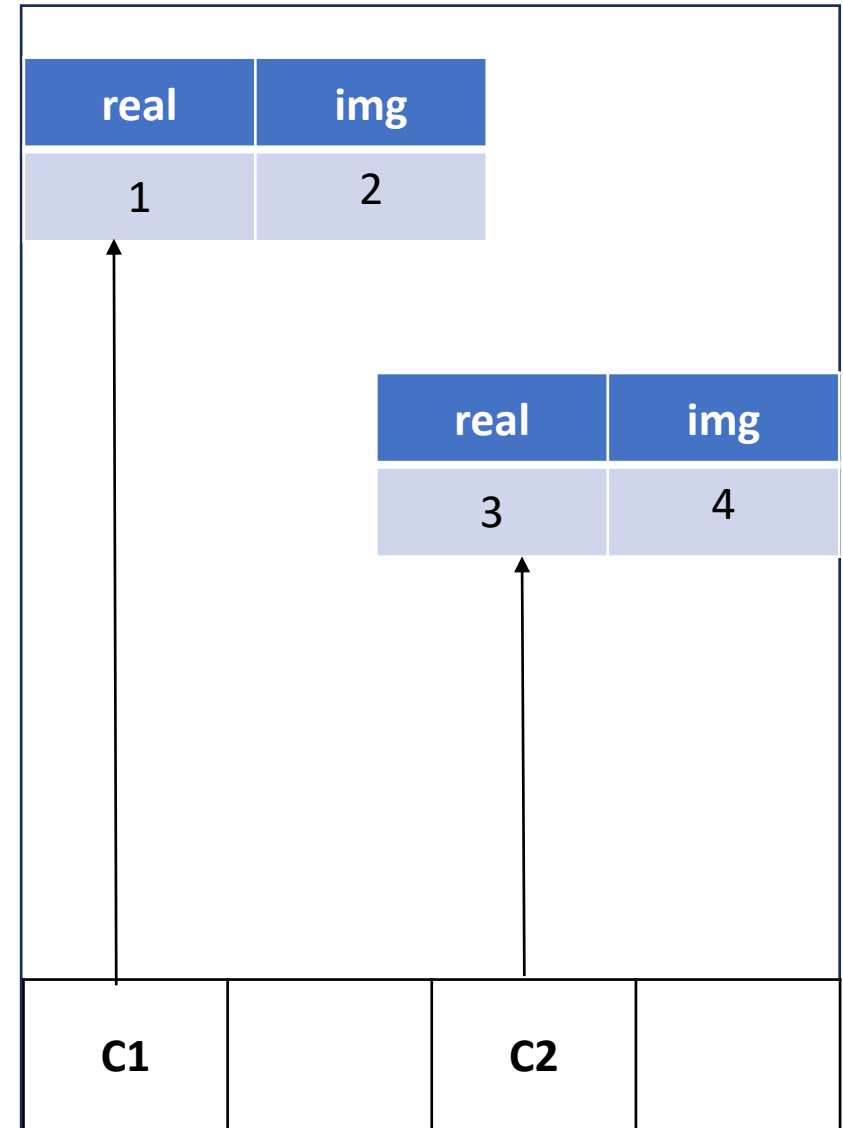
# Instance Members Example

- Complex c1 = new Complex( )
  - Complex c2 = new Complex( )
  - c1.setReal(1);
  - c2.setReal(3);
- 
- c2.setReal(**c2**, 3);  
public void setReal(Complex **this**, int r){  
    **this**.real = r;  
}



# Instance Members Example

- `Complex c1 = new Complex( )`
- `Complex c2 = new Complex( )`
- `c1.setReal(1);`
- `c1.setImg(2);`
- `c2.setReal(3);`
- `c2.setImg(4);`



# this Reference

- **this** is an implicit reference to the current instance of the Class
- Any instance member method has an implicit this reference that is passed to it as an argument.
- It refers to the caller so, the instance member method can access the right instance member variable

# Instance and Static Members cont'd

- There are some cases we must use **this** reference, like:

```
public Complex(int real, int img) {  
    this.real = real;  
    this.img = img;  
}
```

# Static Members (Class Members)

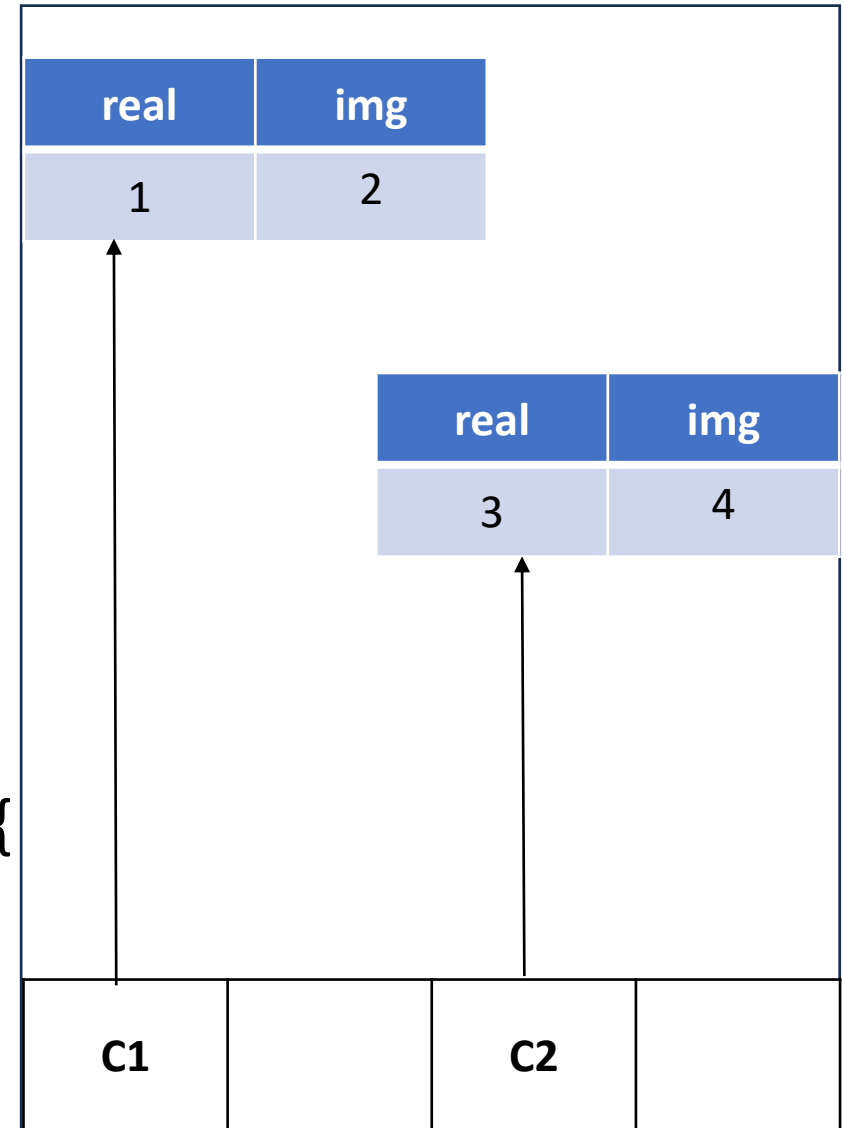
- **Static Member Characteristics:**

1. Describes the class (the Category)
2. Loaded in the memory when the class is loaded in the memory
3. **Exists** in the memory **ONLY once** even if it is a variable or method
4. The right way to be accessed is via the class name,  
but it can be accessed via an object or object method.

# Static Members Example

- `Complex c1 = new Complex( )`
- `Complex c2 = new Complex( )`
- `c1.setReal(1);`
- `c1.setImg(2);`
- `c2.setReal(3);`
- `c2.setImg(4);`

```
public static Complex sum(Complex c1, Complex c2){  
    Complex result = new Complex(c1.real + c2.real,  
                                  c1.img + c2.img);  
    return result;  
}
```

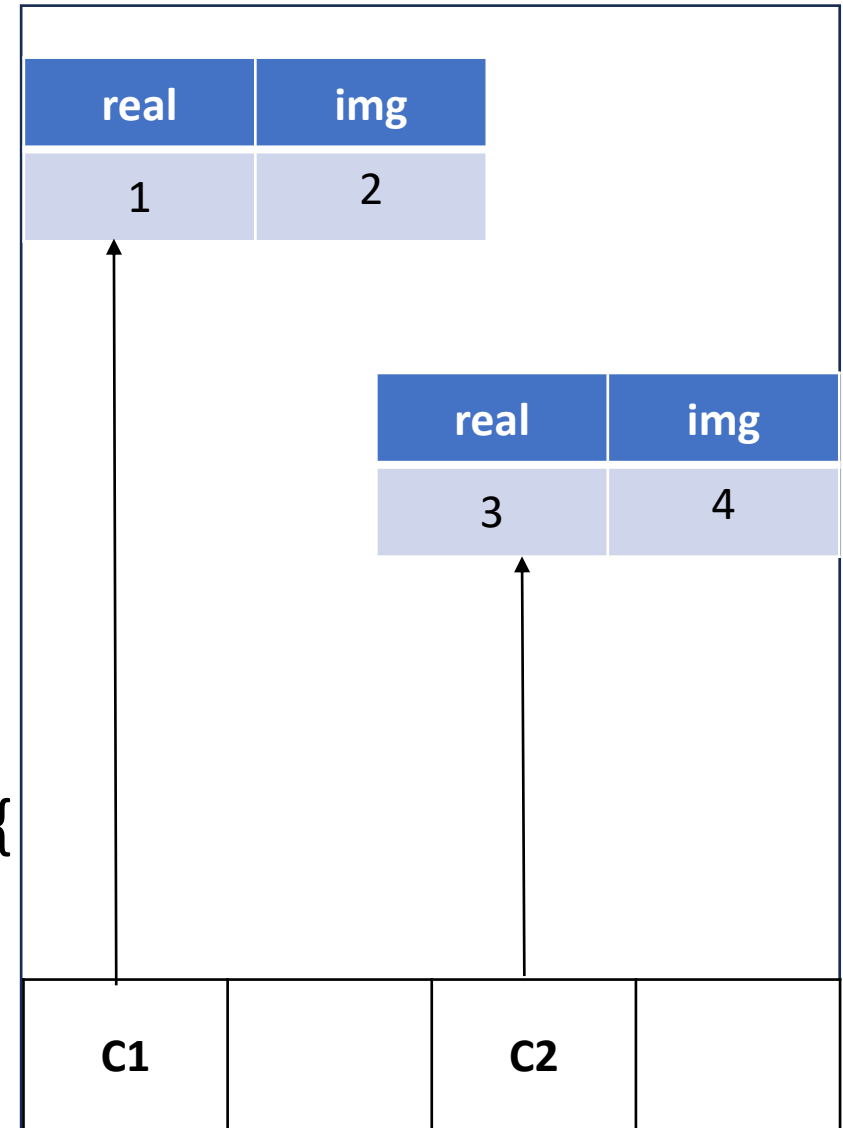


# Static Members Example

- `Complex c1 = new Complex( )`
- `Complex c2 = new Complex( )`
- `c1.setReal(1);`
- `c1.setImg(2);`
- `c2.setReal(3);`
- `c2.setImg(4);`

```
public static Complex sum(Complex c1, Complex c2){  
    Complex result = new Complex(c1.real + c2.real,  
                                  c1.img + c2.img);  
    return result;  
}
```

- `Complex result = Complex.sum(c1, c2);`

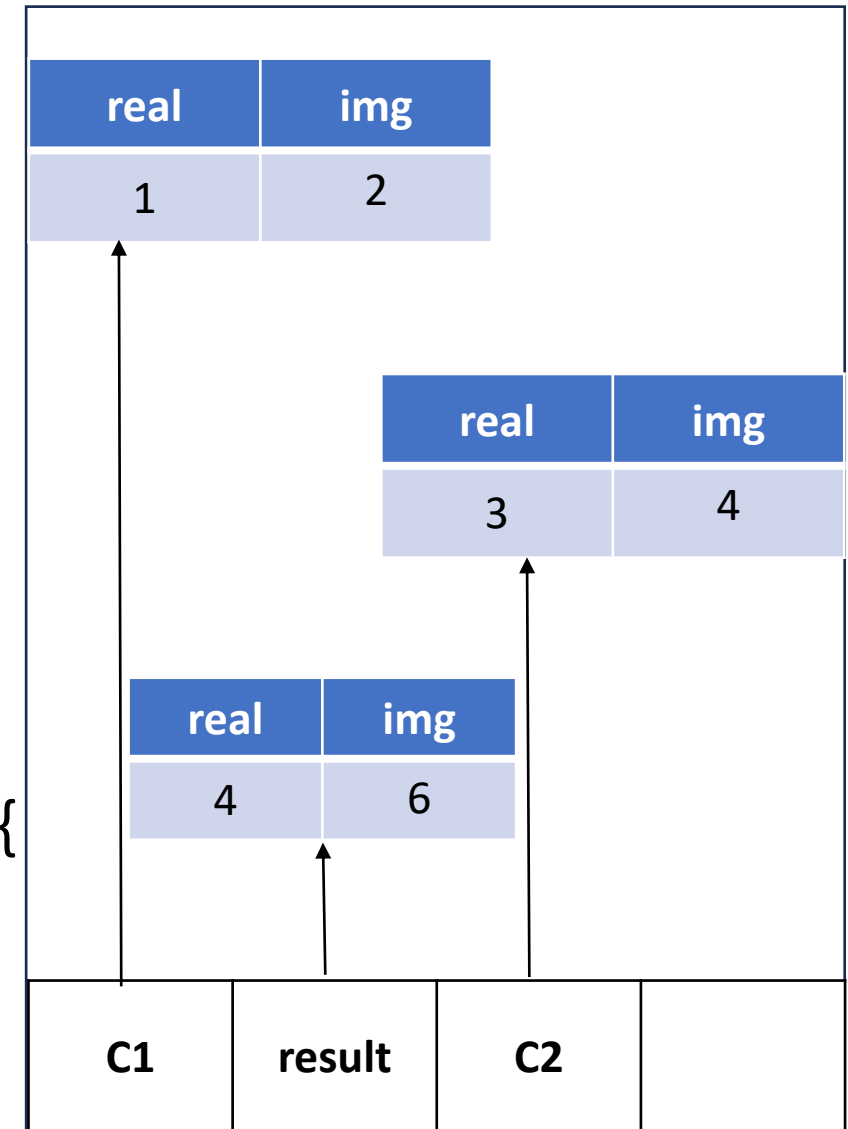


# Static Members Example

- `Complex c1 = new Complex( )`
- `Complex c2 = new Complex( )`
- `c1.setReal(1);`
- `c1.setImg(2);`
- `c2.setReal(3);`
- `c2.setImg(4);`

```
public static Complex sum(Complex c1, Complex c2){  
    Complex result = new Complex(c1.real + c2.real,  
                                  c1.img + c2.img);  
    return result;  
}
```

- `Complex result = Complex.sum(c1,c2);`



# Lab Assignments

- Create a simple non-GUI Application that represent Student and has method to print student info.

**Student:** name, age, track.....

- Create a simple non-GUI Application that represent complex number and has three methods to **add**, **subtract** and **print** complex numbers:

**Complex number:**  $x + yi$  , 5+6i