



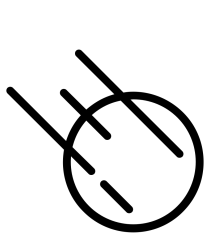
# Object-Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code. The data is in the form of fields (often known as attributes or properties), and the code is in the form of procedures (often known as methods).

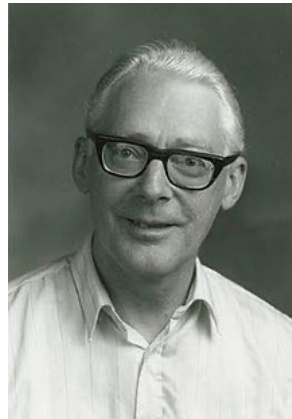


# Reading

- Read section II and III of the paper “Has the Object-Oriented Paradigm Kept Its Promise?”
  - [lutzhamel.github.io/CSC493/docs/OOPP.pdf](https://lutzhamel.github.io/CSC493/docs/OOPP.pdf)
- If you are interested, take a peek at Bertrand Meyer’s classic “Object-Oriented Software Construction”. Of particular interest is Section D
  - [lutzhamel.github.io/CSC493/docs/OOSC.pdf](https://lutzhamel.github.io/CSC493/docs/OOSC.pdf)



# Origins of OOP



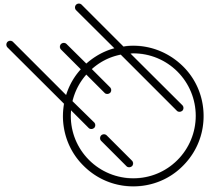
Ole-Johan Dahl, Professor  
Computer Science, 1931-2002



Kristen Nygaard, Computer  
Scientist, 1926-2002

- **Simula** is a language designed to solve problems in simulations
  - Introduced objects, classes, inheritance and subclasses, and featured garbage collection.
    - Inspired by the observation that simulations become more robust when object state and behavior are bundled
  - Considered to be the first truly object-oriented programming language.
  - Developed in the 1960's

**simula**



# Origins of OOP

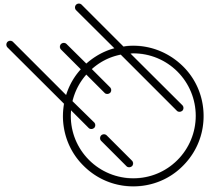


Dr Alan Kay, computer scientist,  
1940-

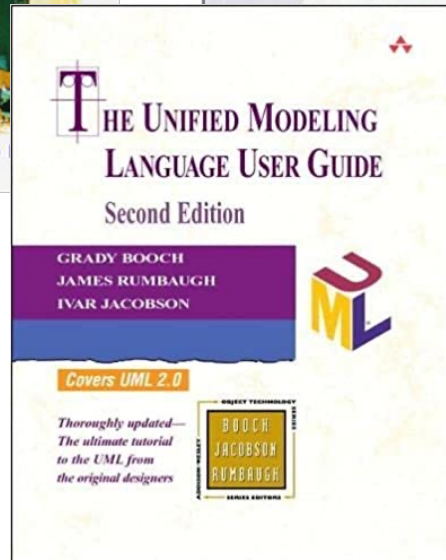
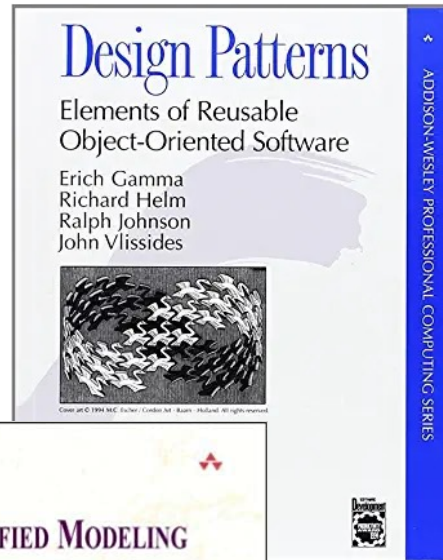
## ○ Smalltalk

- Influenced by the ideas in Simula and Lisp
- Pure OO, “*Everything is an object*” – even primitive entities like integers
- Highly influential because of the pure OO aspect
  - Other languages make tradeoffs due to performance issues
- Developed in the 1970’s
- Open-source modern implementation
  - <https://squeak.org/>





# Origins of OOP

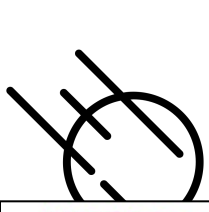


- Some classics from the “classic” OOP period
  - Bertrand Meyer, designer of Eiffel, design by contract
  - Grady Booch, inventor of UML
  - “Gang of Four”, design patterns were hugely influential on OO architectures and OOP



# Object-Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm that uses objects and their interactions to design applications and frameworks (e.g. GUI, webservers).
- It is based on the concept of "objects", which can contain data and code that manipulates that data and an **object identity**.
- “Classic” OOP languages, such as Java and C++ provide features such as encapsulation, inheritance, and polymorphism to help organize and reuse code.
  - **Encapsulation** refers to the practice of keeping an object's internal state and behavior hidden from the outside world, while exposing a public interface.
  - **Inheritance** allows one class to inherit properties and methods from a parent class.
  - **Polymorphism** allows objects of different classes to be treated as objects of a common superclass.



# Object Identity

```
#include <assert.h>
#include <stdlib.h>

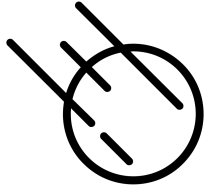
typedef struct rect {
    float xdim;
    float ydim;
} Rect;

Rect* makerect(float x, float y) {
    Rect* r = (Rect*) malloc(sizeof(Rect));
    r->xdim = x;
    r->ydim = y;
    return r;
}

float area(Rect* r) {
    return r->xdim * r->ydim;
}

int main() {
    Rect* r = makerect(2.0, 4.0);
    float a = area(r);
    assert(a == 8.0);
    free(r);
    return 0;
}
```

- In non-OOP setting objects only have external identity (reference)
  - Consider the C code on the left – objects/members are only accessed via external identity
- This changes in the OOP setting where member functions can refer to the object they belong to via an “internal” identity



# Object Identity – OOP

- In Python "self" refers to the internal object identity
- Here we see that the area function access members "internally".

```
class Rect:
    def __init__(self, x, y):
        self.xdim = x
        self.ydim = y

    def area (self):
        return self.xdim * self.ydim # accessing internal identity via self

r = Rect(2.0, 4.0)
a = r.area ()
assert (a == 8.0)
```





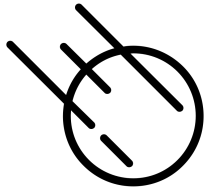
# Object Identity – OOP

- External object identity and internal object identity are the same!

```
class foo:
    def getid (self):
        return id(self)

o = foo()
external_id = id(o)
internal_id = o.getid()
assert(external_id == internal_id)
```

In007/id.py



# Encapsulation

```
class Person {
    private String name;
    private int age;

    public String getName() {
        return this.name;
    }

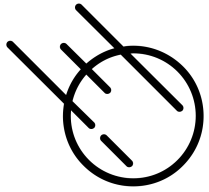
    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

class MyProgram {
    public static void main(String[] args) {
        Person joe = new Person();
        joe.setName("Joe");
        joe.setAge(32);
        System.out.println(joe.getName()+" is "+joe.getAge()+" years old");
    }
}
```

- Data members are “private”
- Setter/getter functions
- Pros: precise modeling of the notion of “object”
- Cons:
  - Cluttering of public interface with trivial setter/getter functions
  - Private members are not available to derived classes, which is strange because derived objects own this attribute
  - To solve this yet another access attribute: protected



# Inheritance

- Inheritance is one of the core concepts of object-oriented programming (OOP) languages.
- It is a mechanism where you can derive a class from another class for a hierarchy of classes that share a set of attributes and methods.
- In statically typed languages like C++/Java it allows for precise modeling of the perceived inheritance relation (is-a) of objects

Partial hierarchy  
of the Microsoft  
Foundation Classes  
(MFC)

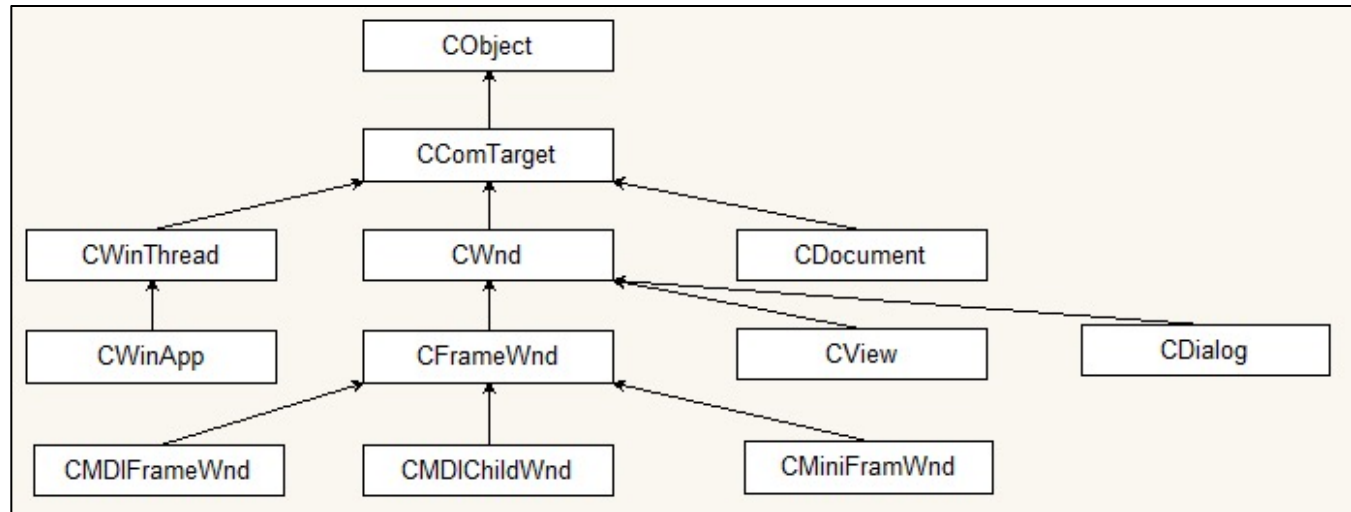
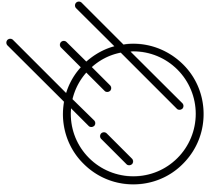



Image source: [https://commons.wikimedia.org/wiki/File:MFC\\_hierarchy.png](https://commons.wikimedia.org/wiki/File:MFC_hierarchy.png)



# Subtype Polymorphism

- Subtype polymorphism is a feature in object-oriented programming (OOP) in which a subclass or derived class can be used in place of its superclass or base class.
- This means that an object of a subclass can be treated as an object of its superclass, and it will respond to the same methods and properties as an object of the superclass.
- This allows for more flexibility and reusability in code, as objects can be treated generically and interchangeably based on their common base class(es), rather than having to be treated as specific instances of a class.



```
import java.util.*;
import java.util.Vector;

class Shape {
    public void draw() { System.out.println("Drawing a shape."); }
}

class Circle extends Shape {
    private String name;
    Circle(String name) { this.name = name; }
    public void draw() { System.out.println("Drawing a circle "+this.name); }
}

class Square extends Shape {
    private String name;
    Square(String name) { this.name = name; }
    public void draw() { System.out.println("Drawing a square "+this.name); }
}

class Main {
    public static void main(String[] args) {
        Vector<Shape> v = new Vector<Shape>(3);
        v.add(new Circle("Circle1"));
        v.add(new Square("Square1"));
        v.add(new Circle("Circle2"));
        for (int i = 0; i < v.size(); i++) {
            v.get(i).draw();
        }
    }
}
```


Generic Shape Container



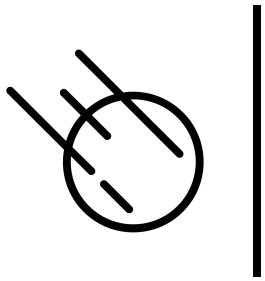
Adding specific Shapes



Dynamic dispatching  
to call the correct draw()  
method

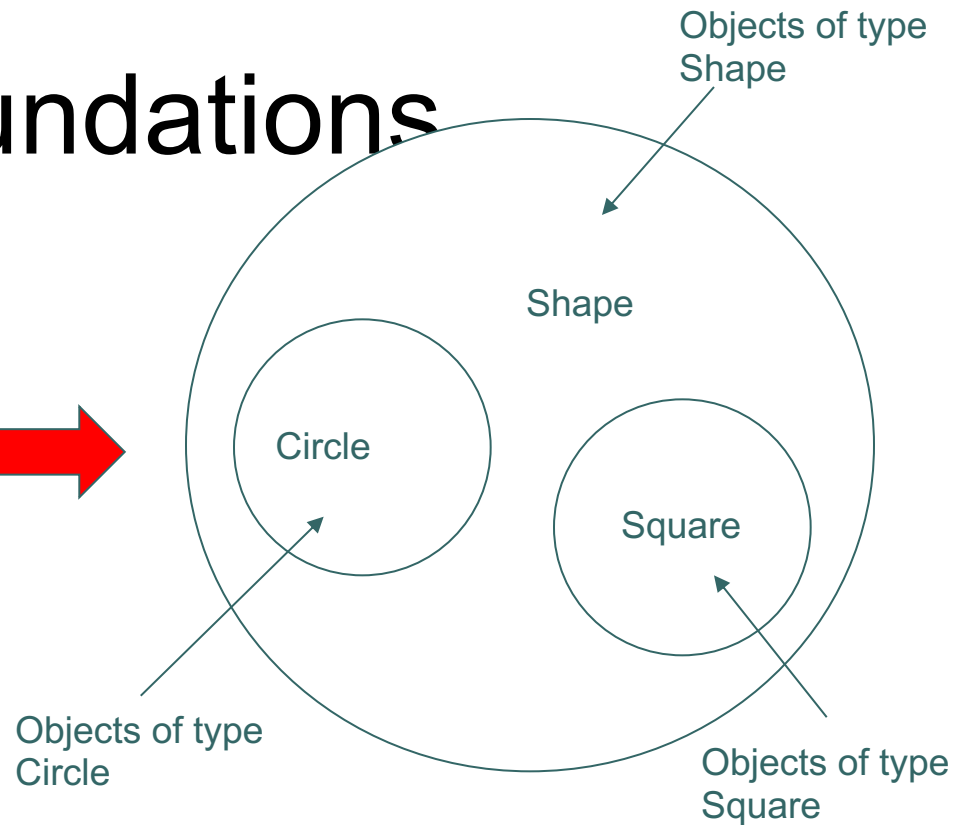
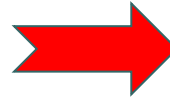


In007/subpoly.java

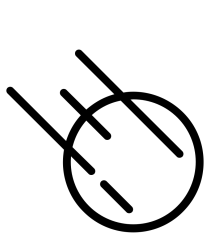


# OOP – Foundations

```
class Shape { ...  
}  
  
class Circle extends Shape { ...  
}  
  
class Square extends Shape { ...  
}
```

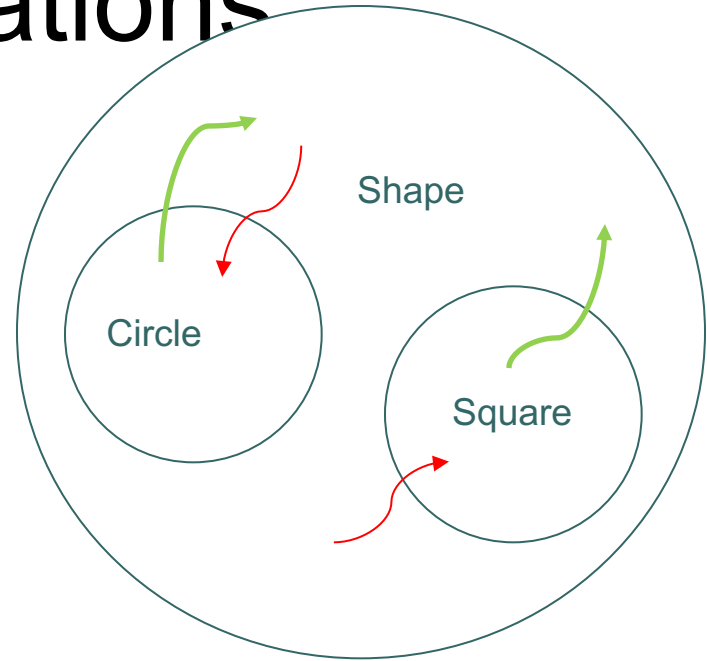


- A class defines a type
- A type is a set of values
- The values of a type defined by a class are the objects that can be instantiated from that class, e.g.
  - `new Circle()`

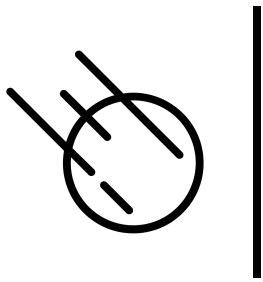


# OOP – Foundations

```
class Shape { ...  
}  
  
class Circle extends Shape { ...  
}  
  
class Square extends Shape { ...  
}
```

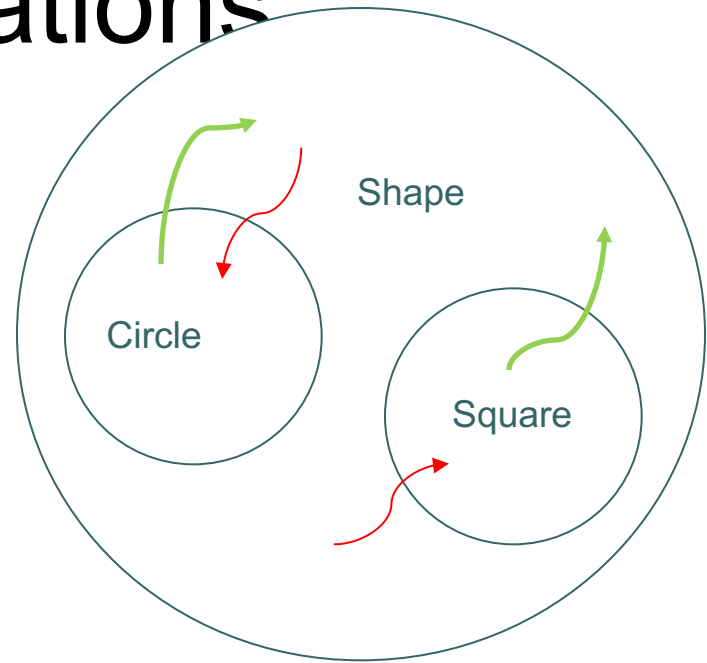


- From our class hierarchy we have
  - Circle < Shape
  - Square < Shape
- From our interpretation of subtypes as subsets we have
  - Every Circle is a Shape, and every Square is a Shape; green arrows, **widening conversion**
  - But not every Shape is either a Circle or a Square; red arrows, **narrowing conversion**



# OOP – Foundations

```
class Shape { ...  
}  
  
class Circle extends Shape { ...  
}  
  
class Square extends Shape { ...  
}
```



```
14 public static void main(String[] args) {  
15     Shape s1 = new Circle();  
16     Shape s2 = new Square();  
17     Circle c = new Shape();  
18     Square sq = new Shape();  
19 }
```

```
ubuntu$ javac subpoly1.java
```

```
subpoly1.java:17: error: incompatible types: Shape cannot be converted to Circle  
    Circle c = new Shape();  
                  ^
```

```
subpoly1.java:18: error: incompatible types: Shape cannot be converted to Square  
    Square sq = new Shape();  
                  ^
```

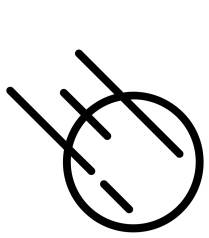
```
2 errors
```





# OOP – Subtype Polymorphism

- Now that we have looked at the set theoretic foundations of OOP let's take another look at subtype polymorphism
- In particular, the idea of dynamic dispatch which makes this so extremely useful



# OOP – Subtype Polymorphism

- We will use the example from before: create a list of circles and squares and then have each object on the list draw itself.
- Caveat: in statically types languages lists/vectors can only have homogeneously type elements
- Solution: Use a list/vector where the elements are elements of the base type of our Shape hierarchy BUT we insert our actual Circle and Square objects.

```
class Shape { ...  
}
```

```
class Circle extends Shape { ...  
}
```

```
class Square extends Shape { ...  
}
```

```
Vector<Shape> v = new Vector<Shape>(3);  
v.add(new Circle("Circle1"));  
v.add(new Square("Square1"));  
v.add(new Circle("Circle2"));
```

We say that *v* is a subtype polymorphic list/vector because the objects on the list/vector consist of different subtypes of Shape.



# OOP – Subtype Polymorphism

- Finally, dynamic dispatch makes this all work.
- In the code below we call the draw function on the base class Shape
- But what is actually called are the draw functions of the subtypes – dynamic dispatch

```
Drawing a circle Circle1  
Drawing a square Square1  
Drawing a circle Circle2
```

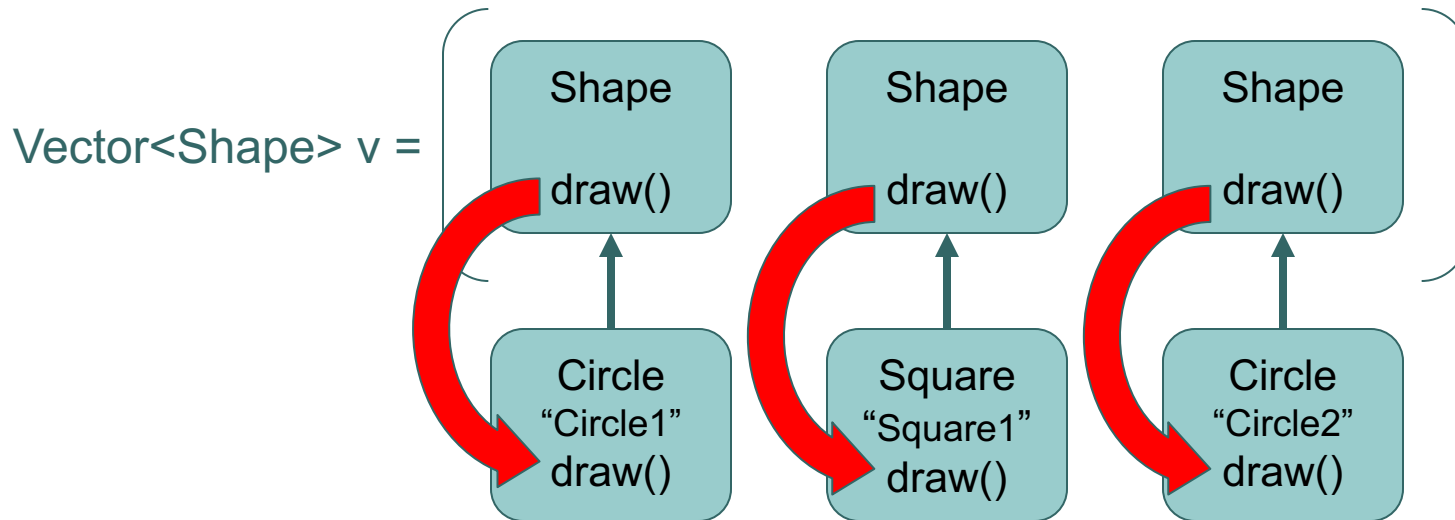
```
class Shape {  
    public void draw() { System.out.println("Drawing a shape."); }  
}  
  
class Circle extends Shape {  
    private String name;  
    Circle(String name) { this.name = name; }  
    public void draw() { System.out.println("Drawing a circle "+this.name); }  
}  
  
class Square extends Shape {  
    private String name;  
    Square(String name) { this.name = name; }  
    public void draw() { System.out.println("Drawing a square "+this.name); }  
}
```

```
Vector<Shape> v = new Vector<Shape>(3);  
v.add(new Circle("Circle1"));  
v.add(new Square("Square1"));  
v.add(new Circle("Circle2"));  
for (int i = 0; i < v.size(); i++) {  
    v.get(i).draw();  
}
```



# OOP – Subtype Polymorphism

```
Vector<Shape> v = new Vector<Shape>(3);  
v.add(new Circle("Circle1"));  
v.add(new Square("Square1"));  
v.add(new Circle("Circle2"));  
for (int i = 0; i < v.size(); i++) {  
    v.get(i).draw();  
}
```



- Dynamic dispatch realizes when calling the draw function of the base class that a more specific draw function exists and calls that instead of the draw function of the base class.



# Dynamically Typed Languages

- In dynamically typed languages like Asteroid and Python lists are untyped containers, i.e.
  - [1,"two",3.0] is legal
- That means, lists in dynamically typed languages are by default polymorphic!
- Here, **subtype polymorphism** with dynamic dispatch is **replaced by duck typing**,
  - *"If it walks like a duck and it quacks like a duck, then it must be a duck"* – the duck test.



# Dynamically Typed Languages

```
class Circle:
    def __init__(self, name):
        self.name = name
    def draw(self):
        print("Drawing a circle "+str(self.name))

class Square:
    def __init__(self, name):
        self.name = name
    def draw(self):
        print("Drawing a square "+str(self.name))

v = []
v.append(Circle("Circle1"))
v.append(Square("Square1"))
v.append(Circle("Circle2"))
for i in range(len(v)):
    v[i].draw();
```

In007/subpoly.py

- In duck typing the objects on a list have to support the behavior required by the list
  - However, that behavior does not have to come from a base class!
- Consider the shape example written in Python
  - no base class required
  - list is polymorphic

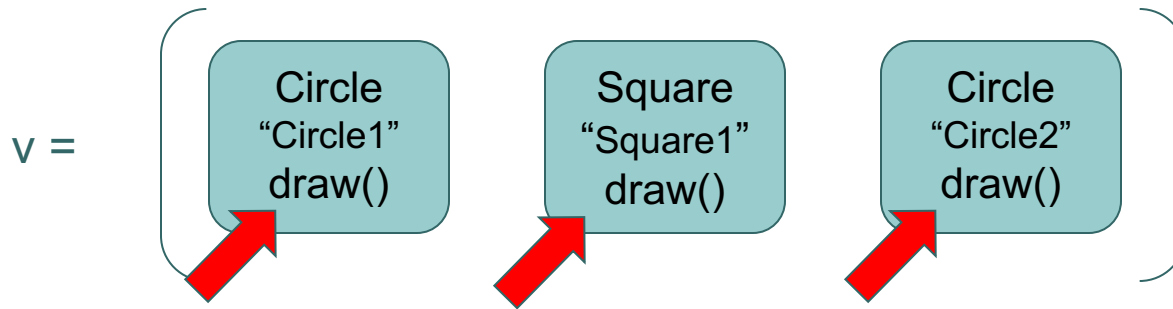


# OOP – Duck Typing

```
class Circle:
    def __init__(self, name):
        self.name = name
    def draw(self):
        print("Drawing a circle "+str(self.name))

class Square:
    def __init__(self, name):
        self.name = name
    def draw(self):
        print("Drawing a square "+str(self.name))

v = []
v.append(Circle("Circle1"))
v.append(Square("Square1"))
v.append(Circle("Circle2"))
for i in range(len(v)):
    v[i].draw();
```



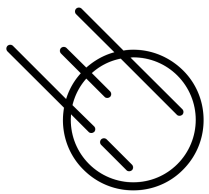
- Duck typing simply assumes that the required functions are present at runtime.



# OOP -- Criticisms

- “Classic” OOP is increasingly coming under scrutiny
- More modern approaches try to address this





# Inheritance

- Cons:

- **Static structure:** difficult to evolve with changing software requirements
- **Aggregation:** classes at the leaves inherit ALL of the data members and functions of the preceding classes

Partial hierarchy  
of the Microsoft  
Foundation Classes  
(MFC)

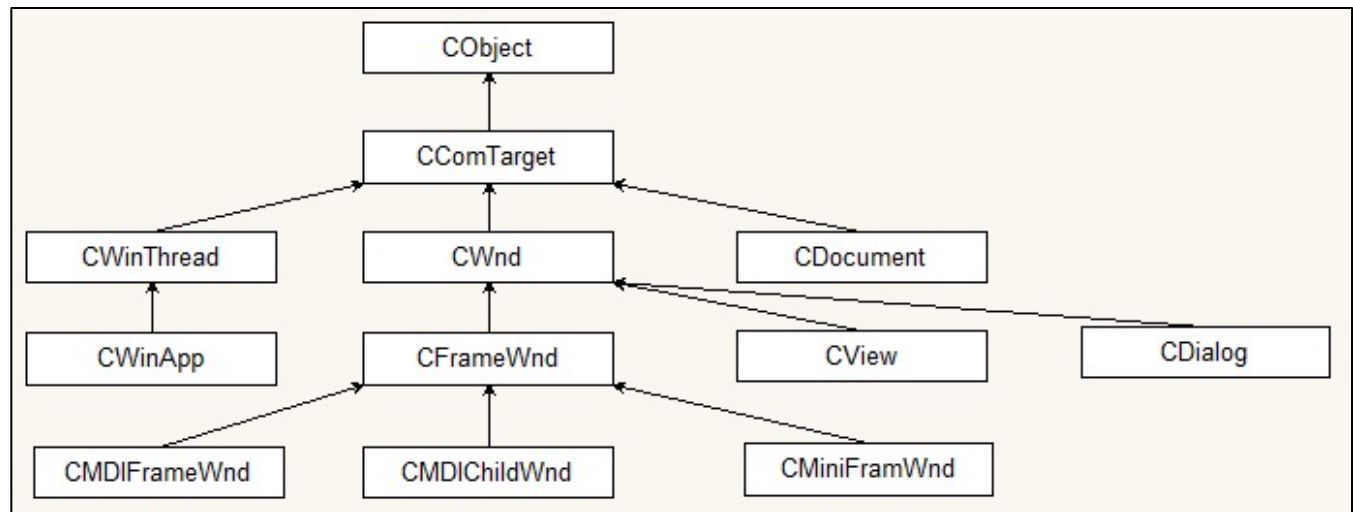
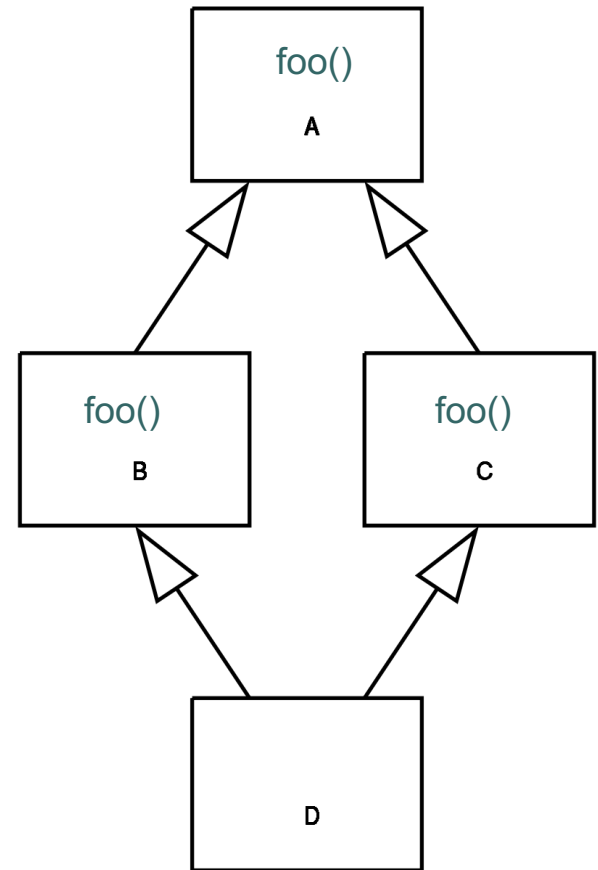


Image source: [https://commons.wikimedia.org/wiki/File:MFC\\_hierarchy.png](https://commons.wikimedia.org/wiki/File:MFC_hierarchy.png)



# Multiple-Inheritance – The Diamond Problem

- Briefly:
  - An ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C.
  - If there is a method in A that B and C have overridden, and D does not override it, then which version of the method does D inherit: that of B, or that of C?
  - That is: D.foo() – which foo() should be called?
- This gets really problematic in deep inheritance structures.

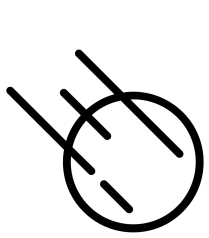




# The Diamond Problem

- Different languages deal with the diamond problem in different ways
  - C++ uses a fully qualified syntax
  - Python uses a class hierarchy linearization algorithm (C3 linearization or MRO) to resolve ambiguities
  - Java does not support multiple inheritance.

MRO: Method Resolution Order

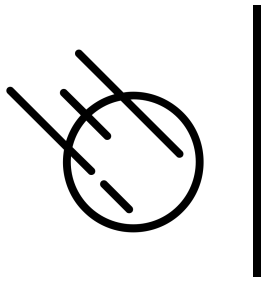


# The Diamond Problem

- In Python, the method that is called depends on the order of your inheritance specification !?!

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass
...
>>> class D(B,C): ←
...     pass
...
>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass
...
>>> class D(C,B): ←
...     pass
...
>>> D.mro()
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.B'>, <class '__main__.A'>, <class 'object'>]
```



# Other Criticisms

[Luca Cardelli](#) claims that OOP code is "intrinsically less efficient" than procedural code, that OOP can take longer to compile, and that OOP languages have "extremely poor modularity properties with respect to class extension and modification" and tend to be extremely complex.

[Joe Armstrong](#), the principal inventor of [Erlang](#), is quoted as saying: The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

[Alexander Stepanov](#) (main author of the C++ STL) compares object orientation unfavorably to [generic programming](#): I find OOP philosophically unsound. It claims that everything is an object. Even if it is true, it is not very interesting — saying that everything is an object is saying nothing at all.

[Eric S. Raymond](#), a [Unix](#) programmer and [open-source software](#) advocate, has been critical of claims that present object-oriented programming as the "One True Solution", and has written that object-oriented programming languages tend to encourage thickly layered programs that destroy transparency.

[Rob Pike](#), a programmer involved in the creation of [UTF-8](#) and [Go](#), has called object-oriented programming "the [Roman numerals](#) of computing" and cites an instance of a [Java](#) professor whose "idiomatic" solution to a problem was to create six new classes, rather than to simply use a [lookup table](#).



## On the other Hand...

- OO approaches work extremely well for GUI and related frameworks.
- Like everything else, OOP is just a tool to be used in situations where it makes sense,
  - Bundling behavior with state is in itself not a bad idea,
  - Trying to see every programming problem through this lens is...