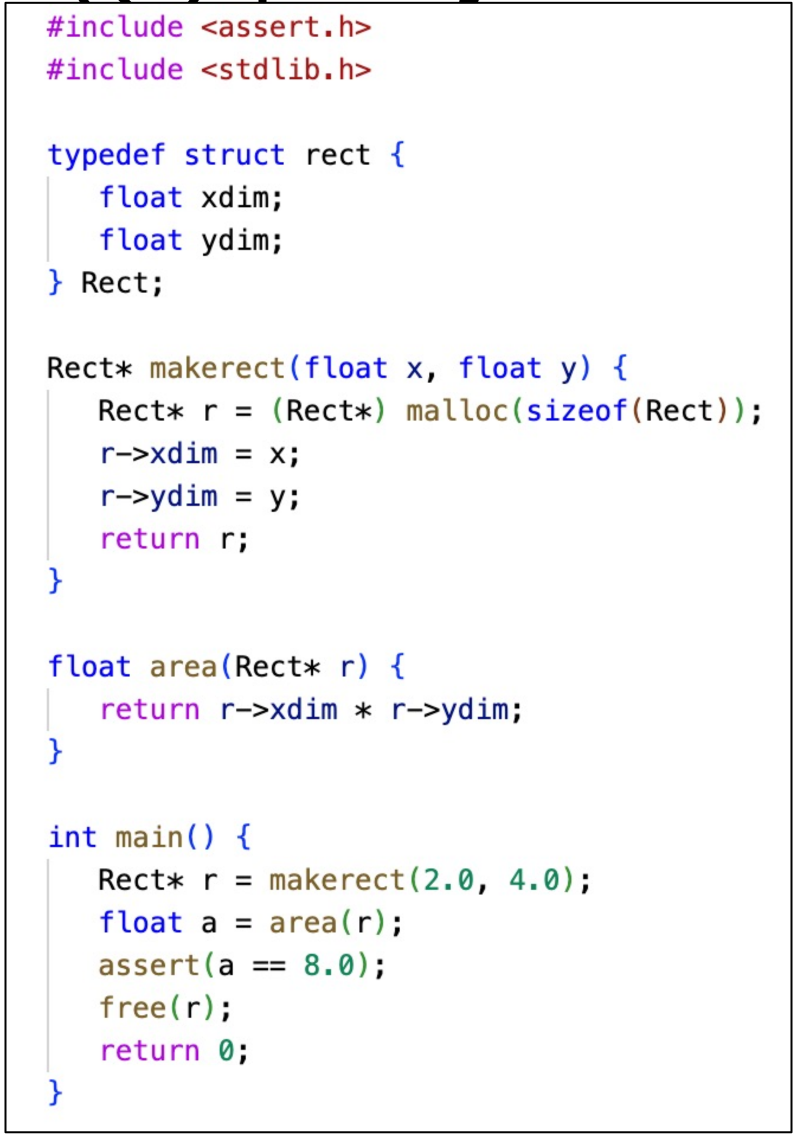
Object-Oriented Programming

- 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.

Reading

- Read section II and III of the paper "Has the Object-Oriented Paradigm Kept Its Promise?"
  - 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

Object Identity



```c
#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;
}
```

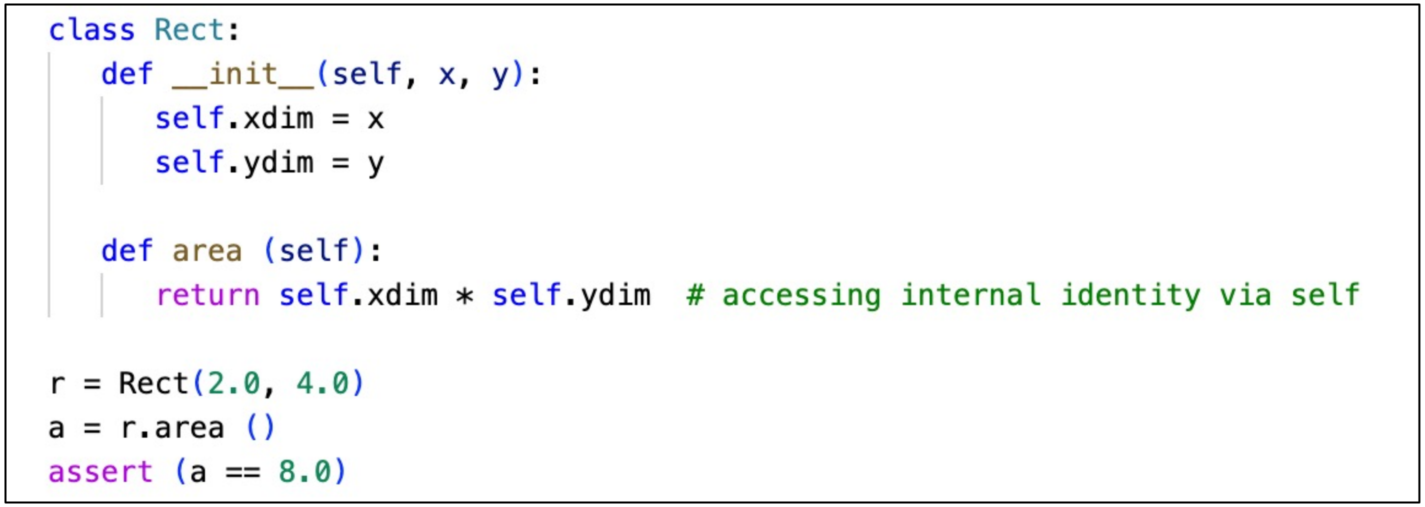
- In non-OOP setting objects only have external identity (reference)
- This changes in the OOP setting where member functions can refer to the object they belong to via an “internal” identity

ln007/rect.c

Object Identity – OOP

- In Python ”self” refers to the internal object identity

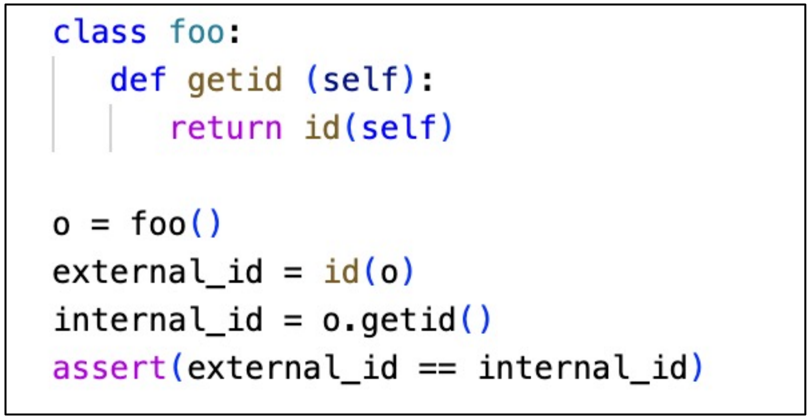
ln007/rect.py



```python
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

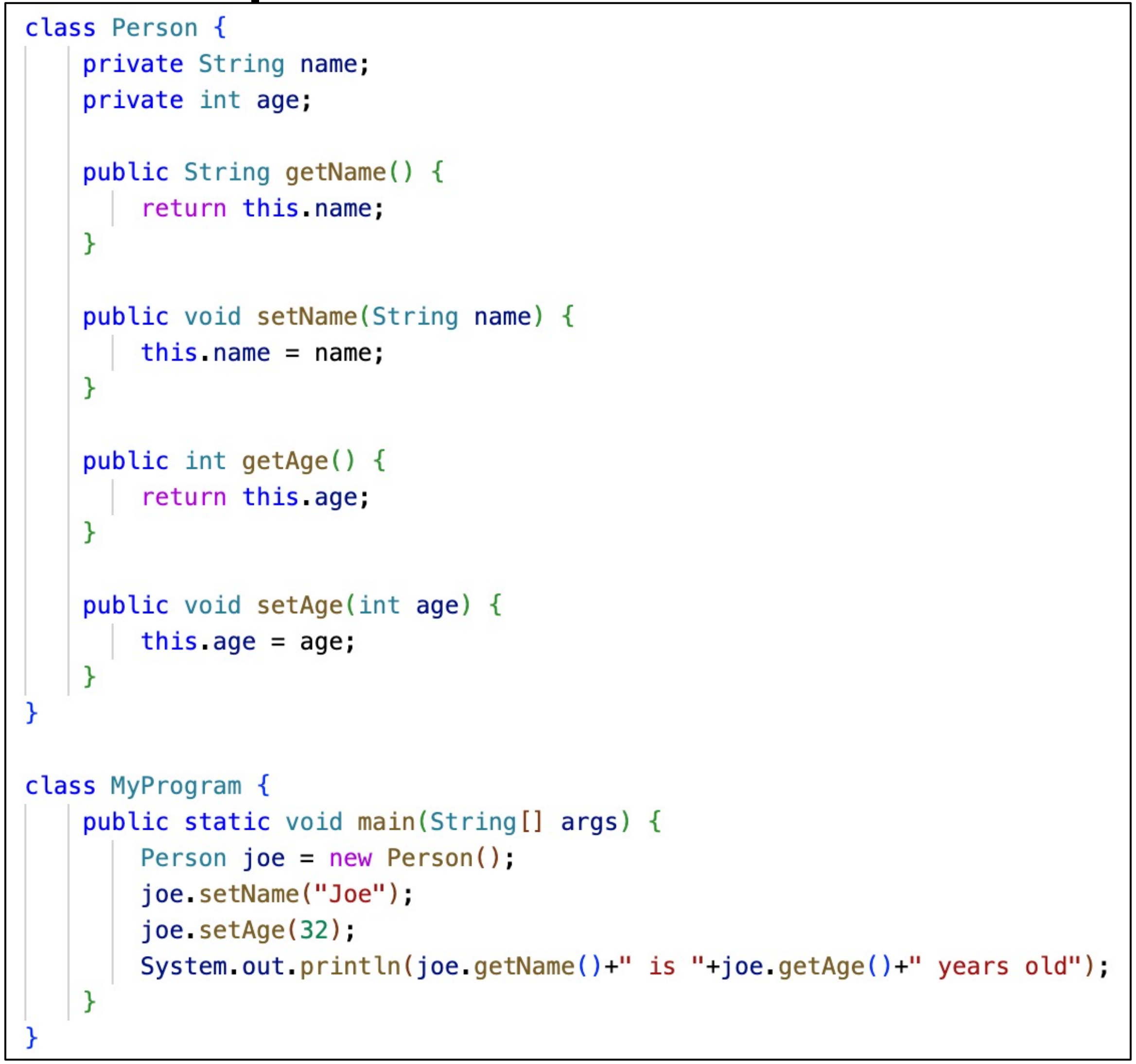
- External object identity and internal object identity are the same!



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

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

ln007/id.py

Encapsulation



```java
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

ln007/person.java

Inheritance

- Pros: precise modeling of the perceived inheritance relation (is-a) of objects
- 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)

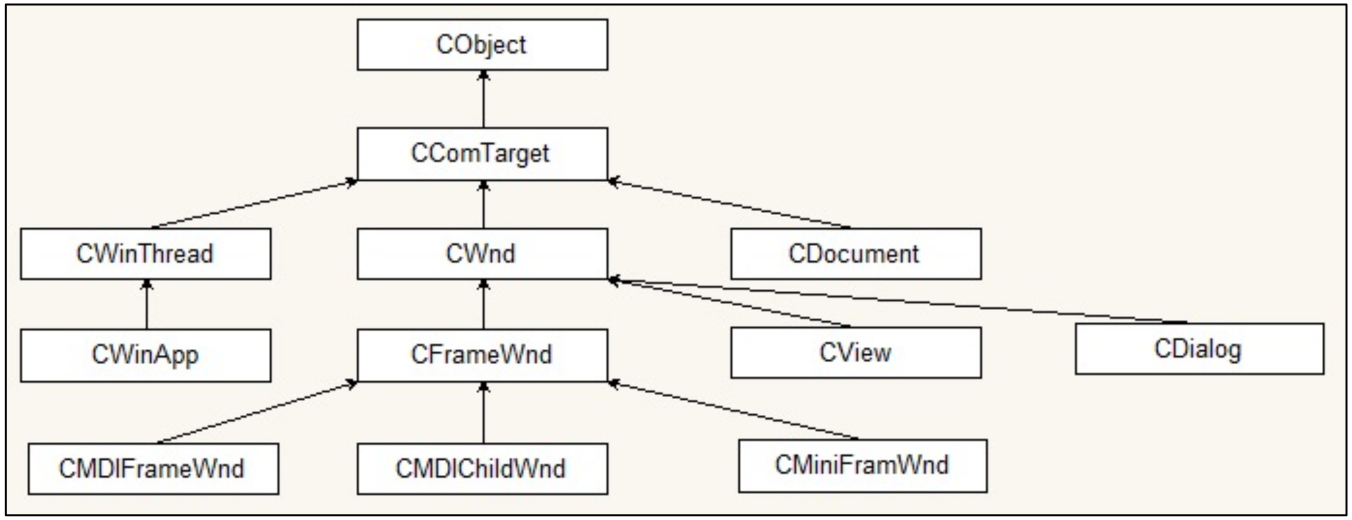
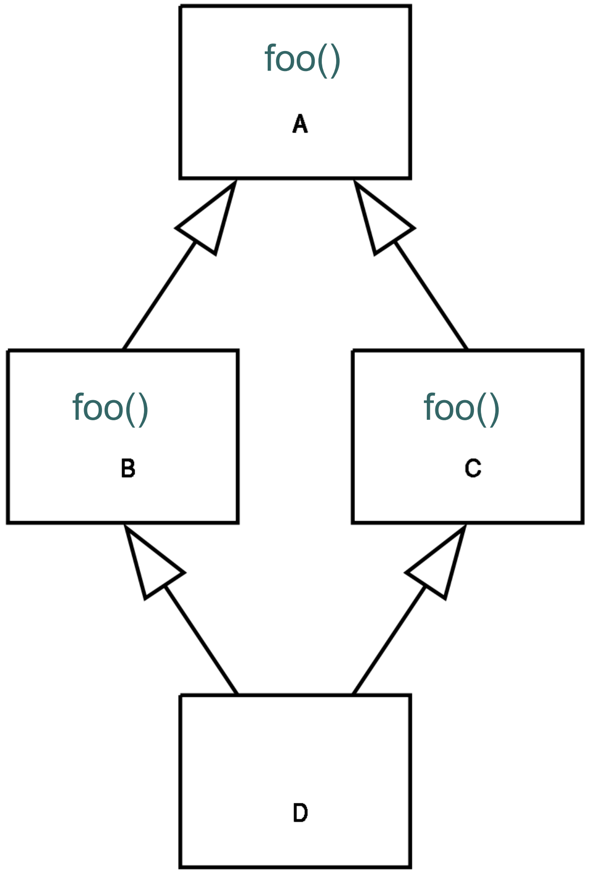


Image source: https://commons.wikimedia.org/wiki/File:MFC_hierarchy.png

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.



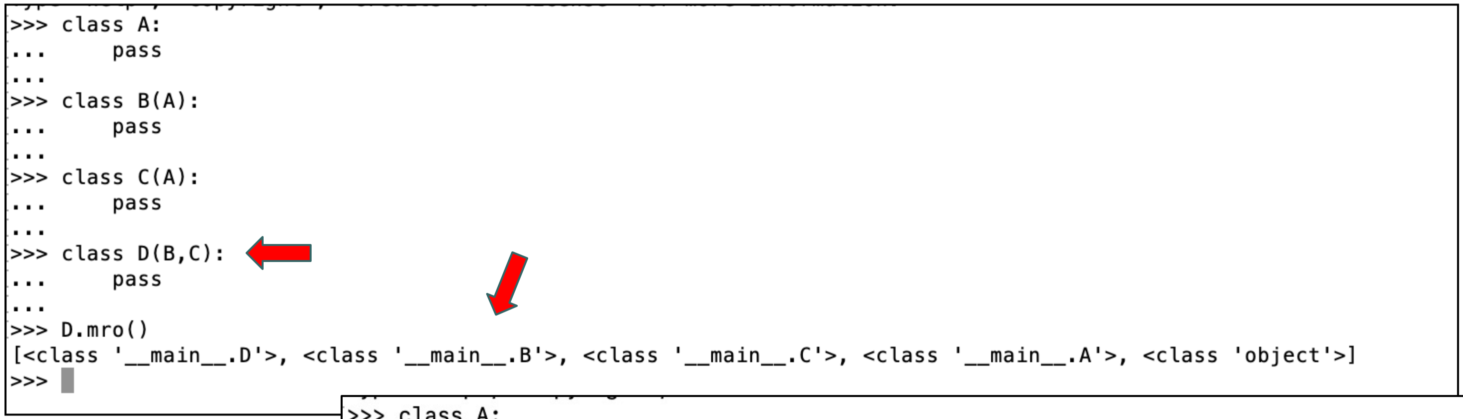
The Diamond Problem

- 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.

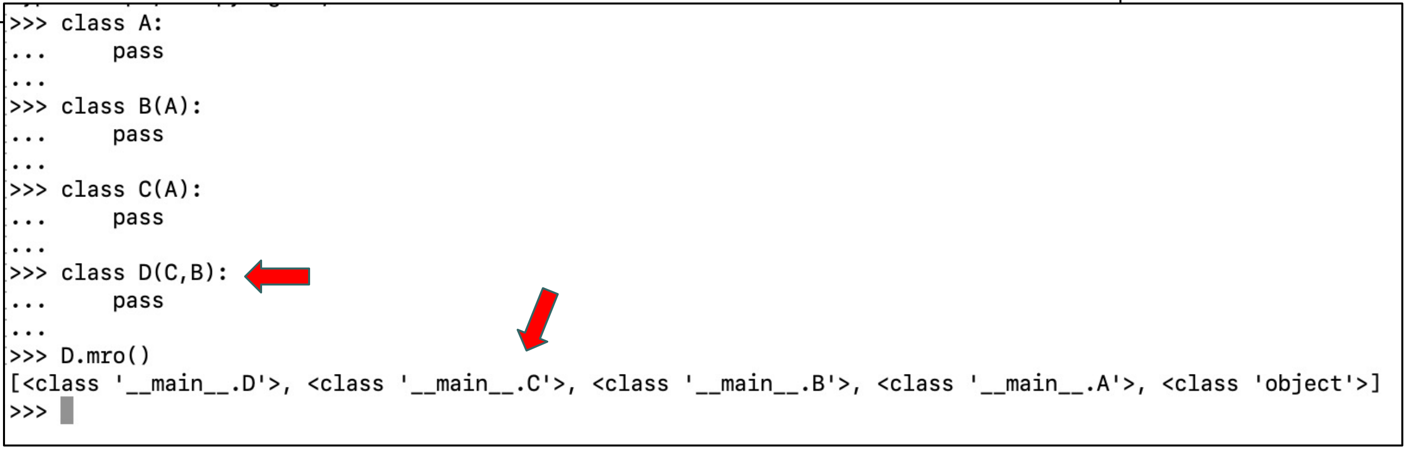
MRO: Method Resolution Order

The Diamond Problem

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



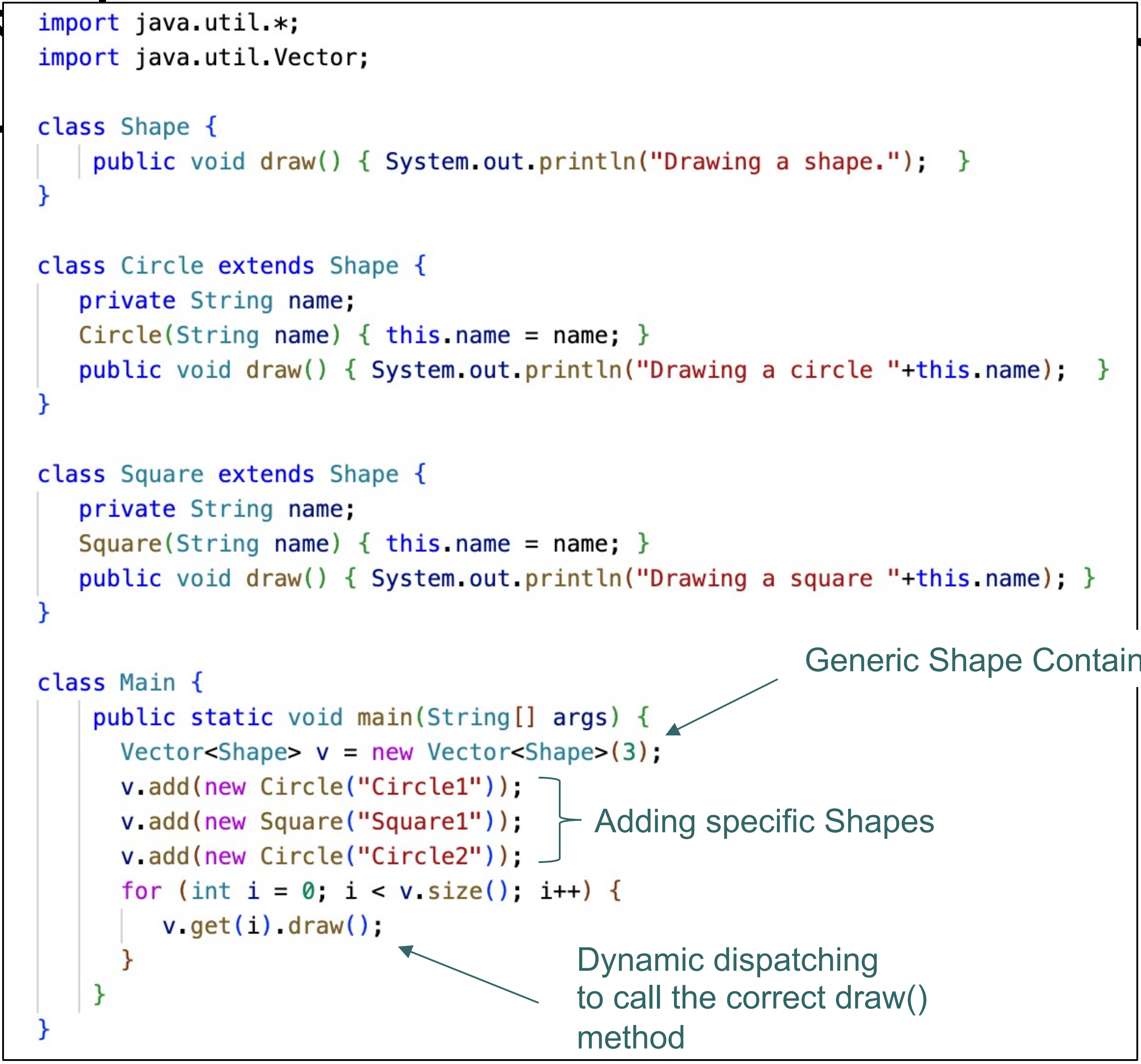
```
>>> class A:
...     pass
...
>>> class B(A):
...     pass
...
>>> class C(A):
...     pass
...
>>> class D(B,C):
...     pass
...
>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class 'object'>]
>>>
```



```
>>> 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'>]
>>>
```

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.



```java
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();
      }
    }
}
```

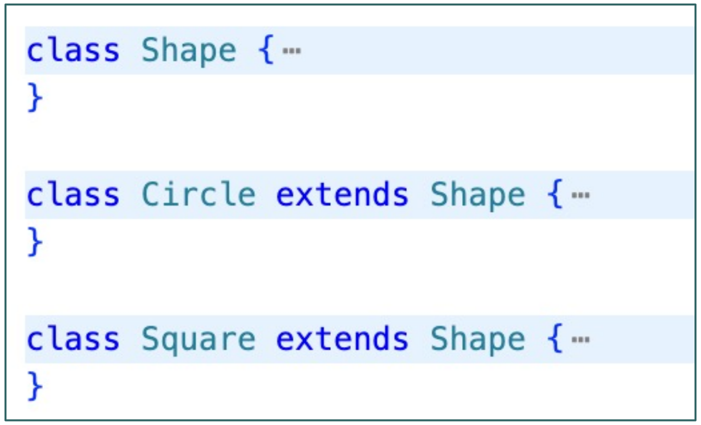
Generic Shape Container

Adding specific Shapes

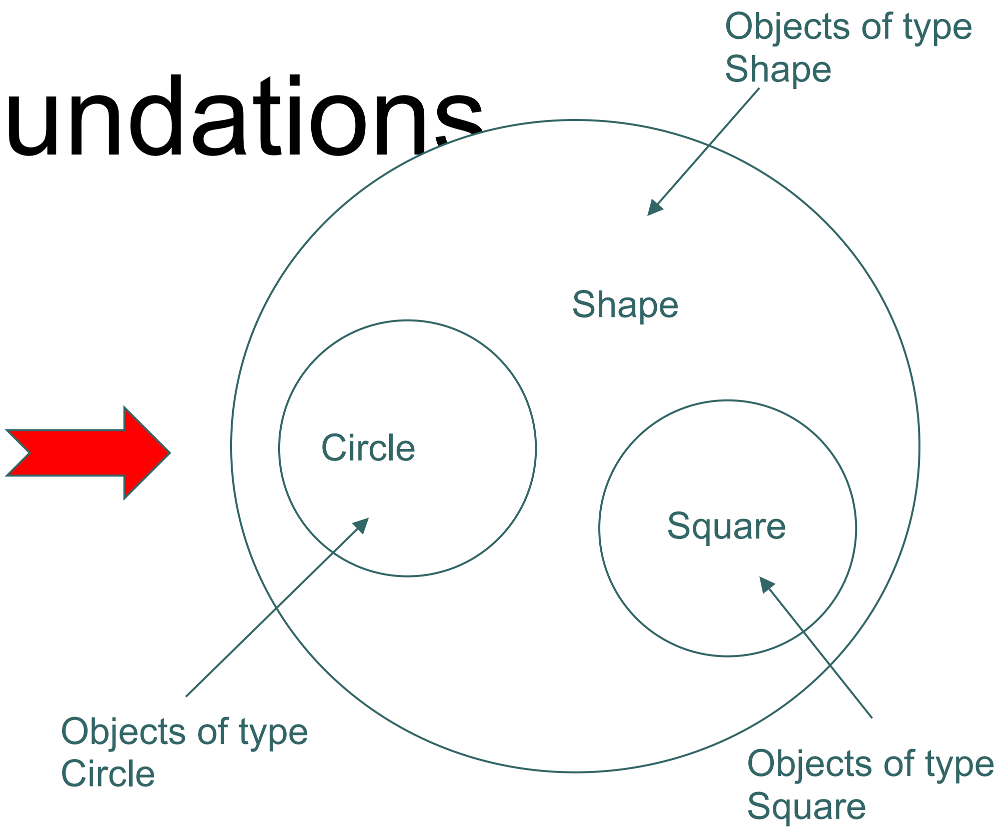
Dynamic dispatching
to call the correct draw()
method

ln007/subpoly.java

OOP – Foundations

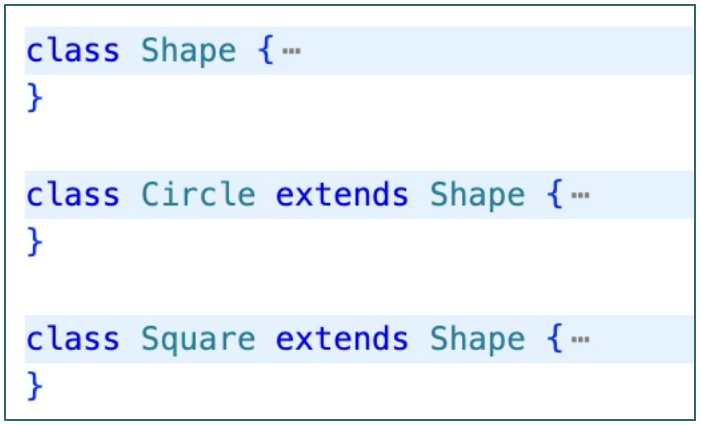


```
class Shape {⋯
}

class Circle extends Shape {⋯
}

class Square extends Shape {⋯
}
```

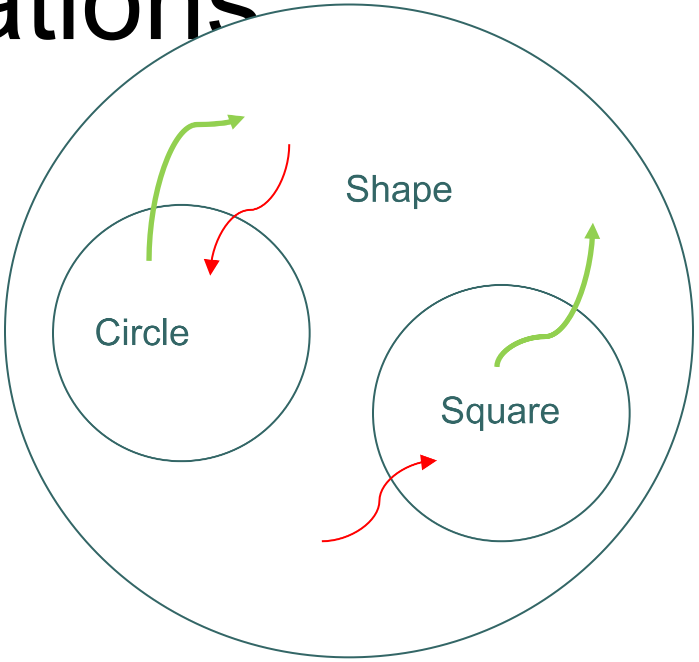


- A class defines a type
- A type is a set of values
- The values of a type defined by a class are the objects that can be instantiated from that class, e.g.
  - new Circle()

OOP – Foundations

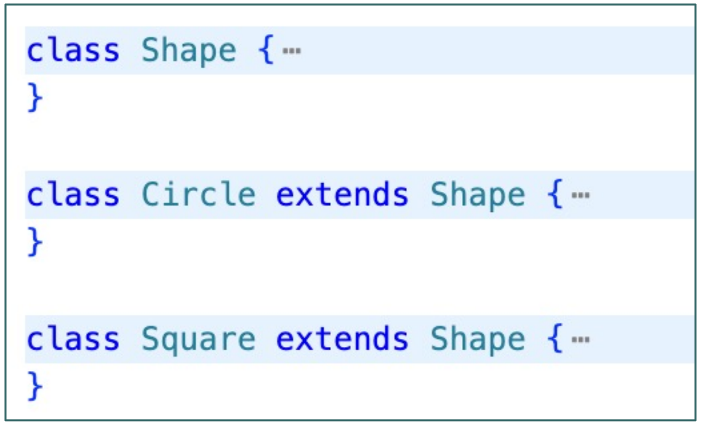


```
class Shape {⋯
}

class Circle extends Shape {⋯
}

class Square extends Shape {⋯
}
```

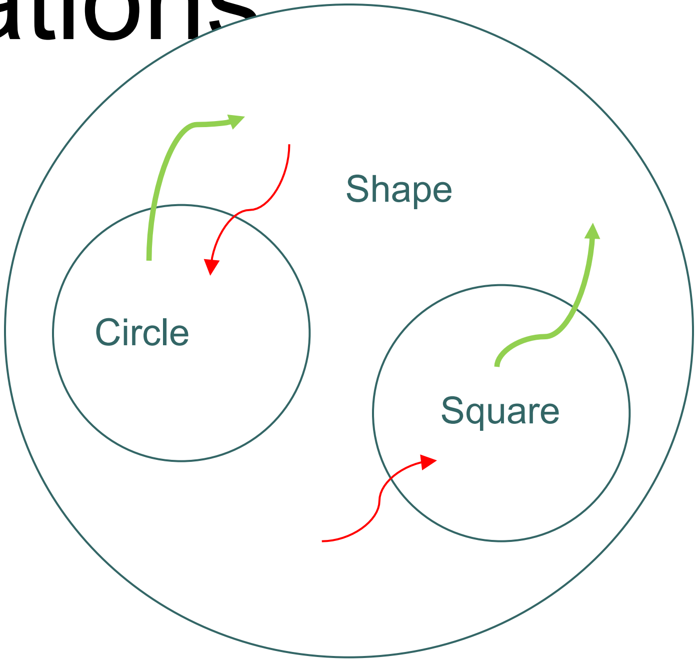


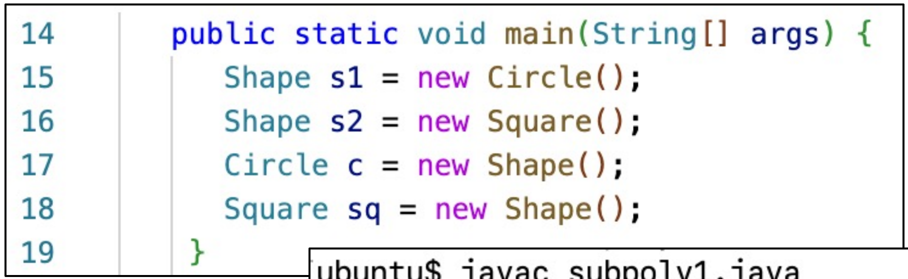
- From our class hierarchy we have
  - Circle < Shape
  - Square < Shape
- From our interpretation of subtypes as subsets we have
  - Every Circle is a Shape, and every Square is a Shape; green arrows, **widening conversion**
  - But not every Shape is either a Circle or a Square; red arrows, **narrowing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

OOP – Foundations

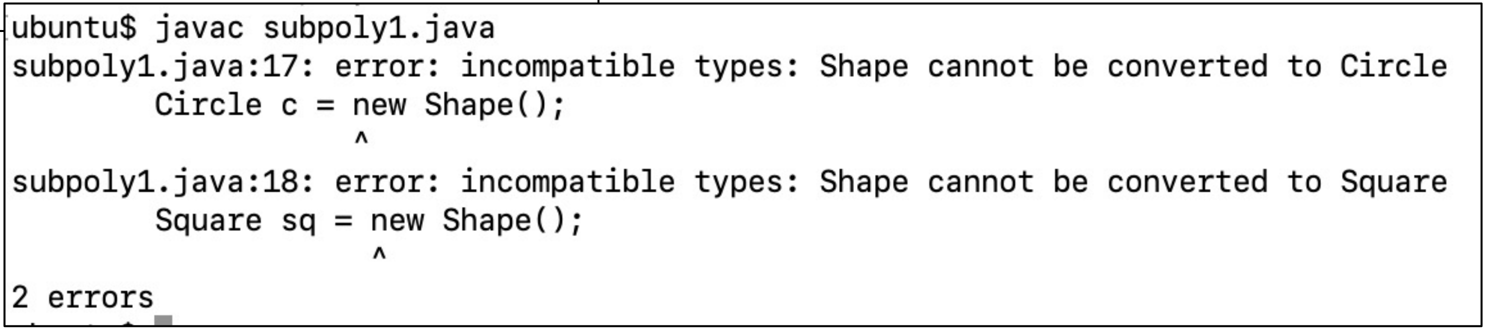


```
class Shape {…
}

class Circle extends Shape {…
}

class Square extends Shape {…
}
```





```
    public static void main(String[] args) {
        Shape s1 = new Circle();
        Shape s2 = new Square();
        Circle c = new Shape();
        Square sq = new Shape();
    }
```



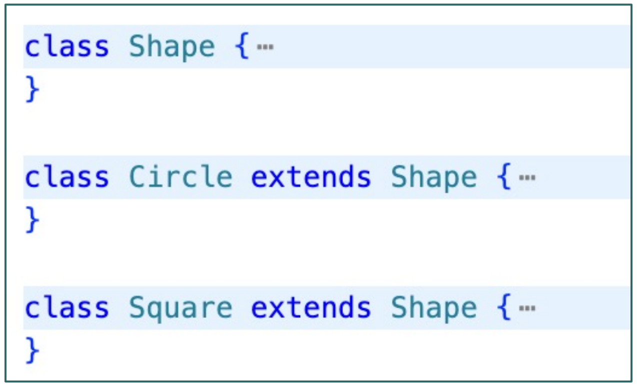
```
ubuntu$ javac subpoly1.java
subpoly1.java:17: error: incompatible types: Shape cannot be converted to Circle
        Circle c = new Shape();
                   ^
subpoly1.java:18: error: incompatible types: Shape cannot be converted to Square
        Square sq = new Shape();
                    ^
2 errors
```

OOP – Subtype Polymorphism

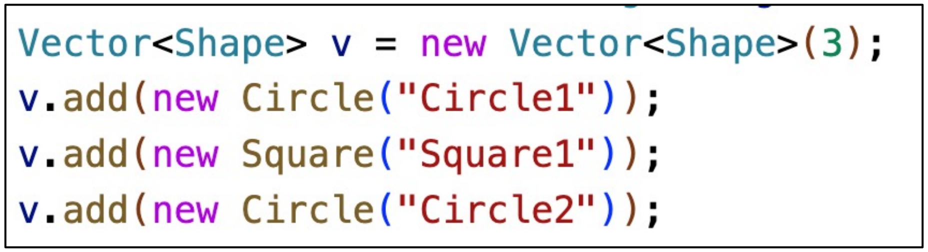
- Now that we have looked at the set theoretic foundations of OOP let’s take another look at subtype polymorphism
- In particular, the idea of dynamic dispatch which makes this so extremely useful

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 {…
}
```

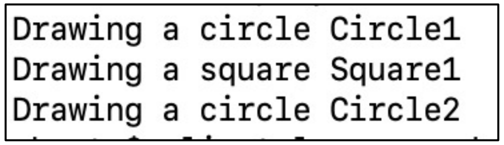


```
Vector<Shape> v = new Vector<Shape>(3);
v.add(new Circle("Circle1"));
v.add(new Square("Square1"));
v.add(new Circle("Circle2"));
```

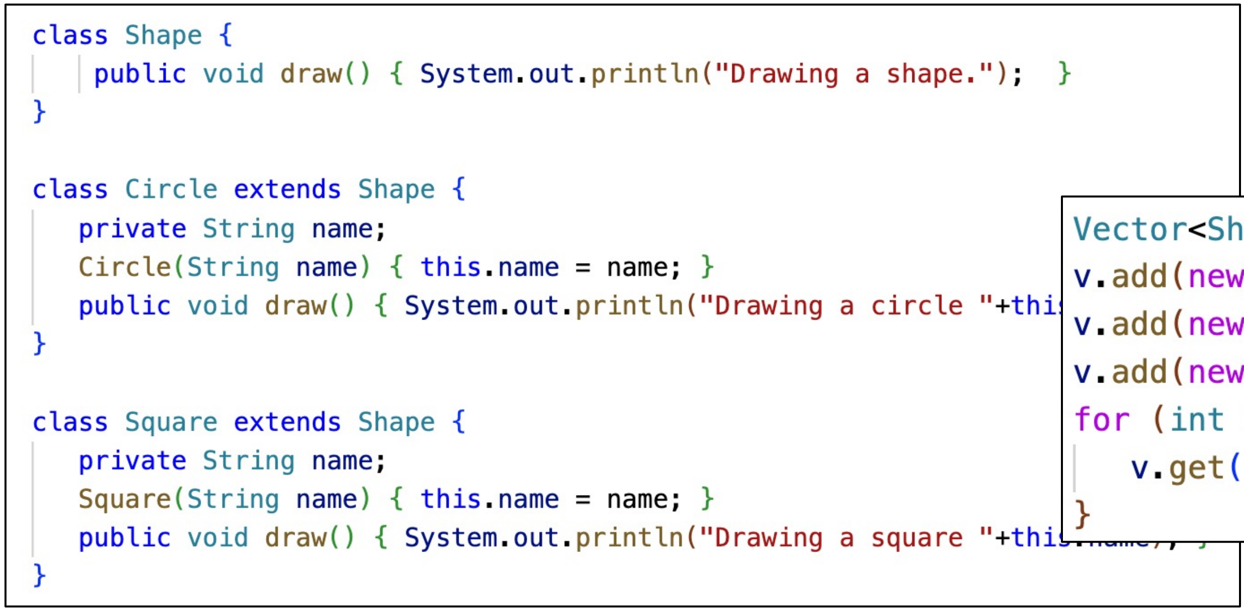
We say that v is a subtype polymorphic list/vector because the objects on the list/vector consist of different subtypes of Shape.

OOP – Subtype Polymorphism

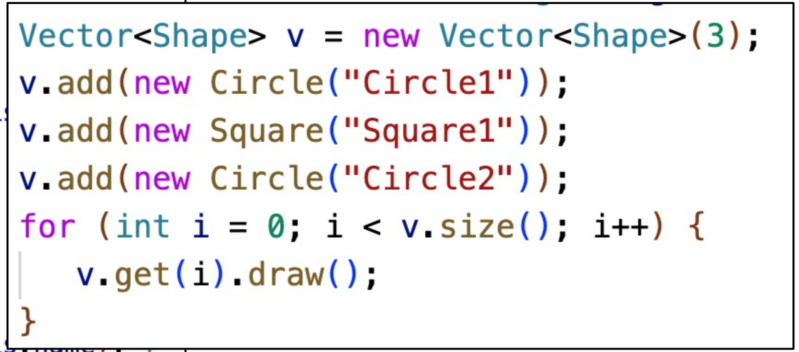
- Finally, dynamic dispatch makes this all work.
- In the code below we call the draw function on the base class Shape
- But what is actually called are the draw functions of the subtypes – dynamic dispatch



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

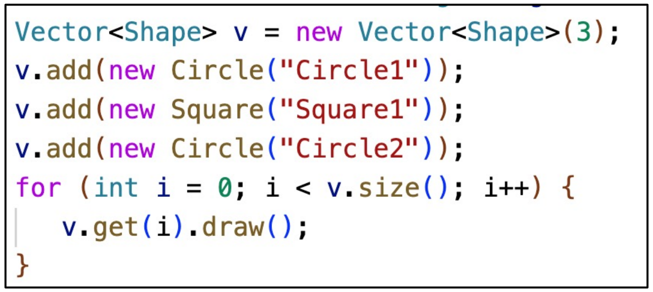


```java
class Shape {
    public void draw() { System.out.println("Drawing a shape.");  }
}

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

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

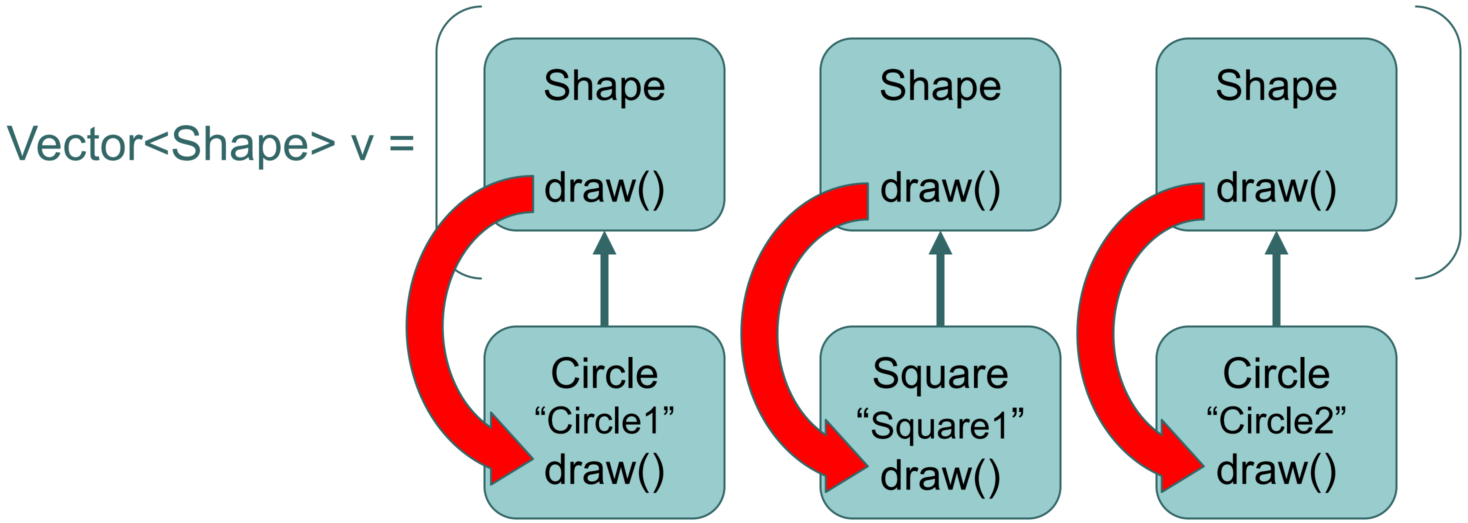


```java
Vector<Shape> v = new Vector<Shape>(3);
v.add(new Circle("Circle1"));
v.add(new Square("Square1"));
v.add(new Circle("Circle2"));
for (int i = 0; i < v.size(); i++) {
    v.get(i).draw();
}
```

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.