Mid-term test : what’s in it?

- Everything up to and including the last lecture
  1. Basic objects
  2. The this pointer
  3. Constructors
  4. Destructors
  5. References
  6. Const parameters
  7. Function overloading
  8. Operator overloading
  9. Assignment operator
  10. Templates
  11. Inheritance
  12. Virtual methods

- Things not covered (as you haven’t used it yet)
  - STL
  - Iterators
  - Extended access mod. (today)
  - Different types of polymorphism

Mid-term test : format?

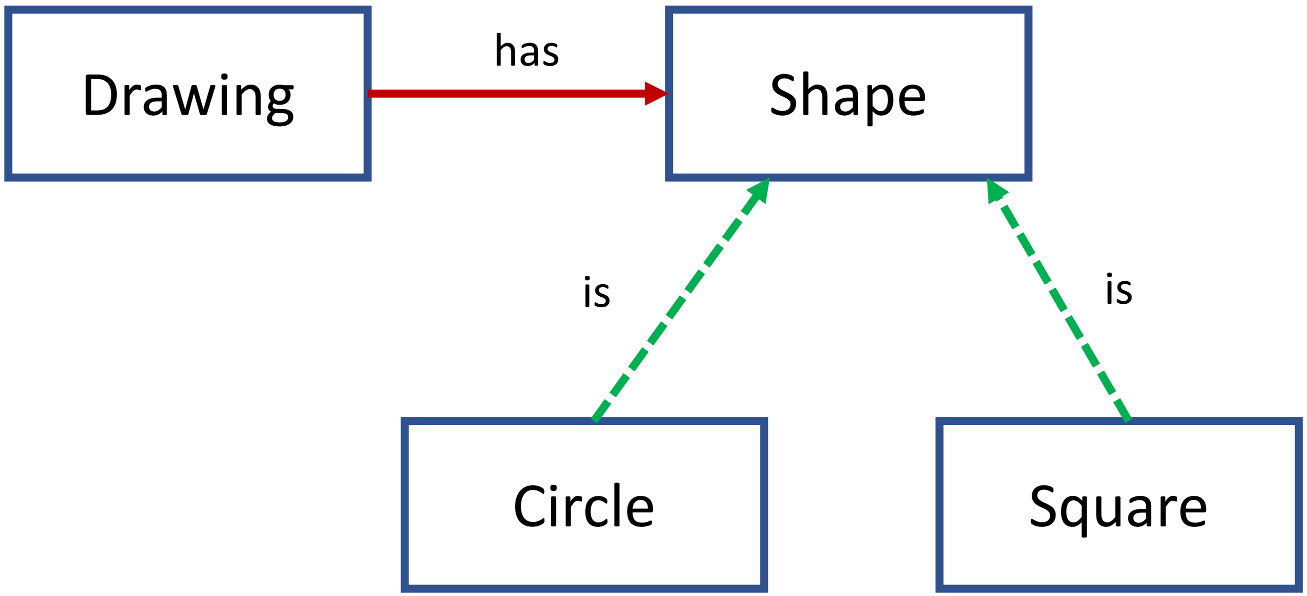
- Same as previous two tests
  - Multiple choice: 12 questions, 5 options per question
- What we are trying to assess:
  - Can you read the syntax?
  - Do you understand the concepts?
  - Are you able to analyse the code ***without*** running it?
- Weekly TBL questions remain a good guide
  - We draw from the same ideas pool for both
  - Mid-term questions/answers are just more heavily vetted

Portfolio : apologies again

- Q2 should have been turned round in 10 days
  - We’re now at more than twice that
- It is a priority, but keeps getting pre-empted
  - Mostly by teaching, not research

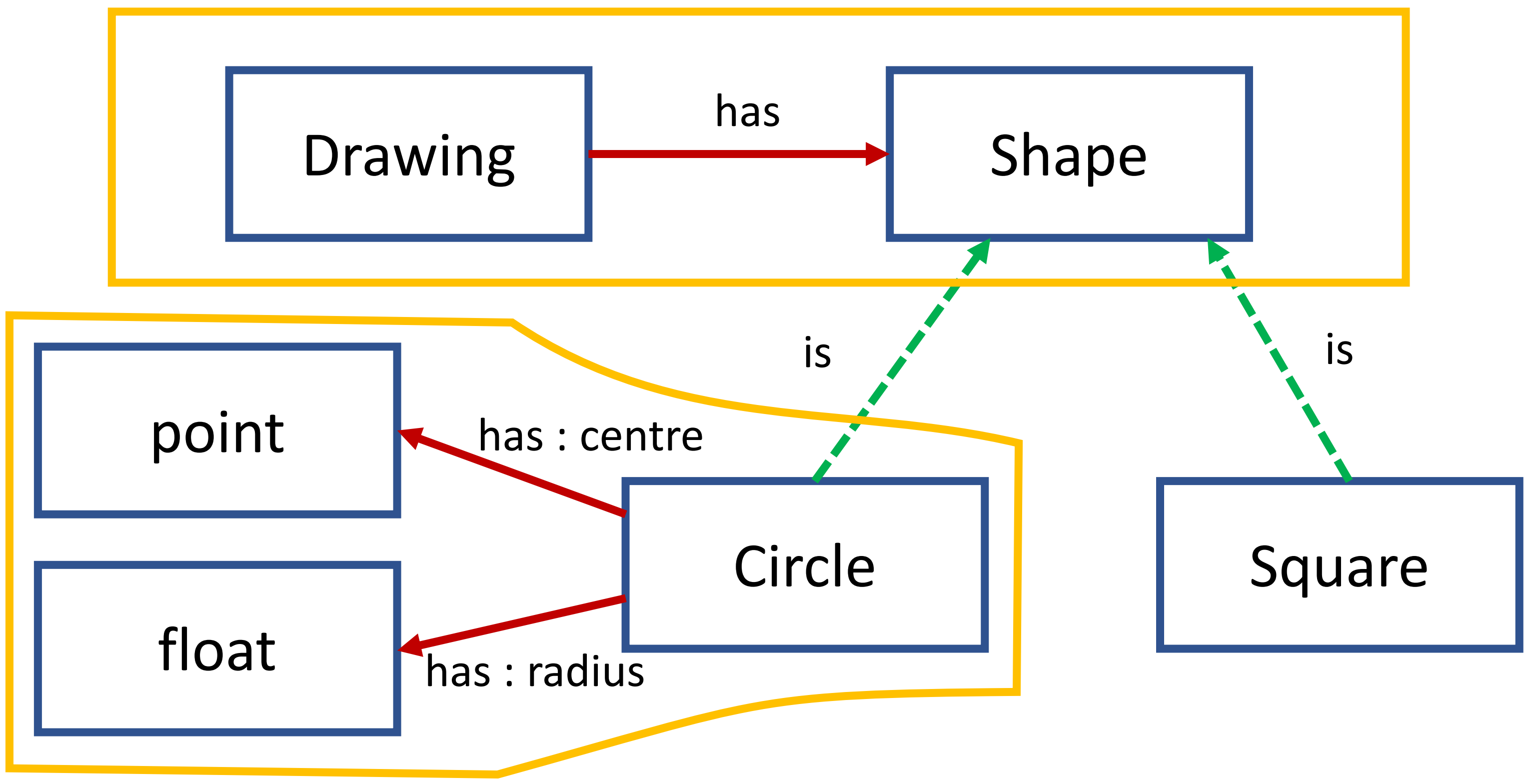
Inheritance and sub-types

Recap: Relationships between classes



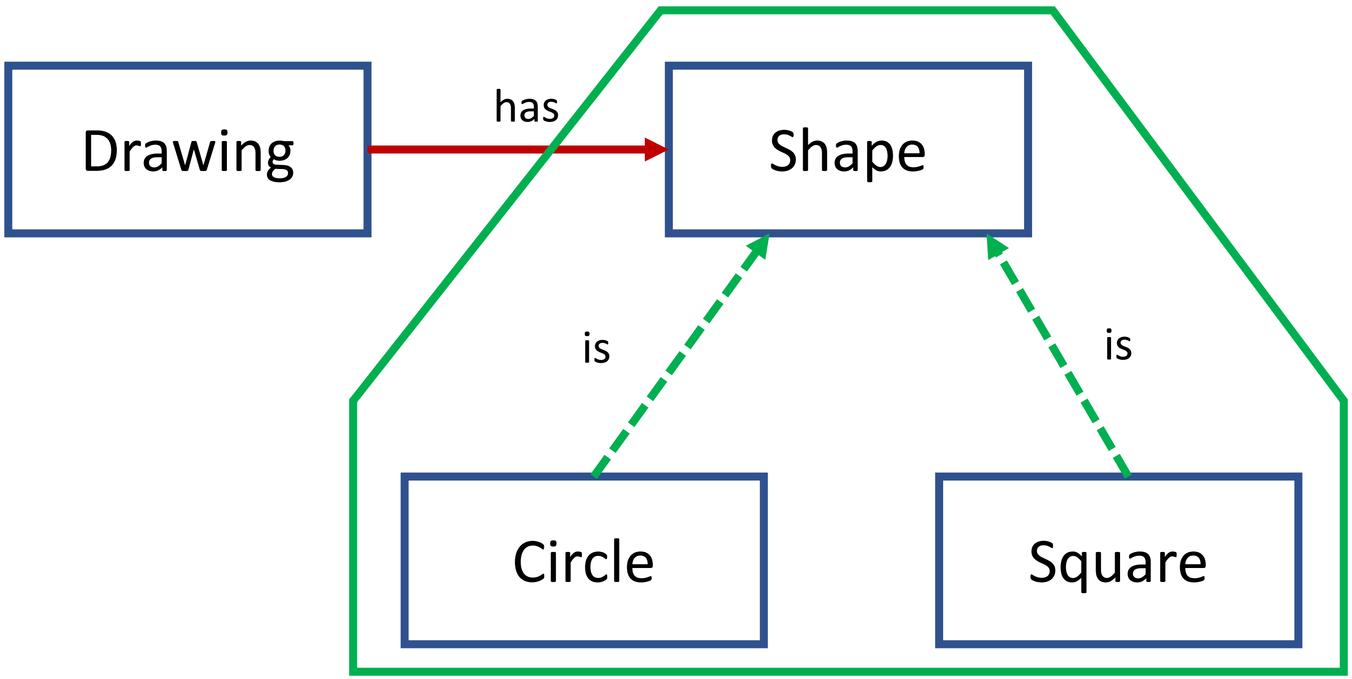
Recap: Relationships between classes

Aggregation : objects containing/owning other objects

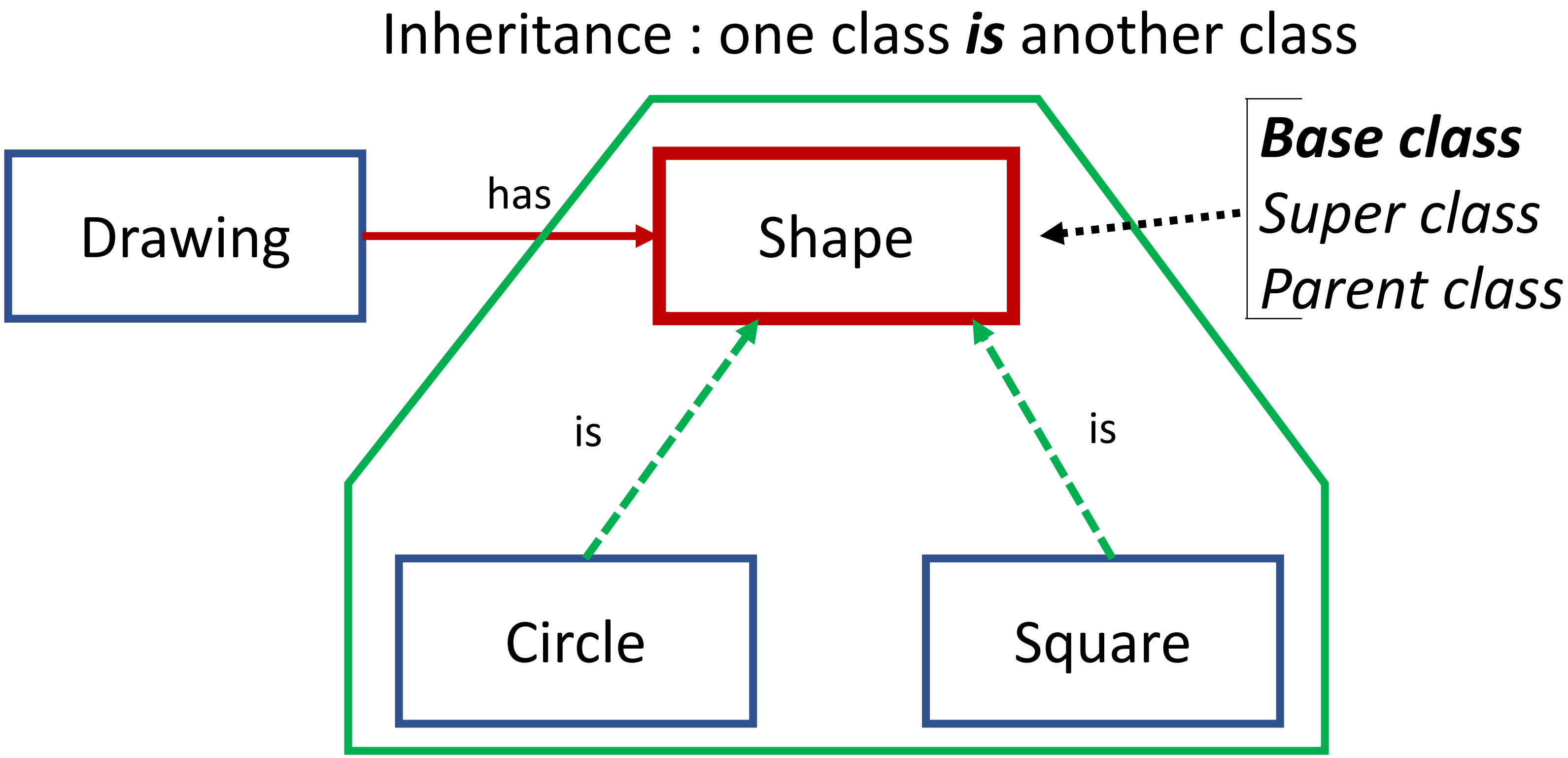


Recap: Relationships between classes

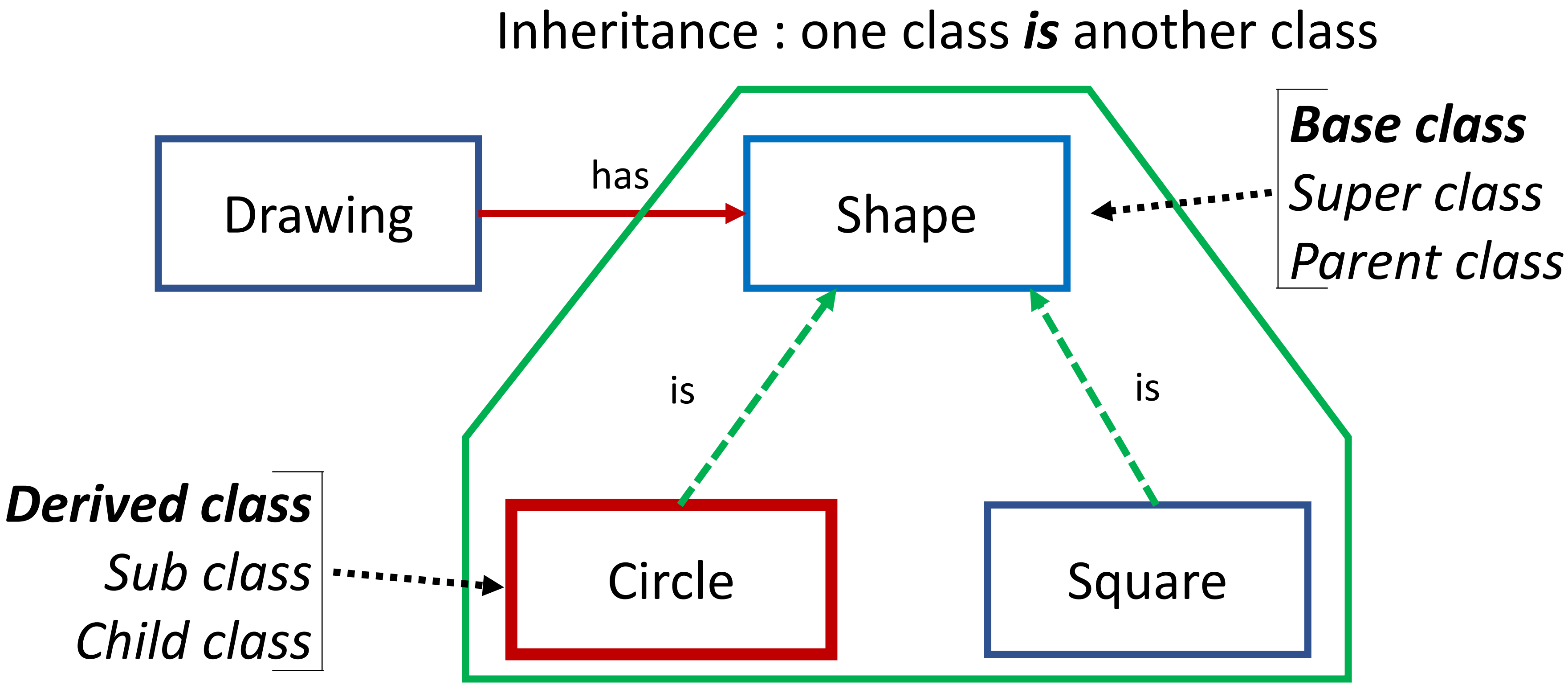
Inheritance : one class ***is*** another class



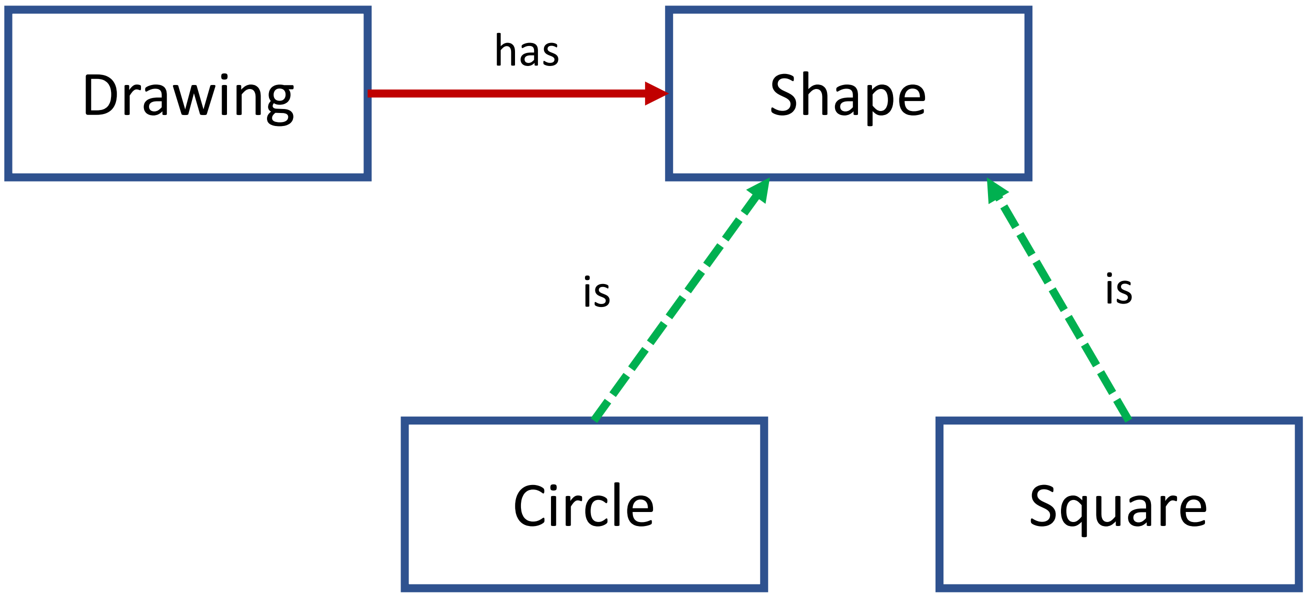
Recap: Relationships between classes



Recap: Relationships between classes

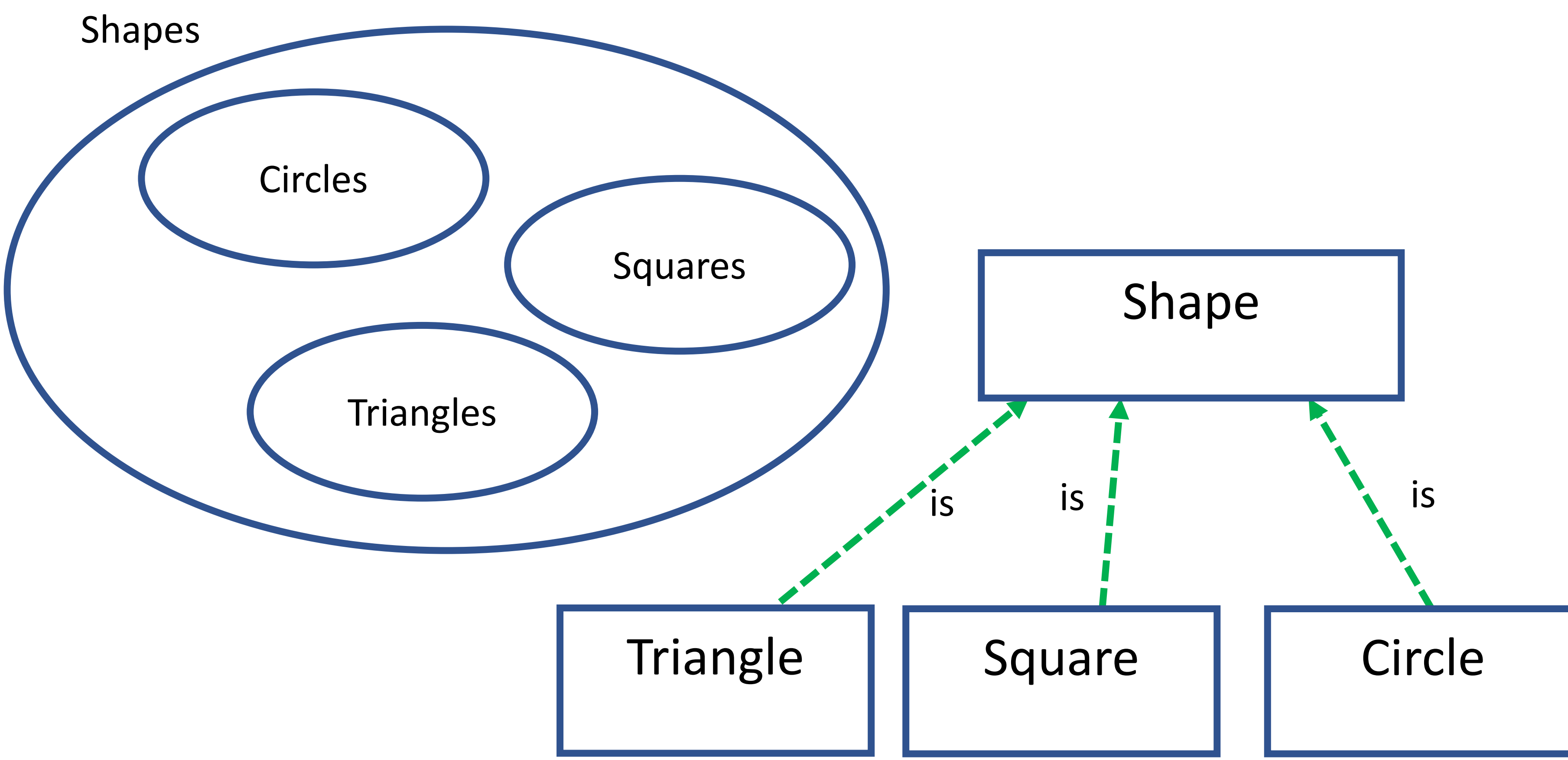


Recap: Relationships between classes



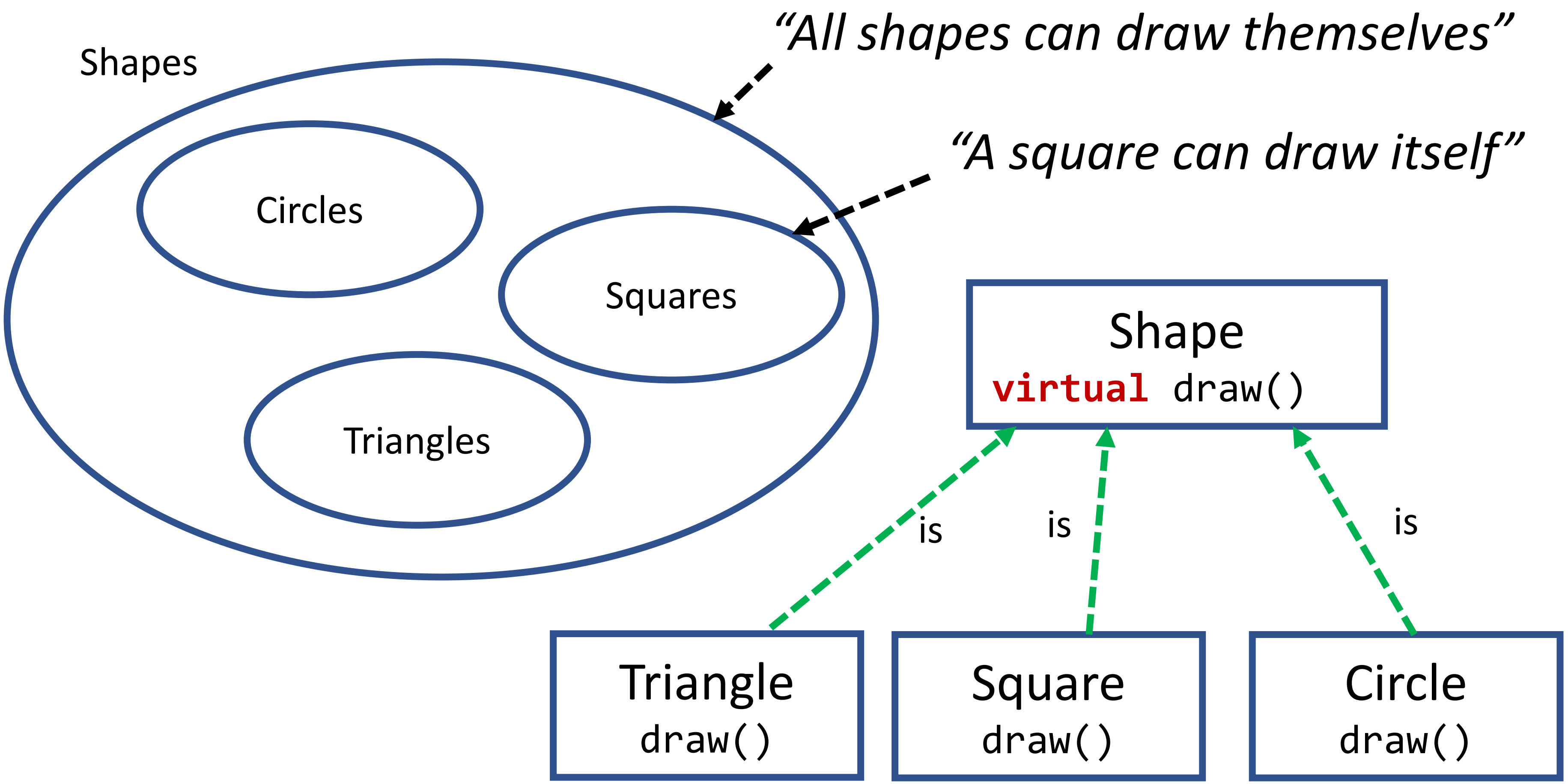
Inheritance and subtypes

New derived types can be added -> new sub-types



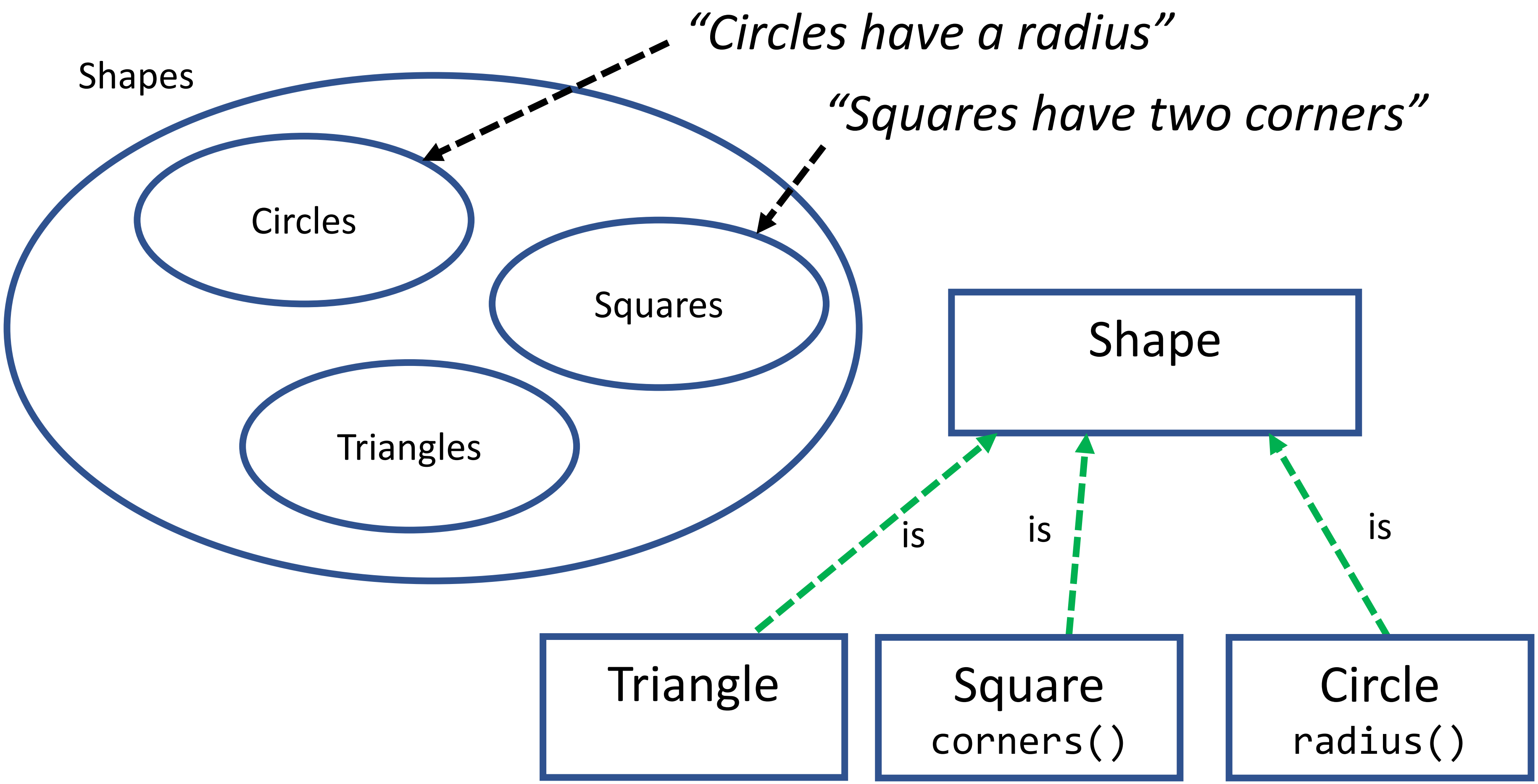
Inheritance and subtypes

Base classes should define common behaviour/functions



Inheritance and subtypes

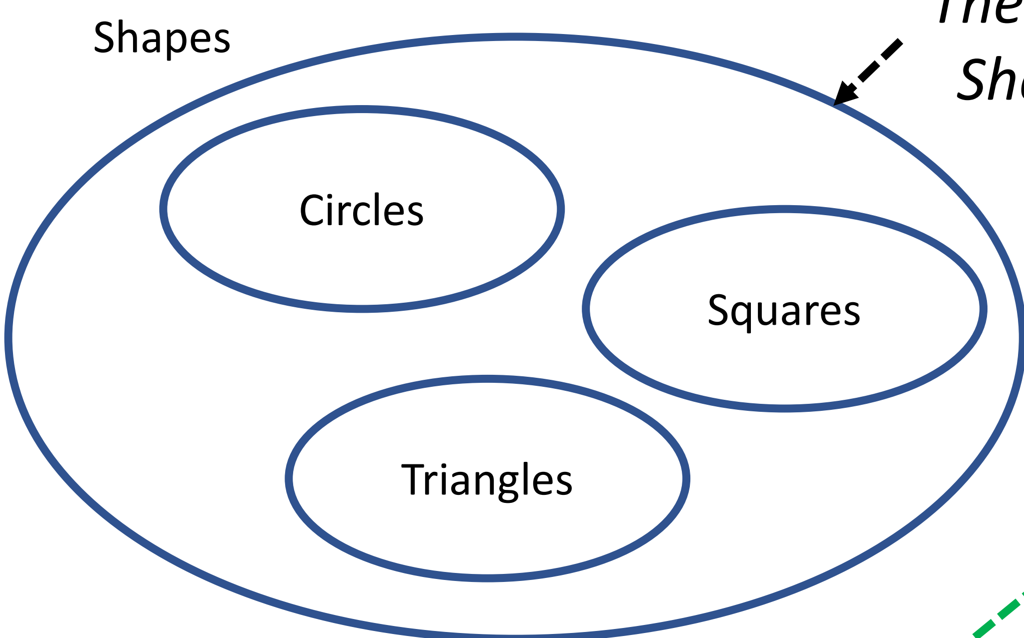
Derived types can have features that the base doesn't

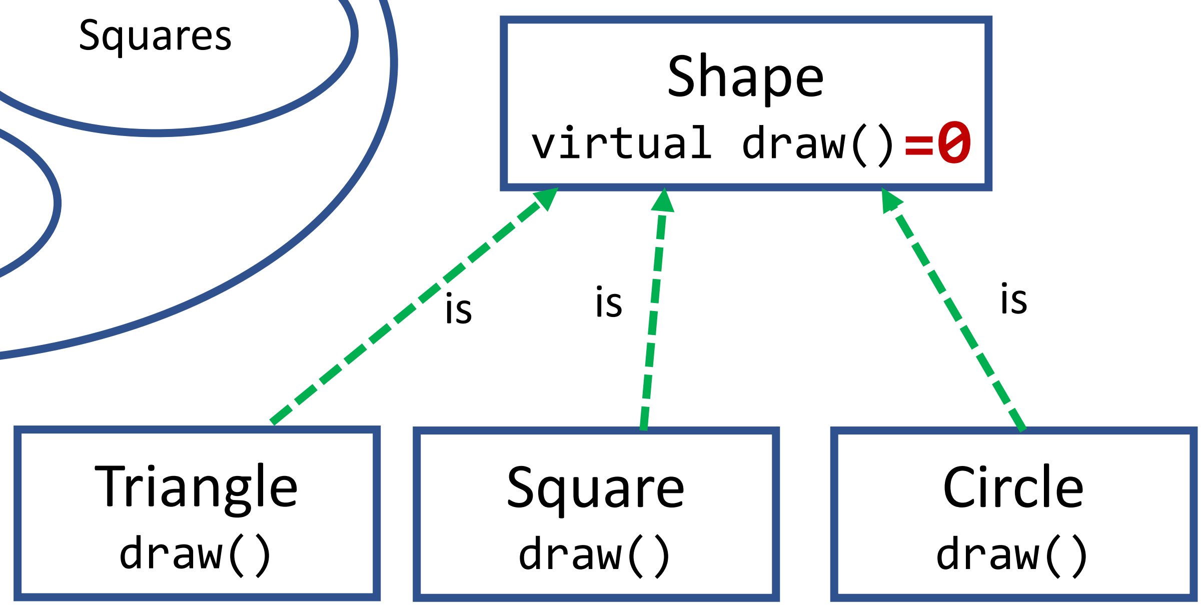


Inheritance and subtypes

Abstract base types : can only create derived types

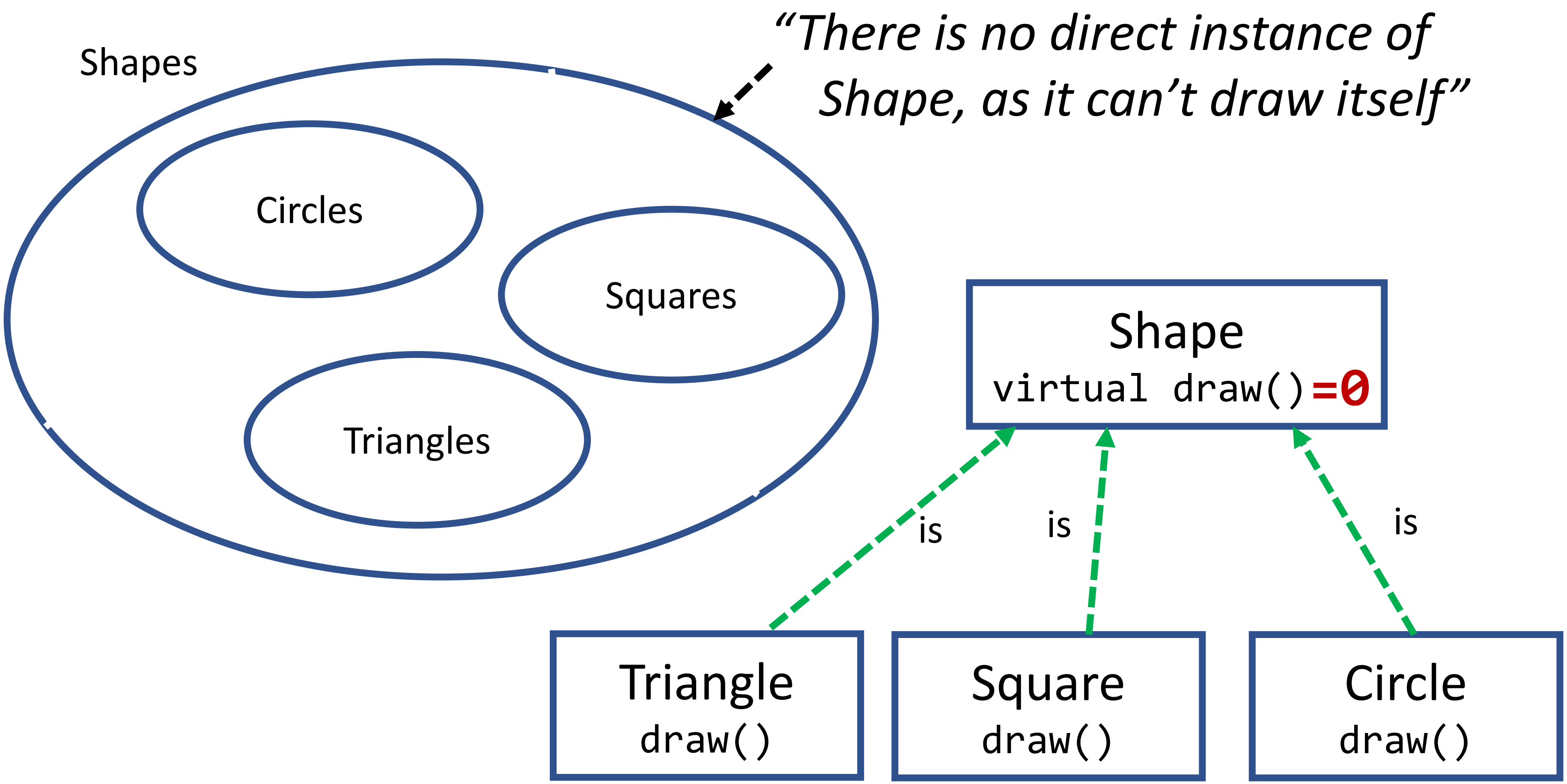
"There is no direct instance of Shape, as it can't draw itself"





Inheritance and subtypes

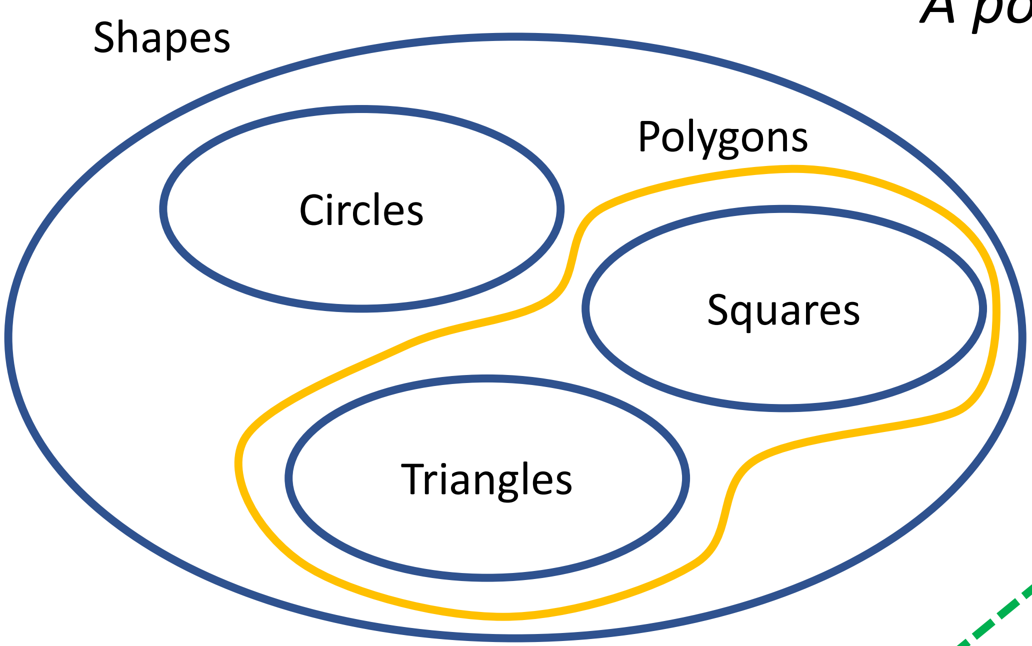
Abstract base types : can only create derived types

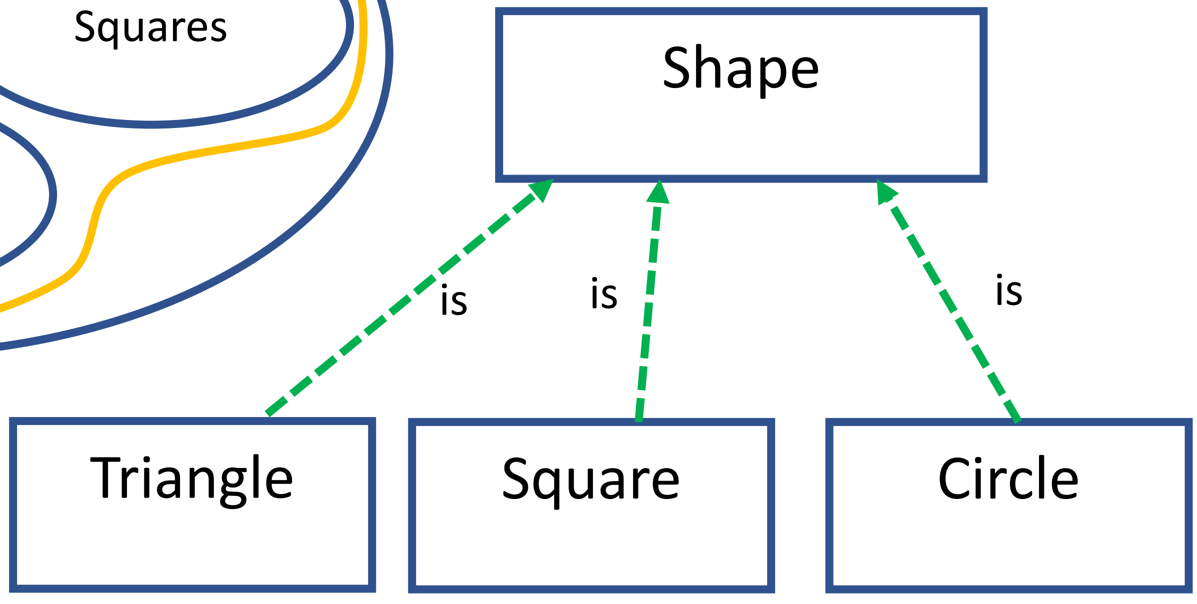


Inheritance and subtypes

Some classes are both base and derived classes

"A polygon is a shape with vertices"

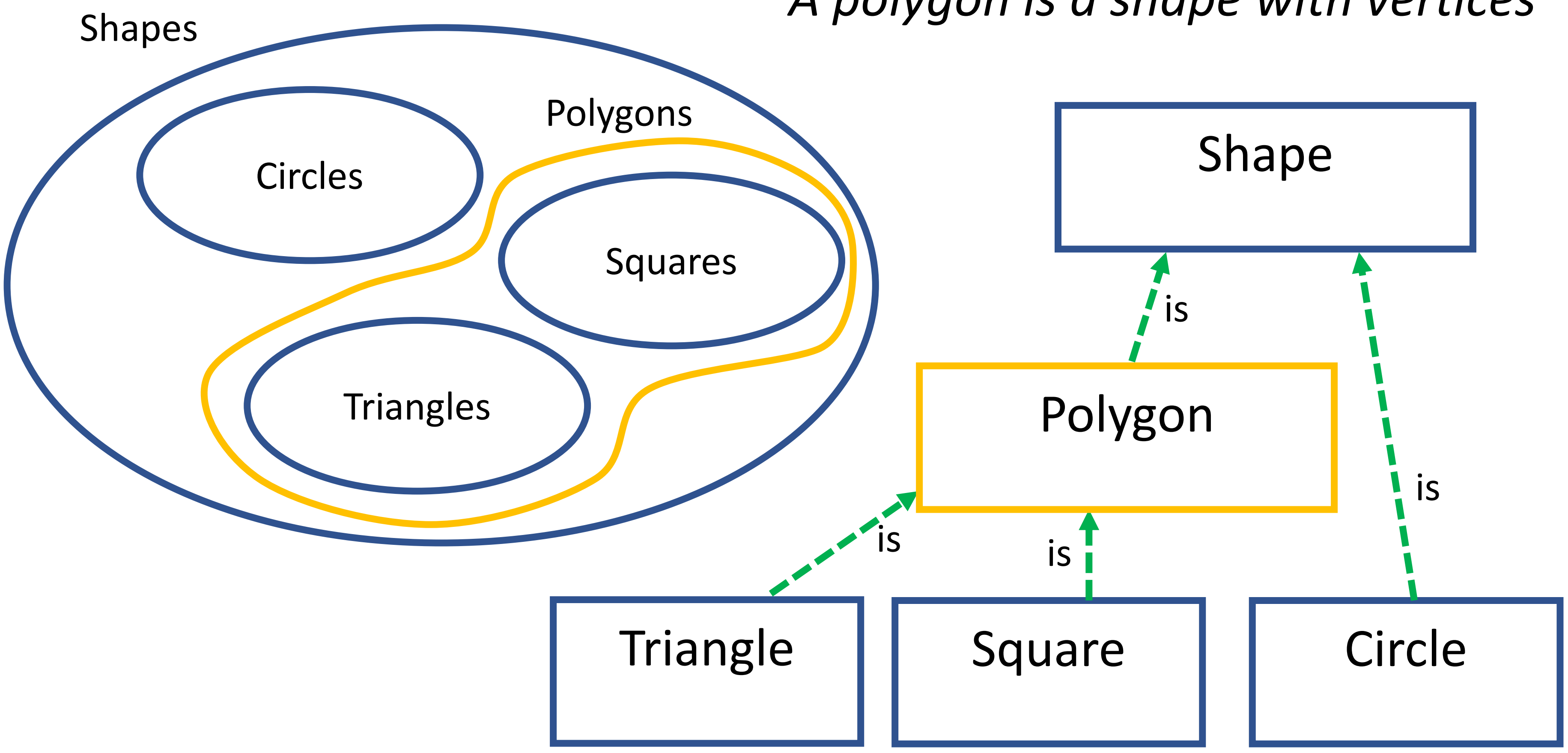




Inheritance and subtypes

Some classes are both base and derived classes

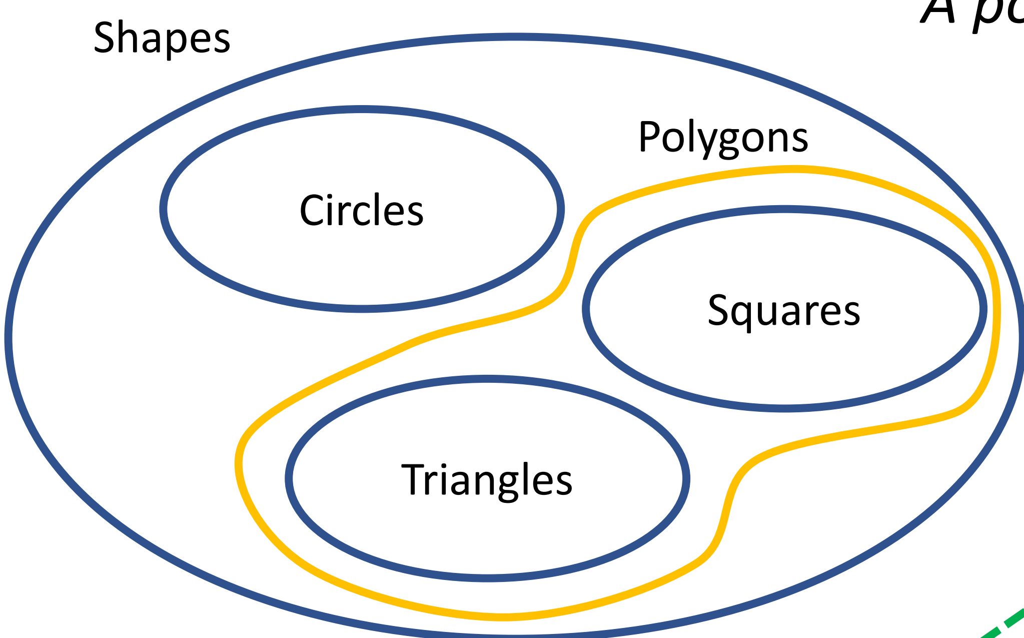
"A polygon is a shape with vertices"

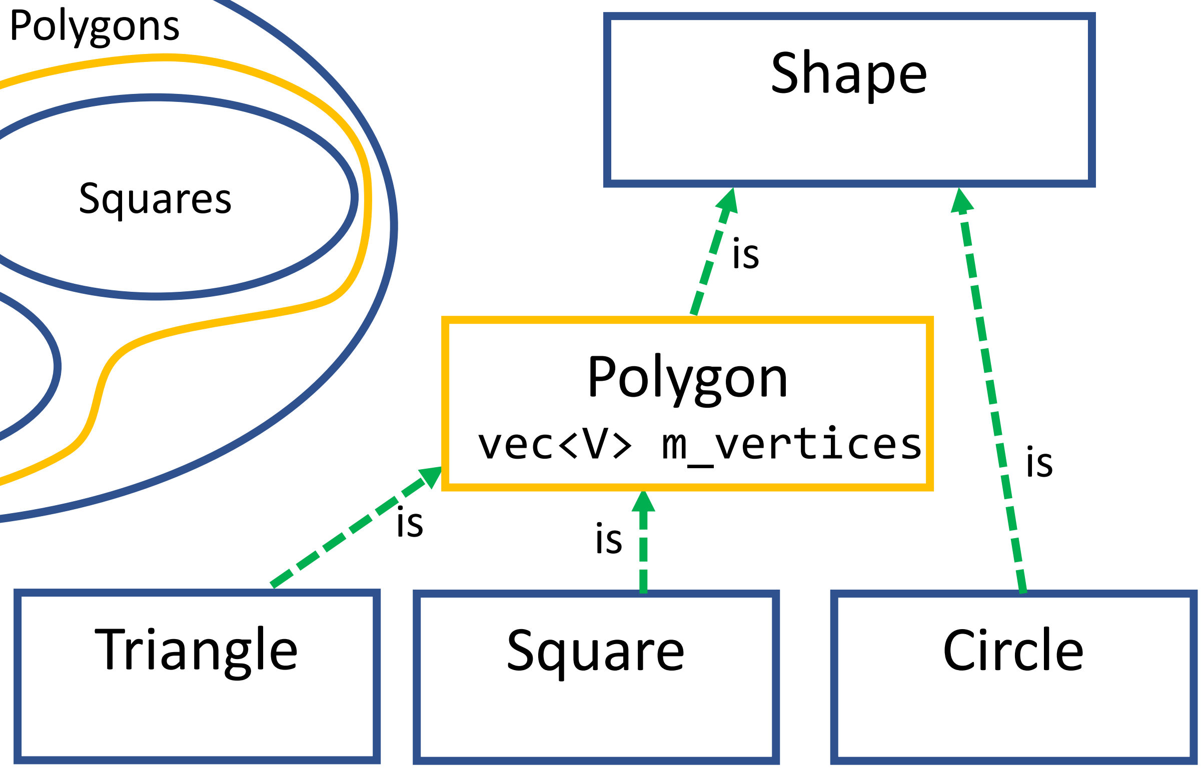


Inheritance and subtypes

Inheritance might mean derived classes inherit ***data***

“A polygon **has a** *vector of vertices”*

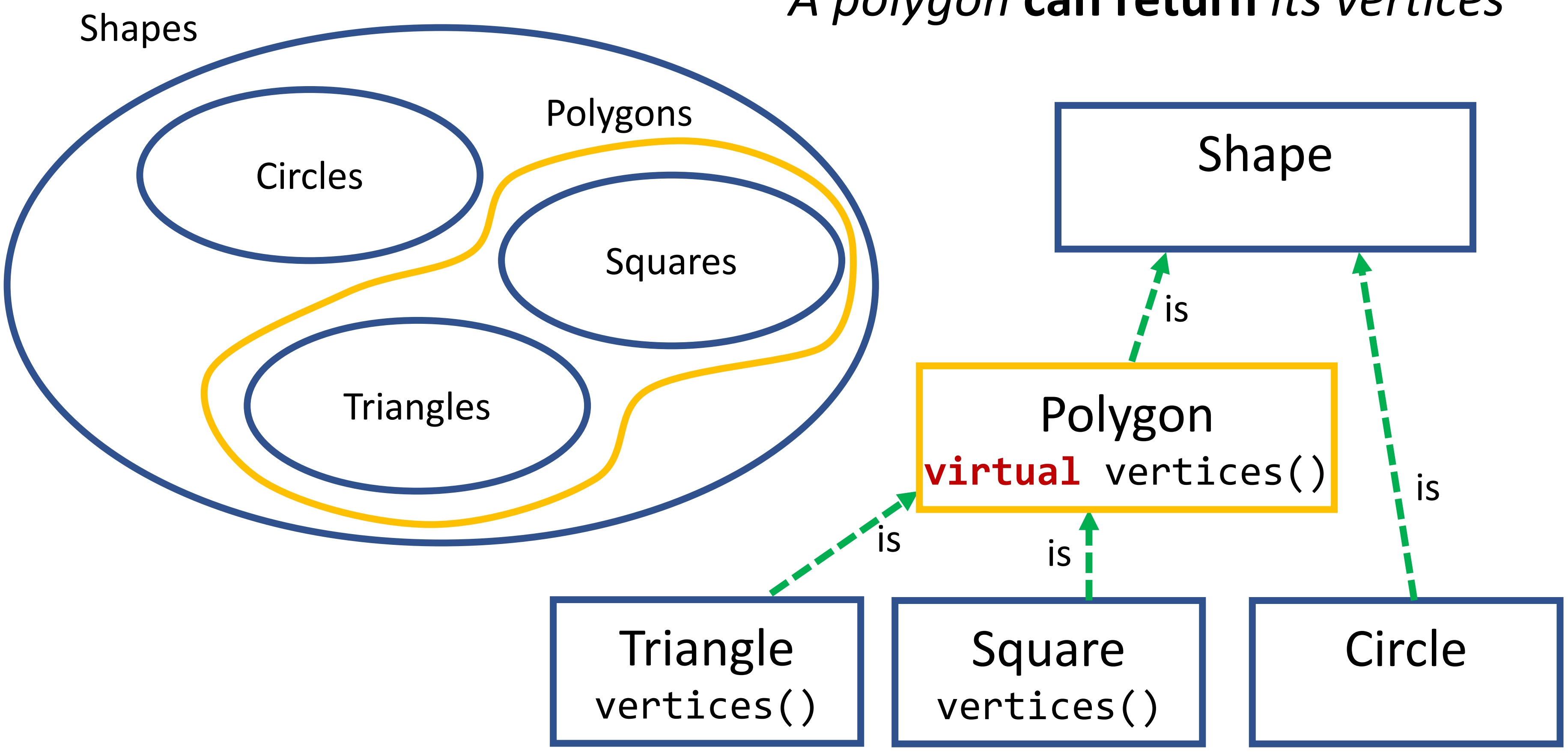




Inheritance and subtypes

Inheritance might mean derived classes inherit ***behaviour***

"A polygon **can return** *its vertices"*



Inheritance and subtypes

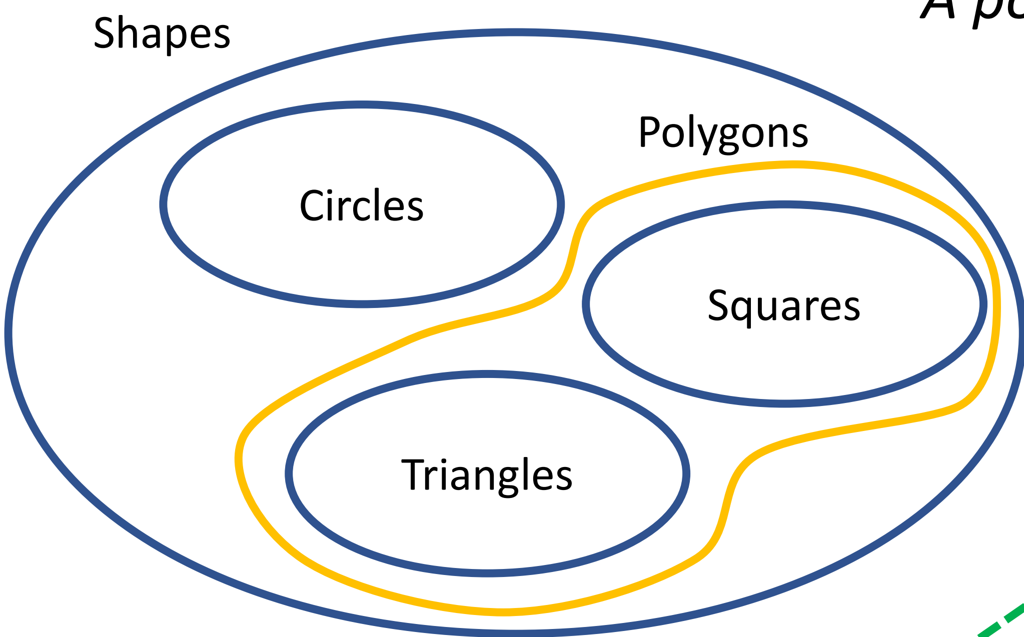
- Base classes declare common state and function
  - Any member data on base is something all derived classes ***have***
  - Any method on base is something all derived classes ***can do***
  - Methods on base can be { non-virtual, virtual, pure-virtual }
    - *Non-virtual*: derived classes cannot change meaning of function
    - *Virtual*: derived classes can choose to change meaning of function
    - *Pure-virtual*: concrete derived classes must add meaning of function
- Derived classes refine or extend the base classes
  - Add member data that only the derived class needs
  - Add new methods that only make sense for derived class
  - Override virtual methods from the base class

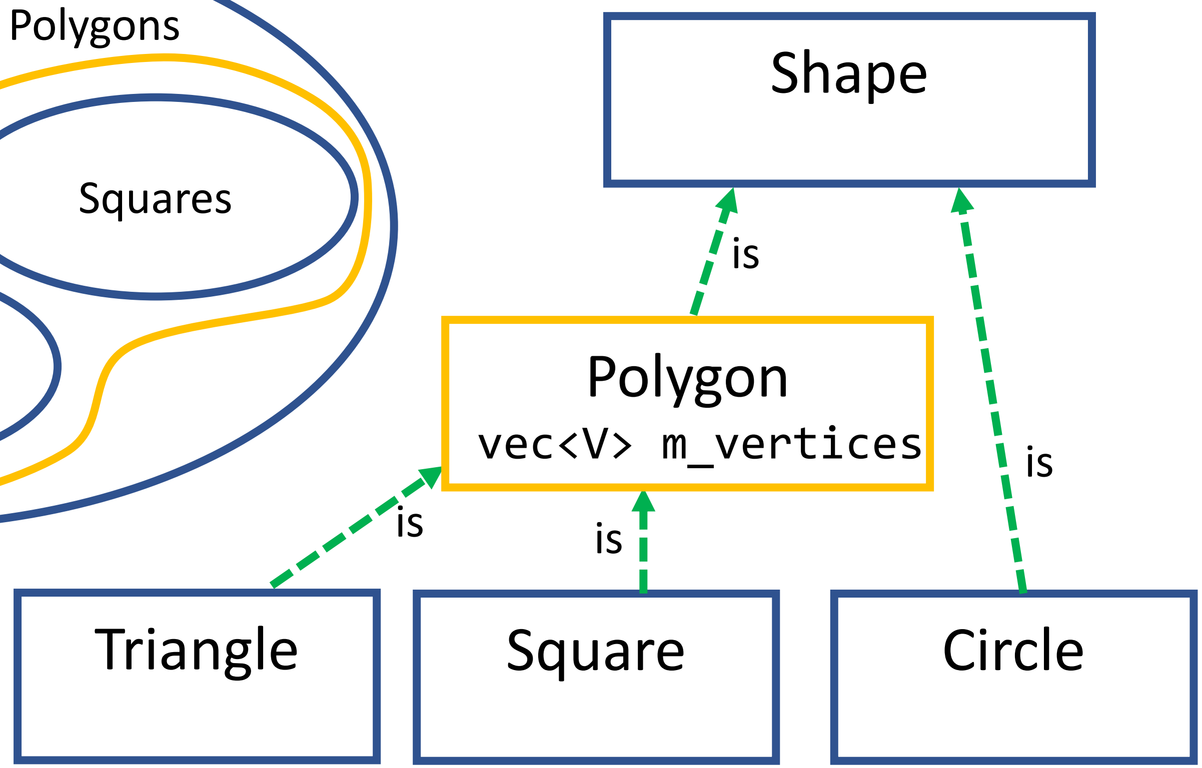
Managing access to members

Inheritance and subtypes

Inheritance might mean derived classes inherit ***data***

"A polygon **has a** *vector of vertices"*





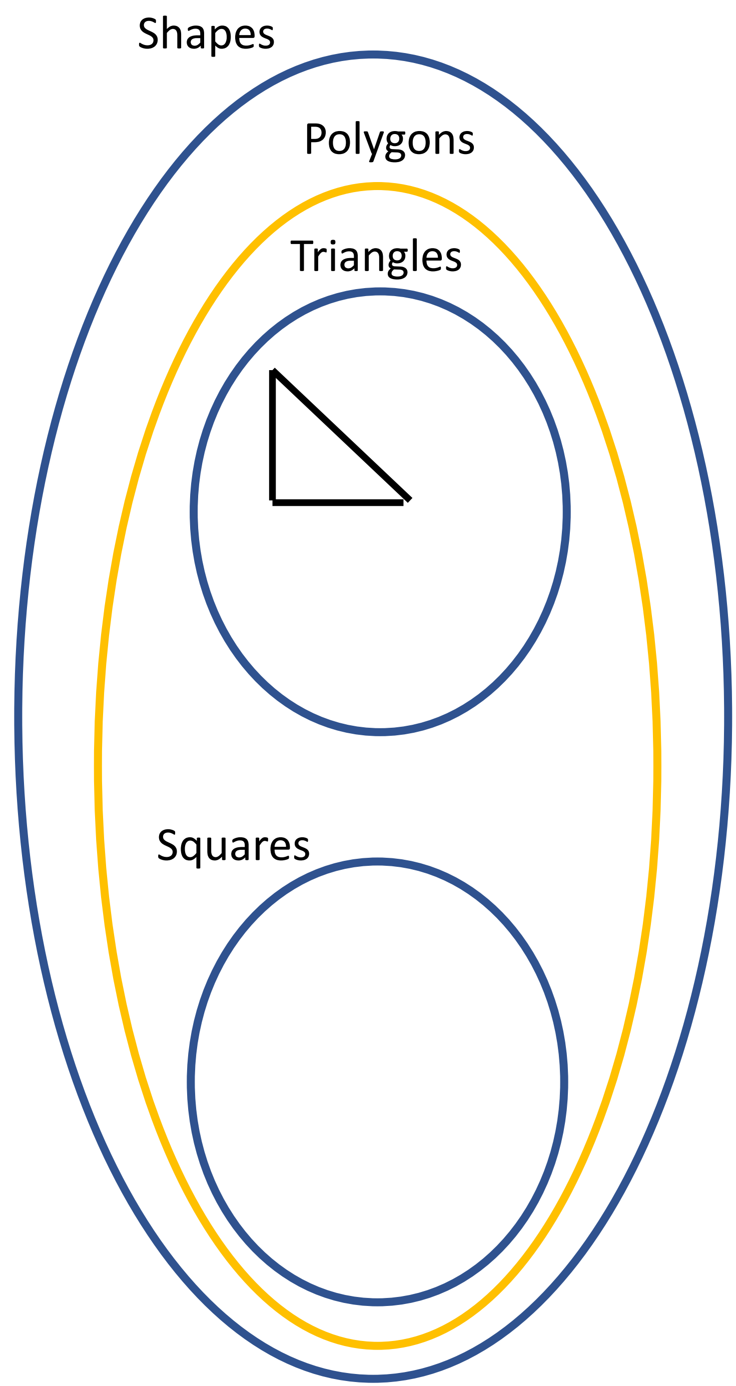
```cpp
class Shape
{
public:
    virtual ~Shape()
    {}

    virtual void draw(ostream &dst) const =0 ;
};

class Polygon
    : public Shape
{
public:
    vector<point> vertices;

    virtual void draw(ostream &dst);
};
```

```
class Shape
{
public:
    virtual ~Shape()
    {}

    virtual void draw(ostream &dst) const =0 ;
};

class Polygon
    : public Shape
{
public:
    vector<point> vertices;

    virtual void draw(ostream &dst)
    {
        dst<<"<polyline points='"
        for(unsigned i=0; i<vertices.size(); i++){
            dst<<" "<<vertices[i].x<<","<<vertices[i].y;
        }
        dst<<"' />"
    }
};
```

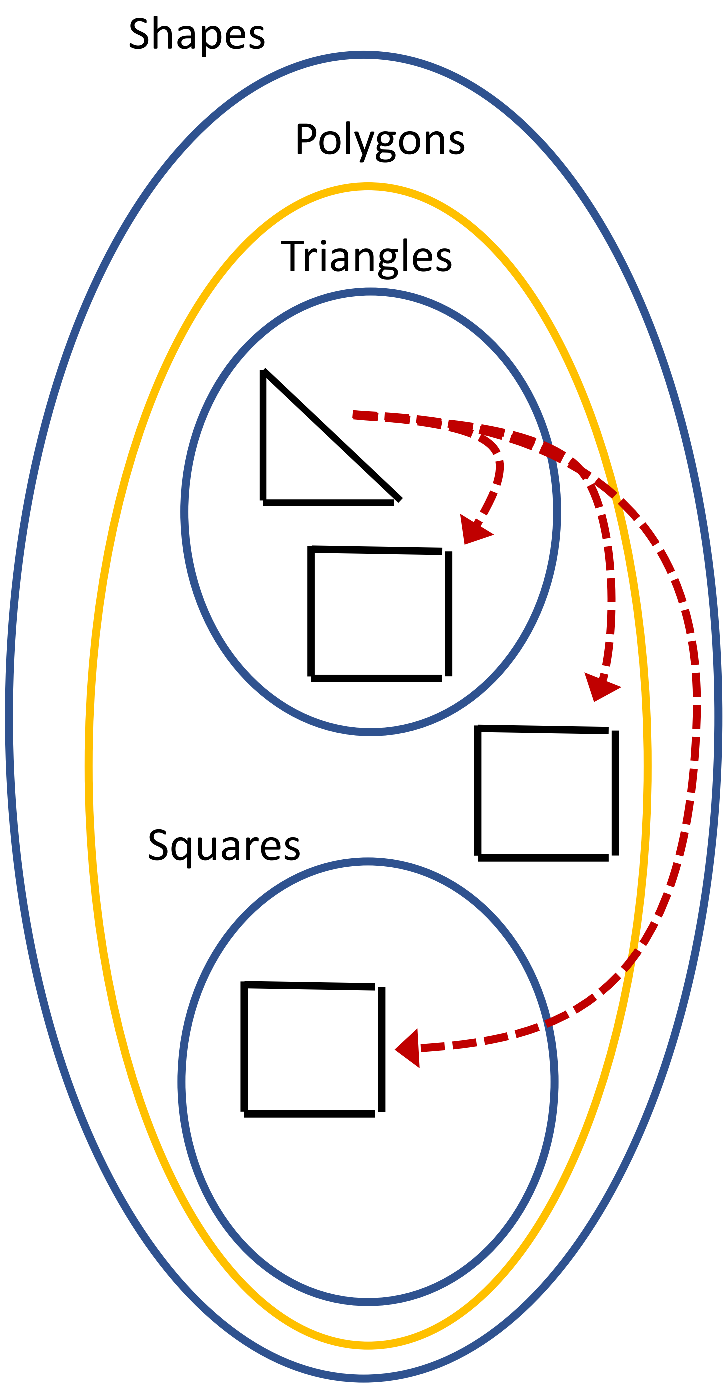
```cpp
class Polygon
    : public Shape
{
public:
    vector<point> vertices;
    virtual void draw(ostream &dst);
};


class Triangle
    : public Polygon
{
public:
    Triangle(point p1, point p2, point p3)
    {
        vertices={p1,p2,p3};
    }
};
```

```
class Polygon
    : public Shape
{
public:
    vector<point> vertices;
    virtual void draw(ostream &dst);
};


class Square
    : public Polygon
{
public:
    Square(point pos, float size)
    {
        vertices={
            pos+{+size/2,+size/2}, pos+{+size/2,-size/2},
            pos+{-size/2,-size/2}, pos+{-size/2,+size/2}
        };
    }
};
```

```
class Polygon
    : public Shape
{
public:
    vector<point> vertices;
    virtual void draw(ostream &dst);
};


class Triangle
    : public Polygon
{
public:
    Triangle(point p1, point p2, point p3)
    {
        vertices={p1,p2,p3};
    }
};

int main()
{
    Triangle tri{ {0,0}, {0,1}, {1,0} };


}
```



```cpp
class Polygon
    : public Shape
{
public:
    vector<point> vertices;
    virtual void draw(ostream &dst);
};


class Triangle
    : public Polygon
{
public:
    Triangle(point p1, point p2, point p3)
    {
        vertices={p1,p2,p3};
    }
};

int main()
{
    Triangle tri{ {0,0}, {0,1}, {1,0} };

    tri.vertices.push_back( {1,1} );
}
```



An object's class cannot change

- An instance's type is fixed at construction time
  - The type is determined by the constructor used
  - `Triangle::Triangle(…)` → always a `Triangle`
  - There is always one single most-derived "leaf" type
- An instance can also be accessed through base types
  - A `Triangle` is-a `Polygon` → it can be treated as a `Polygon`
  - A `Triangle` is-a `Shape` → it can be treated as a `Shape`
  - But its true dynamic type is always `Triangle`
- The API should ensure logical type matches C++ type
  - The set of "triangles" all have three vertices
  - So `Triangle` class must ensure there are only three vertices

Maintaining class invariants

- We want to exploit shared features of Polygon
  - A user should be able to get the vertices of any polygon
- Our problem: users of the class can modify vertices
  - But: we must ensure there are only three vertices

- Option 1: change inheritance from data to method
  - ~~Polygon has vertices~~
  - Polygon can provide vertices on demand
- Option 2: control access to the member variables
  - Polygon has vertices, but ensure they can only be read

Inheritance : data -> method

Change “vertices” from data to a method

```
class Polygon
    : public Shape
{
public:
    vector<point> vertices;

    virtual void draw(ostream &dst)
    {
        dst<<"<polyline points='"
        for(unsigned i=0; i<vertices.size(); i++){
            dst<<" "<<vertices[i].x<<","<<vertices[i].y;
        }
        dst<<"' />";
    }
};
```

Inheritance : data -> method

Change “vertices” from data to a method

```
class Polygon
    : public Shape
{
public:
    virtual vector<point> vertices() const =0;

    virtual void draw(ostream &dst)
    {
        dst<<"<polyline points='"
        for(unsigned i=0; i<vertices.size(); i++){
            dst<<" "<<vertices[i].x<<","<<vertices[i].y;
        }
        dst<<"' />";
    }
};
```

Inheritance : data -> method

Ask derived class to create vertices on demand

```
class Polygon
    : public Shape
{
public:
    virtual vector<point> vertices() const =0;

    virtual void draw(ostream &dst)
    {
        dst<<"<polyline points='"
        for(unsigned i=0; i<vertices().size(); i++){
            dst<<" "<<vertices()[i].x<<","<<vertices()[i].y;
        }
        dst<<"' />";
    }
};
```

Inheritance : data -> method

For efficiency : generate vertices once and re-use

```
class Polygon
    : public Shape
{
public:
    virtual vector<point> vertices() const =0;

    virtual void draw(ostream &dst)
    {
        vector<point> l_vertices=vertices();
        dst<<"<polyline points='"
        for(unsigned i=0; i<l_vertices.size(); i++){
            dst<<" "<<l_vertices[i].x<<","<<l_vertices[i].y;
        }
        dst<<"' />";
    }
};
```

Inheritance : data -> method

Derived classes use type-specific data members

```
class Triangle
    : public Polygon
{
private:
    point2d m_p1, m_p2, m_p3;
public:
    Triangle(point2d p1, point2d p2, point2d p3)
    {
        m_p1=p1;  m_p2=p2;  m_p3=p3;
    }
};
```

Inheritance : data -> method

Derived classes override behaviour of vertices

```
class Triangle
    : public Polygon
{
private:
    point2d m_p1, m_p2, m_p3;
public:
    Triangle(point2d p1, point2d p2, point2d p3)
    {
        m_p1=p1;  m_p2=p2;  m_p3=p3;
    }

    vector<point> vertices() const
    {
        return {m_p1, m_p2, m_p3};
    }
};
```

Inheritance : data -> method

```
class Square
    : public Polygon
{
private:
    point m_pos;
    float m_size;
public:
    Square(point pos, float size)
    { m_pos=pos;  m_size=size; }

};
```

Inheritance : data -> method

```
class Square
    : public Polygon
{
private:
    point m_pos;
    float m_size;
public:
    Square(point pos, float size)
    { m_pos=pos;  m_size=size; }

    virtual vector<point> vertices() const
    {
      return {
        m_pos+{+m_size/2,+m_size/2}, m_pos+{+m_size/2,-m_size/2},
        m_pos+{-m_size/2,-m_size/2}, m_pos+{-m_size/2,+m_size/2}
      };
    }
};
```

Inheritance : data -> method

- Using inheritance of methods provides freedom
    - Defining what a type can ***do***, rather than what it ***contains***
- *Usually* class APIs should be defined using methods
    - Base classes define methods ***all*** derived classes support
    - The implementation is hidden from users
    - The implementation can be changed without breaking users
- *Usually* data inheritance is about implementation
    - An internal detail, shared between base and derived class

Maintaining class invariants

- We want to exploit shared features of Polygon
  - A user should be able to get the vertices of any polygon
- Our problem: users of the class can modify vertices
  - But: we must ensure there are only three vertices

- Option 1: change inheritance from data to method
  - ~~Polygon has vertices~~
  - Polygon can provide vertices on demand
- Option 2: control access to the member variables
  - Polygon has vertices, but ensure they can only be read

Controlling access to data members

There is a run-time cost for re-generating vertices

A new vector needs to be constructed and filled in whenever we access vertices

```
class Polygon
    : public Shape
{
public:
    virtual vector<point> vertices() const =0;

    virtual void draw(ostream &dst)
    {
        vector<point> l_vertices=vertices();
        dst<<"<polyline points='"
        for(unsigned i=0; i<l_vertices.size(); i++){
            dst<<" "<<l_vertices[i].x<<","<<l_vertices[i].y;
        }
        dst<<"' />";
    }
};
```

Controlling access to data members

There is a flexibility cost due to lack of direct access

```
class Polygon
    : public Shape
{
public:
    vector<point> vertices;

    void translate(float dx, float dy)
    {
        for(int i=0; i<vertices.size(); i++){
            vertices[i] += point{dx,dy};
        }
    }
};
```

One implementation of translate works for all classes derived from Polygon

Only works if Polygon can both read and write the vertices

Controlling access to data members

There is a flexibility cost due to lack of direct access

```
class Polygon
    : public Shape
{
public:
    virtual vector<point> vertices() const =0;

    virtual void translate(float dx, float dy) =0;
};
```

Polygon class cannot write to the vertex vector, as they may be separate points

Every derived class must provide a custom override of translate functionality

Controlling access to data members

We could make the vertices member private

```
class Polygon
    : public Shape
{
private:
    vector<point> m_vertices;
public:
    const vector<point> &vertices() const
    { return m_vertices; }

    void translate(float dx, float dy)
    {
        for(int i=0; i<m_vertices.size(); i++){
            m_vertices[i] += point{dx,dy};
        }
    }
};
```

Users cannot directly access data members;
They can no longer turn triangles into squares

Can return const reference to member;
No more copying of data

Can use member variable directly within class

Controlling access to data members

But private members are not accessible to derived class

```
class Polygon : public Shape
{
private:
    vector<point> m_vertices;
public:
    // ...
};

class Triangle : public Polygon
{
public:
    Triangle(point2d p1, point2d p2, point2d p3)
    {
        m_vertices={p1,p2,p3};
    }
};
```

Compilation error: Triangle cannot access private member of Polygon

The protected access modifier

- Access modifiers change accessibility of class members
  - Apply to both member variables and methods

- Our access modifiers:
  - `public` : can be accessed from anywhere
  - `private` : only accessible inside class
  - `protected` : accessible inside class ***and*** derived classes

- Protected members provide safe access to internal details
  - *Derived classes can be trusted (a bit)* : they are on the inside
  - *Don't trust code outside the inheritance hierarchy* : we keep our secrets

Controlling access to data members

Can set protected member variables in constructor

```
class Polygon : public Shape
{
protected:
    vector<point> m_vertices;
public:
    // ...
};

class Triangle : public Polygon
{
public:
    Triangle(point2d p1, point2d p2, point2d p3)
    {
        m_vertices={p1,p2,p3};
    }
};
```

Protect data member can be accessed in derived classes

Triangle is derived from Polygon; So Triangle can access m_vertices

Why use sub-types?

Why have Triangle and Square?

- Why not do everything in Polygon?
  - It can draw itself
  - It can translate itself
  - You can get the corners/vertices
  - Anything you can do to a Triangle you can do to a Polygon

- The more specialised sub-classes could offer:
  - Extra functionality not available in base (though not for Poly)
  - More efficient storage/representation
  - More efficient implementation of behaviour

Special-casing functions

We can add extra functions on the base

```
class Shape
{
public:
    virtual ~Shape()
    {}

    virtual void draw(ostream &dst) const =0 ;

    virtual float area() const=0;
};
```

Special-casing functions

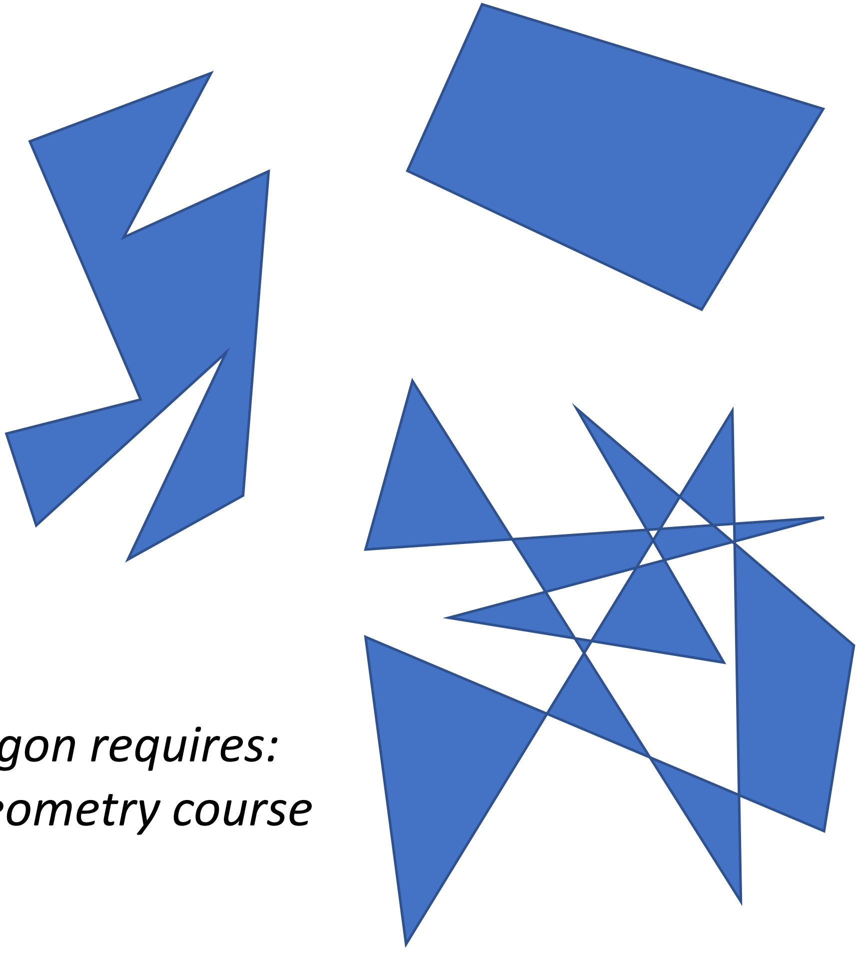
Then implement in derived classes

```
class Circle : public Shape
{
private:
    point m_centre;
    float m_radius;
public:
    float area() const
    {
        const float PI = 3.14159265358;
        return PI * m_radius * m_radius;
    }
};
```

Special-casing functions

General cases are often complex and slow

```
class Polygon : public Shape
{
private:
    vector<point> m_vertices;
public:
    float area() const
    {
        // TODO
    }
};
```



Calculating the area of an arbitrary polygon requires:
- An entire lecture of a computational geometry course
- Multiple auxiliary data-structures
- A few-hundred lines of code

Special-casing functions

Special cases can be fast and efficient

```
class Triangle : public Polygon
{
public:
    float area() const
    {
      assert(m_vertices.size()==3);
      point a=m_vertices[0], b=m_vertices[1], c=m_vertices[2];
      return (a.x*(b.y-c.y) + b.x*(c.y-a.y) + c.x*(a.y-b.y))/2;
    }
};
```

Inheritance
+ Constructors
+ Destructors

Object creation and destruction

- Each object instance has one concrete type
  - But… it may have multiple base types
- Base types might still need constructors
  - Abstract base types might have data members
  - Concrete types can also be base types
- Base types might still need destructors
  - Might need to release resources from base class

Order of construction

- Object instances are constructed down the hierarchy
  - First construct the base class
  - Then construct the derived class
  - …
  - Finally: construct the concrete class

```
1. Shape::Shape()
2. Polygon::Polygon(const vector<point> &)
3. Triangle::Triangle(point p1, point p2, point p3)
```

```
int main()
{
    Triangle *tri=new Triangle({0,0},{0,1},{1,0});
    delete tri;
}
```

Order of destruction

- Object instances are constructed down the hierarchy
  - First call the concrete (most-derived) types destructor
  - Then call destructor of next base class
  - …
  - Then call final destructor of final base class

```
1. Triangle::~Triangle()
2. Polygon::~Polygon()
3. Shape::~Shape()
```

```
int main()
{
    Triangle *tri=new Triangle({0,0},{0,1},{1,0});
    delete tri;
}
```

Default constructors + destructors

- The compiler makes sure each step is followed
  - If you don’t specify a constructor, the default is used
  - If you don’t define a constructor, an implicit one is built
  - If you don’t define a destructor, an implicit one is built

- The implicit (built-in) default constructor will:
  1. Call the default constructor of the base class
  2. Call the default constructor of member variables
- The implicit (built-in) destructor will:
  1. Call the destructor of member variables
  2. Call the destructor of the base class

Calling a specific base constructor

- Sometimes you need a non-default base constructor
  - You cannot call the base constructor like a function
  - It has to happen ***before*** the derived constructor runs
- You can call it using the base class name
  - Parameters can be used to supply constructor arguments

```
class Derived
    : public Base
{
    Derived()
        : Base()
    {}
};
```

```
class Derived
    : public Base
{
    Derived()
        : Base(4.5, “Hello”)
    {}
};
```

Constructing member variables

- Sometimes you want to construct member variables
- You can use the same syntax with member name

```
class MyClass
{
private:
    string m_string;
public:
    MyClass()
        : m_string("Hello")
    {}
};
```

```
class MyClass
{
private:
    string m_string;
    vector<int> m_vector;
public:
    MyClass()
        : m_string("Hello")
        , m_vector({1,2,3,4})
    {}
};
```

Destructors and Constructors

- Objects always have well define lifetimes
  - Constructors are called in order
  - Destructors are called in reverse order
- You can control access to constructors
  - Protected constructors only accessible to derived classes
- Can explicitly call constructors on base classes
- Can explicitly construct member variables

- The goal is always to maintain valid state