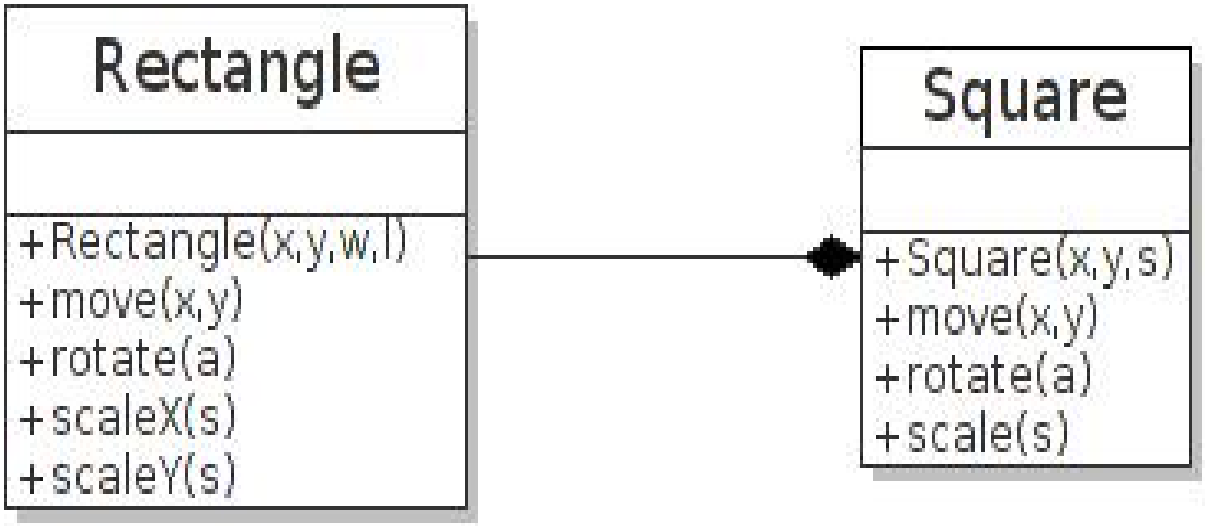
Inheritance

Reuse through Composition

- New objects can be constructed by composing existing ones
- Aggregate reuses implementation of parts wherever possible
- Aggregate hides implementation of parts behind its own interface
- Flexible way of reusing behavior
- Examples
  - Lego blocks
  - Puzzle pieces


Rectangle
+Rectangle(x,y,w,l)
+move(x,y)
+rotate(a)
+scaleX(s)
+scaleY(s)
Square
+Square(x,y,s)
+move(x,y)
+rotate(a)
+scale(s)

```
class Rectangle {

        private:
                Point * center;
                int width;
                int height;
                int angle;

                Point **  getPoints();
                void       redraw(...);

        public:
                Rectangle(...)
                void move(int x,int y);
                void rotate(int angle);
                void scaleX(int s);
                void scaleY(int s);
};
```

```
Rectangle(int x,int y,int w,int h){
        center = new Point(x,y);
        width = w;
        height = h;
        angle = 0;
}
```

```
void Rectangle::move(int x,int y){
        center->move(x,y);
}
```

```
void Rectangle::rotate(int angle){
        Point** points = getPoints();
        for(int i=0; i < 4; i++){
                points[i]->rotate(angle);
        }
}
```

```
 void Rectangle::scaleX(int s){
        width += s;
 }
```

```
 void Rectangle::scaleY(int s){
        height += s;
 }
```

```
class Square {

    private:
        Point * center;
        int size;
        int angle;

        Point **  getPoints();
        void      redraw(...);

    public:
        Square(...)
        void move(int x,int y);
        void rotate(int angle);
        void scale(int s);
};
```

```
Square(int x,int y,int s){
    center = new Point(x,y);
    size = s;
    angle = 0;
}
```

```
void Square::move(int x,int y){
    center->move(x,y);
}
```

```
void Square::rotate(int angle){
    Point** points = getPoints();
    for(int i=0; i < 4; i++){
        points[i]->rotate(angle);
    }
}
```

```
void Square::scale(int s){
    width += s;
    height += s;
}
```

Square composing a Rectangle

```
class Square {

      private:
            Rectangle * rect;

      public:
            Square(...)
            void move(int x,int y);
            void rotate(int angle);
            void scale(int s);
};
```

```
Square(int x,int y,int s){
      rect = new Rectangle(x,y,s,s);
}
```

```
void Square::move(int x,int y){
      rect->move(x,y);
}
```

```
void Square::rotate(int angle){
      rect->rotate(angle);
}
```

```
void Square::scale(int s){
      rect->scaleX(s);
      rect->scaleY(s);
}
```

Inheritance

- Inheritance is another reusability technique
- Children inherit properties from their parents
  - Interface
  - Implementation
- Less flexible than composition but more intuitive
- Children can be seen as specialization of parents
  - They add / update behavior of their parents
- Parents can be seen as generalization of children

Square inheriting a Rectangle

```
class Square : public Rectangle{


      public:
            Square(...)
            void scaleX(int s);
            void scaleY(int s);
            void scale(int s);

};
```

```
Square(int x,int y,int s)
      : Rectangle(x,y,s,s)
{
}
```

```
void Square::scaleX(int s){
      Rectangle::scaleX(s);
      Rectangle::scaleY(s);
}
```

override

```
void Square::scaleY(int s){
      this->scaleX(s);
}
```

override

```
void Square::scale(int s){
      this->scaleX(s);
}
```

Overriding

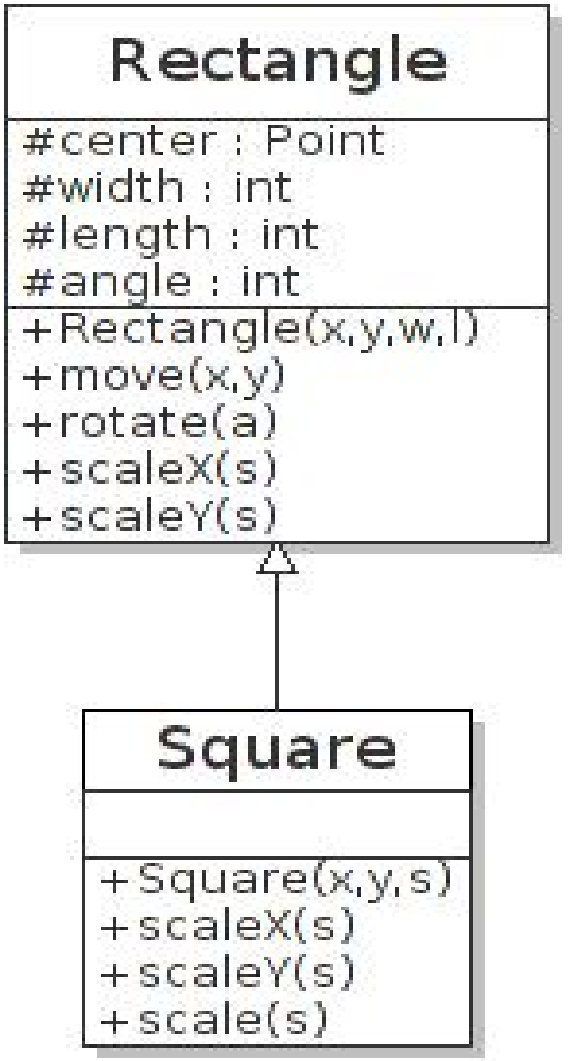
- Its a way to hide an existing behavior and introduce a new behavior with same name
- Newly defined behavior takes precedence
- Hidden behavior is still accessible
- Syntactically overriding occurs when method signature is exactly same for the overriding and overridden method
  - Signature includes name, parameters and their corresponding types

Non-virtual methods should not be overriden

What is inherited?

- Interface as well as implementation
- Public and protected members are inherited and are accessible
- Private members are inherited but are not accessible
  - May be accessible through public methods
- Static members are inherited
- Constructors are **not** inherited

UML Notation



Inheritance as subclass or extension

- Subclass
  - A subset of a larger set
  - Example:
    - Automobile is a subclass of vehicle
    - Car is a subclass of Automobile
- Extension
  - If B inherits from A, all services of A are automatically available in B
  - B can add services of its own
  - B can modify services it inherits

Inheritance as subtype

- *Is A* rule
- A subtype IS A super type
  - Same as super in terms of interface
  - Behavior may be different
- Anything that is supported by super would also be supported by subtype, hence the subtype can be **substituted** wherever super is used without breaking the program
- super **cannot** be substituted for subtype

```
void scaleX(Rectangle *r, int sf) {
	r->scaleX(sf);
}

int main(){

	Rectangle * r  = new Rectangle(...);
	Square * s  = new Square(...)

	scaleX(r,10);
	scaleX(s,10); // ?
}
```

```
void scale(Square *s, int sf) {
	s->scale(sf);
}

int main(){

	Rectangle * r  = new Rectangle(...);
	Square * s  = new Square(...)

	scale(s,10);
	scale(r,10);  // compile error
}
```

Polymorphism

- Ability to take multiple forms
  - Same reference can refer to objects of different child types at runtime
  - When a polymorphic call is made through a polymorphic reference, correct behavior is invoked
- Polymorphism is implemented using:
  - Subtyping
  - Overriding
  - Dynamic binding

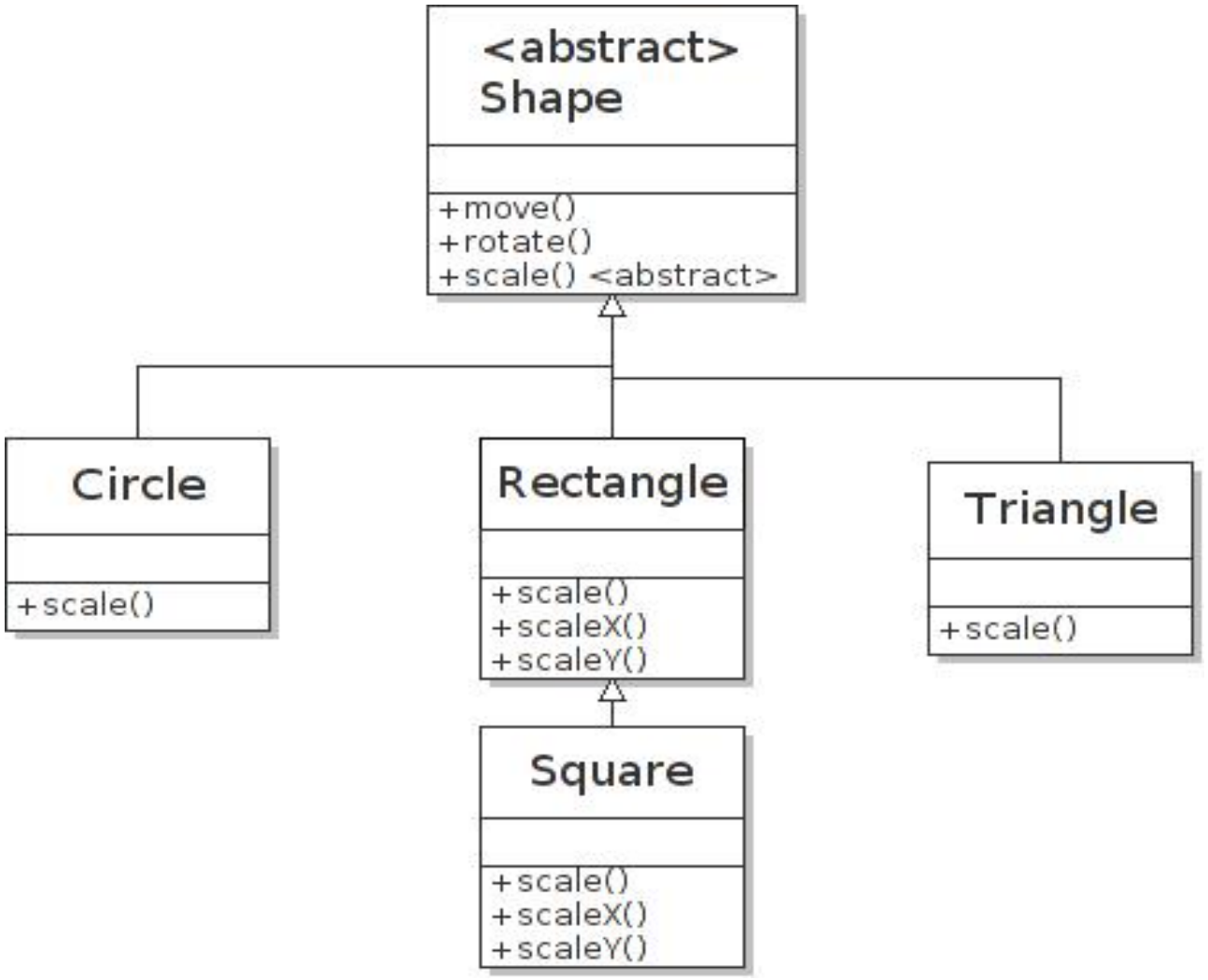
```
class Rectangle {

        private:
                Point * center;
                int width;
                int height;
                int angle;

                Point **  getPoints();
                void       redraw(...);

        public:
                Rectangle(...)
                void move(int x,int y);
                void rotate(int angle);
                virtual void scaleX(int s);
                virtual void scaleY(int s);
};
```

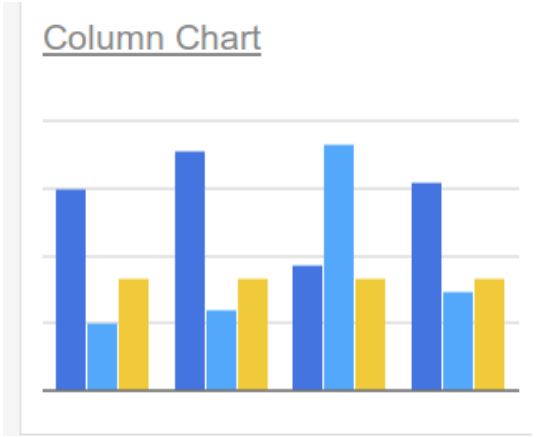
```
class Square : public Rectangle{


        public:
                Square(...)
                virtual void scaleX(int s);
                virtual void scaleY(int s);
                virtual void scale(int s);

};
```

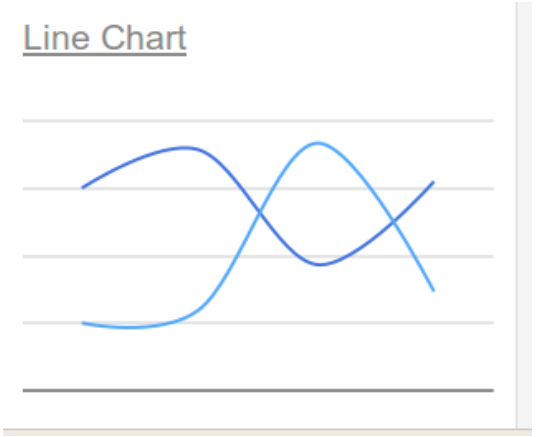
```
void scaleX(Rectangle *r, int sf) {
	r->scaleX(sf);
}

int main(){

	Rectangle * r  = new Rectangle(...);
	Square * s  = new Square(...)

	scaleX(r,10);
	scaleX(s,10); // ?
}
```

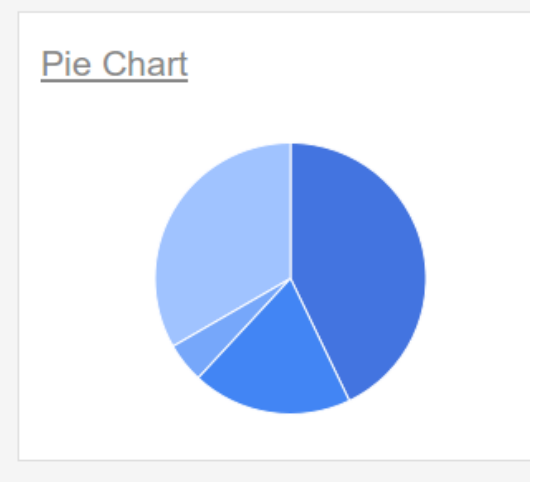
Abstract classes

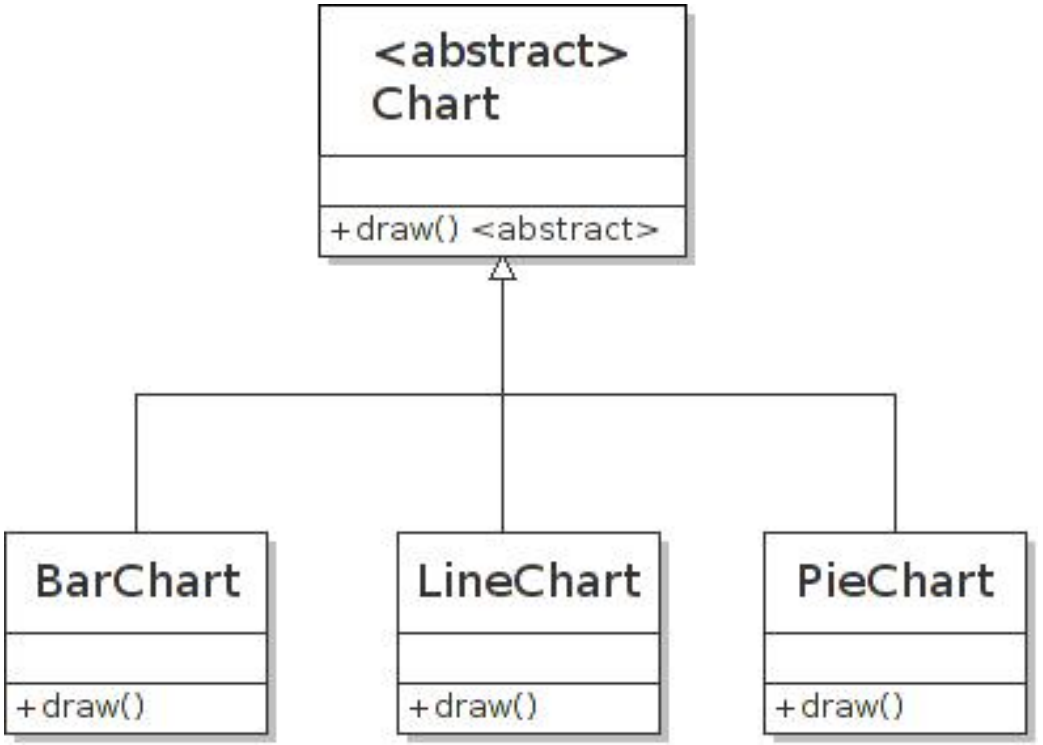
- Partial implementations
- Cannot be instantiated
- Used to tie classes in a common hierarchy
- Makes possible to specify a basic concept that can be extended through subtyping and polymorphism


<abstract>
Shape
+move()
+rotate()
+scale() <abstract>
Circle
+scale()
Rectangle
+scale()
+scaleX()
+scaleY()
Triangle
+scale()
Square
+scale()
+scaleX()
+scaleY()


Column Chart


Line Chart


Pie Chart


<abstract>
Chart
+draw() <abstract>
BarChart
+draw()
LineChart
+draw()
PieChart
+draw()

Down-casting vs Polymorphism

- Downcasting will invoke correct behavior but
  - can result in potentially unsafe calls
  - Not an extensible option – addition of new types require changes to program
- Polymorphism provides extensibility and safety

Quiz

- Bank customer has option to open following types of accounts with the bank:
  - **Current Account:** bears no profit
  - **Savings Account:** bears profit on quarterly period
  - **Time Deposit Account:** bears profit only if deposited amount is kept in the account for the time specified by the bank
- Bank may offer other account types in future
- Customer can maintain multiple accounts with bank
- **Draw a class diagram**
- **Write a program to calculate profit earned (if any) by the customer at the end of the month**

Inheritance vs Composition

- Both techniques provide ability to reuse
- Composition provides better encapsulation
  - Easy to change the interface of whole and parts
  - Easy to change the underlying implementation without ripple effects
  - Easy to switch between different implementations on the runtime
- Inheritance provides subtyping and polymorphism
  - Provides a mechanism to support extensibility in software
  - Provides better understandability

Prefer composition over inheritance for reuse, inheritance over composition for extensibility

Private Inheritance

- Doesn't represent the ISA relationship
- Only inherits the implementation, not the interface
- Almost same as composition
- Some languages don't support private inheritance

```
class Square : private Rectangle{

        public:
                Square(...)
                void scale(int s);
                using move;
                using rotate;
};
```

```
Square(int x,int y,int s)
        : Rectanlge(x,y,s,s)
{
}
```

```
  void Square::scale(int s){
          Rectangle::scaleX(s);
          Rectangle::scaleY(s);
  }
```

```
void scaleX(Rectangle *r, int sf) {
	r->scaleX(sf);
}

int main(){

	Rectangle * r  = new Rectangle(...);
	Square * s  = new Square(...)

	scaleX(r,10);
	scaleX(s,10); // ?
}
```

Inheritance in Java

- Only public inheritance
  - No private inheritance
- Access specifier rules are same as C++ except
  - An inherited public member cannot be declared private
- All methods are virtual

```
class Rectangle {

        private Point * center;
        private int width;
        private int height;
        private int angle;

        public Rectangle(...) { … }
        public void move(int x,int y) { … }
        public void rotate(int angle) { … }
        public void scaleX(int s) { … }
        public void scaleY(int s) { … }
}
```

```
class Square extends Rectangle{


        public Square(...) { … }
        public void scaleX(int s) {… }
        public void scaleY(int s) {… }
        public void scale(int s) {… }

}
```