Lecture 2: Introduction to Objects

Book reference : Thinking in C++: intro to standard c++ Chapter 1

The progress of abstraction

- The **type and quality** of **language abstractions** determine the **complexity** of the problems you can solve with programming languages.
- **Assembly language** and **imperative languages** are **low-level** abstractions of the machine, requiring the programmer to think in terms of the **computer's structure** rather than the **problem's structure**.
- **Object-oriented programming (OOP)** is a **high-level** abstraction that allows the programmer to describe the problem in terms of the **problem's elements**, rather than in terms of the **computer's operations**.
- **Multiparadigm languages** combine various **programming approaches** to solve different kinds of problems with **appropriate abstractions**.

Alan kay and Smalltalk

1. **Everything is an object**
2. **A program is a bunch of objects telling each other what to do by sending message.**
3. **Each object has its own memory made up of other objects**
4. **Every object has a type.**
5. **All objects of a particular type can receive the same messages**

Book page:26

An object has an interface

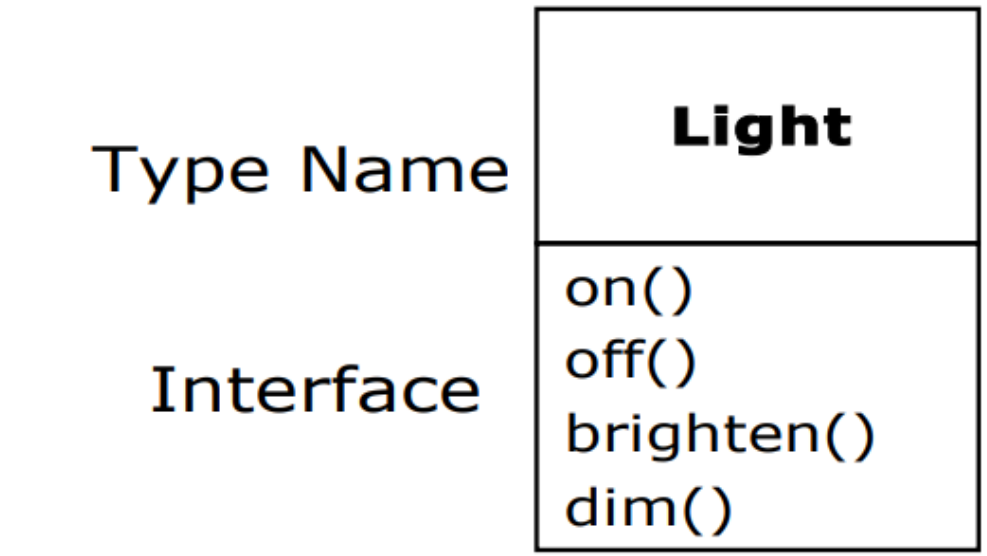
- Sometimes, we want to make programs that act like things in the real world, such as animals, cars, games, etc. These things have some features and can do some actions. For example, a car has a color, a speed, and a number of seats, and it can start, stop, and turn.
- To make programs like this, we use a way of programming called **object-oriented programming (OOP)**. In OOP, we make **objects** that represent the things we want to program. Each object has its own features and actions, which we call **characteristics** and **behaviors**.
- To make objects, we need to define a **class** for each kind of thing we want to program. A class is like a blueprint or a recipe that tells us how to make objects of that kind. For example, we can have a class for cars, and then we can make many different cars from that class. Each car is an **object** of the car class.
- A class is also a kind of **data type**, which means it tells us what kind of data we can store and use in our program. For example, a number is a data type, and it tells us we can do math with it. A class is a data type that we create ourselves, and it tells us what kind of features and actions we can have for our objects.

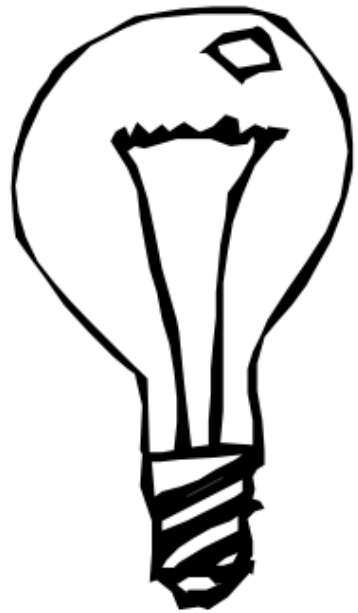
An object has an interface

- The first programming language that used classes was called **Simula-67**, and it was made for making programs that pretend to be real situations, like a bank or a traffic system. These programs are called **simulations**, and they help us understand and test how things work in the real world.
- When we make objects, we need to know what kind of things we can ask them to do or tell them to change. This is called the **interface** of the object, and it is like a menu of options that we can choose from. The interface of an object depends on its **type**, which is the same as its **class**. For example, the type of a car object is the car class, and the interface of a car object is the list of features and actions that the car class defines.
- Sometimes, there can be more than one way to make a class for the same type of thing. For example, we can have different classes for cars that use different engines or designs. This is called having different **implementations** of the same type. The implementation of a class is the details of how the class works inside, and it is not important for us to know when we use the objects. We only need to know the interface of the objects, which is the same for all the classes of the same type.
- Using OOP has many benefits for making programs. It helps us to **extend** the programming language with new data types that we need, to **simplify** complex problems by breaking them into smaller parts, and to **map** the problem space to the solution space by making objects that match the things we want to program.

Type Name

Interface





```
Light lt;
lt.on();
```

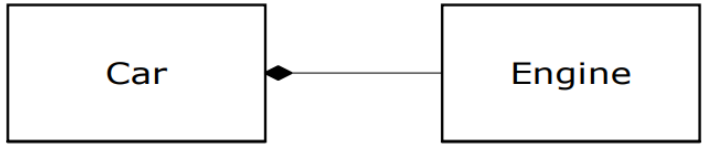
- Here, the name of the type/class is Light, the name of this particular Light object is lt, and the requests that you can make of a Light object are to turn it on, turn it off, make it brighter or make it dimmer.
- You create a Light object by declaring a name (lt) for that object.
- To send a message to the object, you state the name of the object and connect it to the message request with a period (dot).
- From the standpoint of the user of a pre-defined class, that's pretty much all there is to programming with objects. (like String Class in java)

The hidden implementation

- **Class creators** and **client programmers**: two kinds of programmers who make and use **classes**, which are like recipes for making **objects**
- **Objects**: things in the program that have features and actions, like cars, animals, games, etc.
- **Hidden parts** and **visible parts**: parts of a class that only the class creator can see and change, and parts that the client programmer can see and use
- **Access control**: a way to make some parts of a class hidden and some parts visible, using words like **public**, **private**, and **protected**
- **Why hide parts?**: to prevent mistakes, to make things simpler, and to allow changes without breaking the program

Reusing the implementation (Objects)

- Once a class has been created and tested, it should (ideally) represent a useful unit of code that can be reused. It takes experience and insight to produce a **good design**
- Code reuse is one of the greatest advantages of oop
- The simplest way to reuse a class is to just use an object of that class directly, but you can also place an object of that class inside a new class “creating a member object”
- Because you are composing a new class from existing classes, this concept is called composition (or more generally, aggregation). Composition is often referred to as a “has-a” relationship, as in “a car has an engine.



Reusing the implementation (Objects)

- Composition gives you more flexibility than inheritance.
- You can make the member objects of your new class private and change them without affecting the client programmer code.
- You can also change the member objects at runtime to alter the program behavior dynamically.
- Inheritance has compile-time restrictions and can lead to complex and awkward designs.
- You should prefer composition over inheritance when creating new classes, as it is simpler and more flexible.

Inheritance: reusing the interface (Base class)

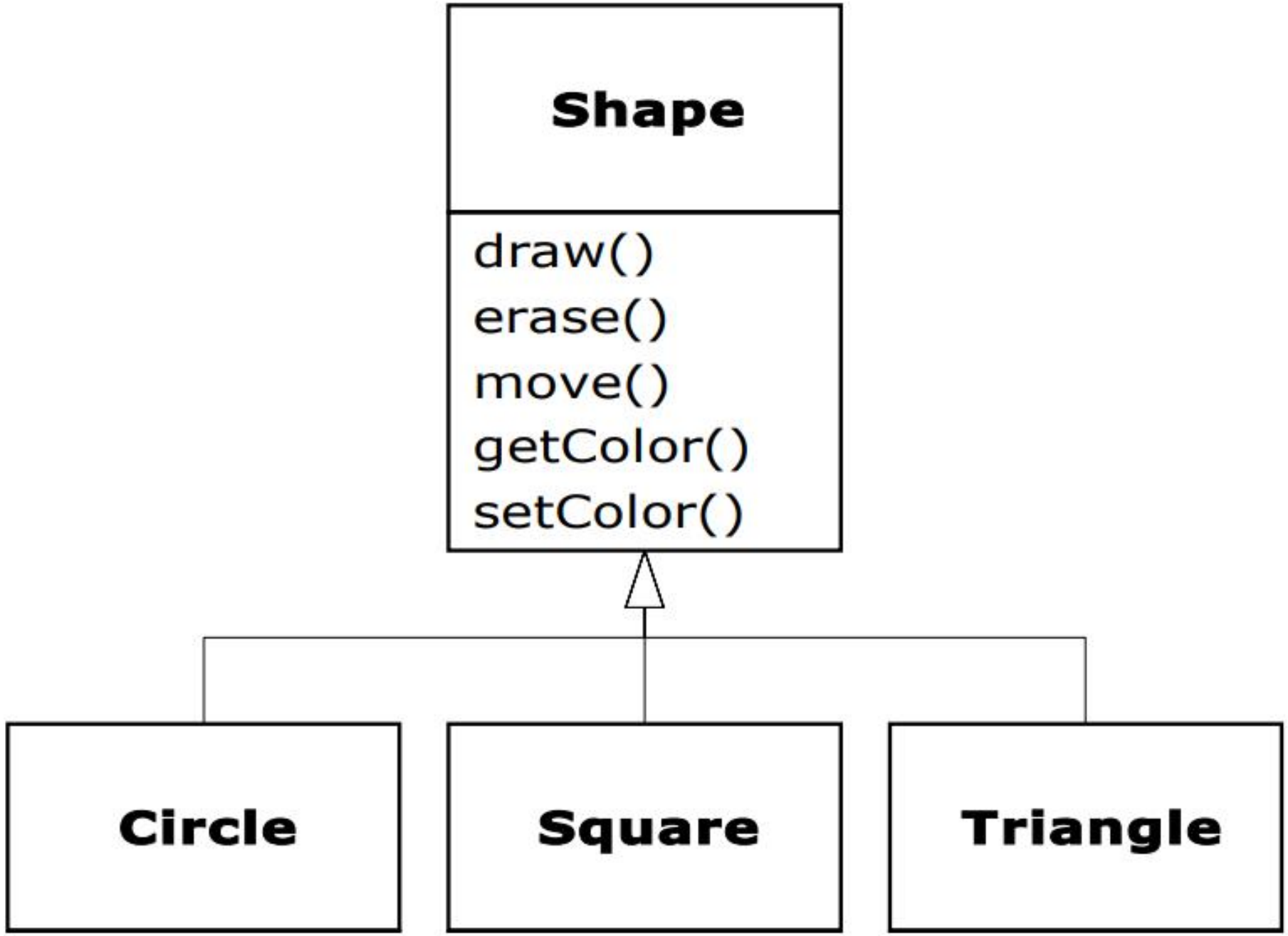
- A class defines the constraints and relationships of a set of objects
- Two classes can share some characteristics and behaviors, but one class may have more or different ones than another
- Inheritance is a way of expressing the similarity between classes using base classes and derived classes
- A base class contains the common characteristics and behaviors of the classes derived from it
- You create a base class to represent the core concept of some objects in your system
- You derive other classes from the base class to represent the different variations of the core concept

Example 1: Trash Recycling

- a trash-recycling machine sorts pieces of trash.
- The base type is “trash,” and each piece of trash has a weight, a value, and so on, and can be shredded, melted, or decomposed.
- From this, more specific types of trash are derived that may have additional characteristics (a bottle has a color) or behaviors (an aluminum can may be crushed, a steel can is magnetic).
- In addition, some behaviors may be different (the value of paper depends on its type and condition).
- Using inheritance, you can build a type hierarchy that expresses the problem you’re trying to solve in terms of its types.

Example 2: Shape

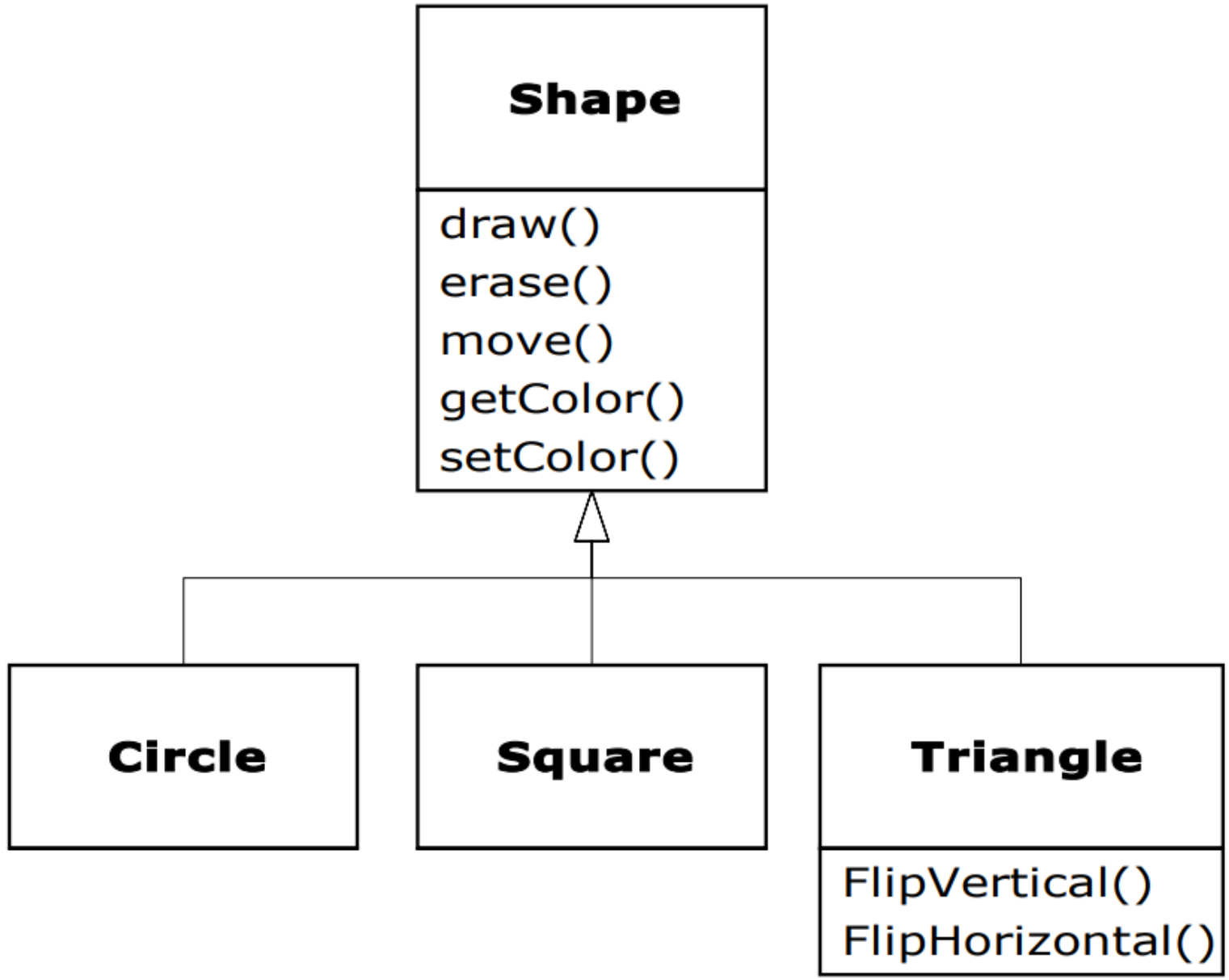
- The base type is “shape,” and each shape has a size, a color, a position, and so on.
- Each shape can be drawn, erased, moved, colored, etc. From this, specific types of shapes are derived (inherited): circle, square, triangle, and so on, each of which may have additional characteristics and behaviors.
- Certain shapes can be flipped, for example.
- Some behaviors may be different, such as when you want to calculate the area of a shape.
- The type hierarchy embodies both the similarities and differences between the shapes.

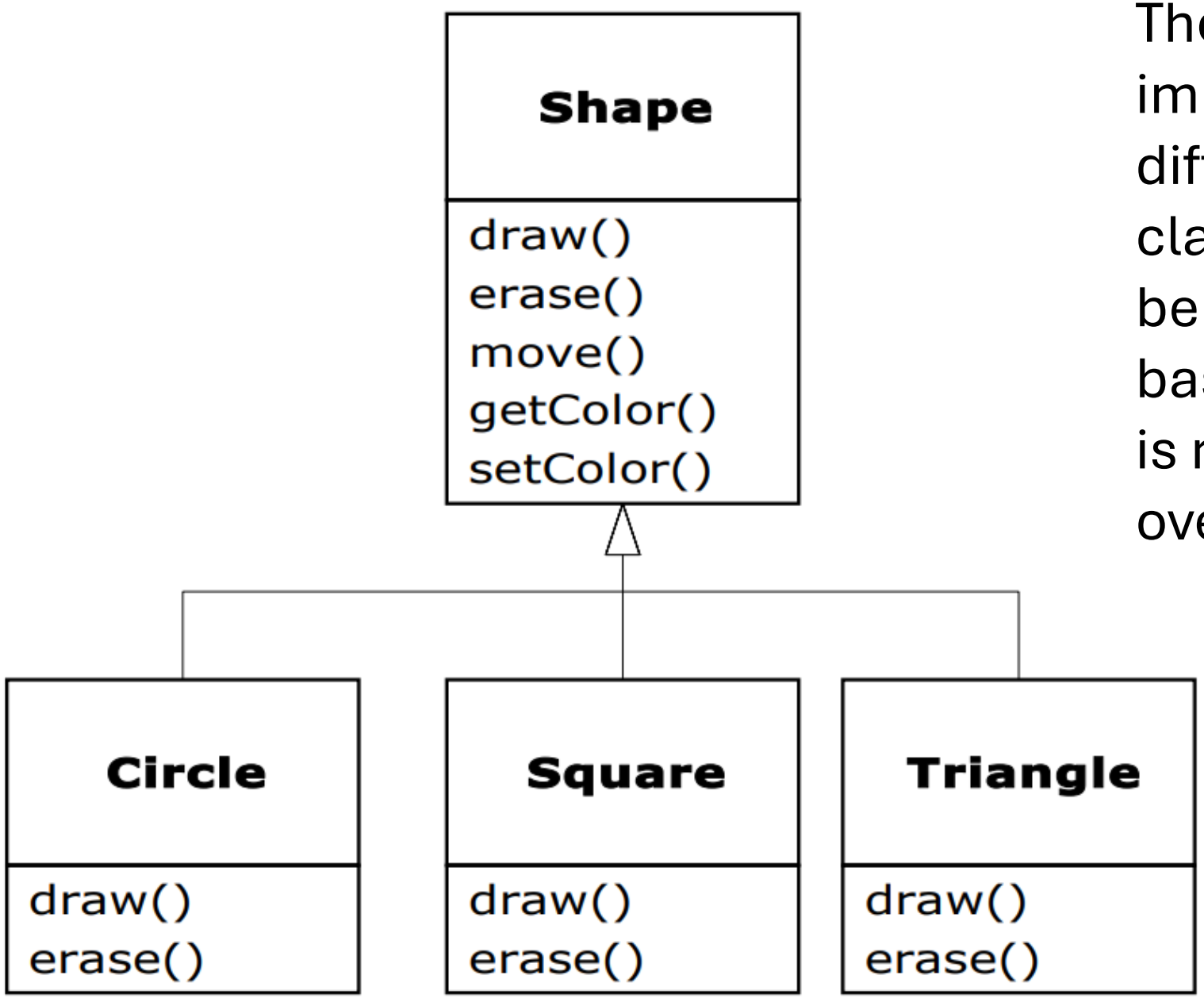

Shape
draw()
erase()
move()
getColor()
setColor()
Circle
Square
Triangle

- Casting the solution in the same terms as the problem is beneficial as it eliminates the need for many intermediate models.
- In object-oriented design, the type hierarchy allows you to go directly from the real-world system description to the code system description.
- When you inherit from an existing type, you create a new type that contains all the members of the existing type and duplicates the base class interface.
- The derived class is the same type as the base class because it can receive all the messages that the base class can.
- Both the base class and derived class have the same interface, so there must be some code to execute when an object receives a particular message.

Inheritance

- If you simply inherit a class and don’t add anything else, the derived class objects have the same behavior as the base class, which can be uninteresting.
- You can differentiate your new derived class from the original base class by adding new functions that are not part of the base class interface.
- This simple use of inheritance can sometimes be the perfect solution to your problem.
- However, you should consider whether your base class might also need these additional functions. This process of discovery and iteration of your design is a regular occurrence in object-oriented programming (OOP).


Shape
draw()
erase()
move()
getColor()
setColor()
Circle
Square
Triangle
FlipVertical()
FlipHorizontal()



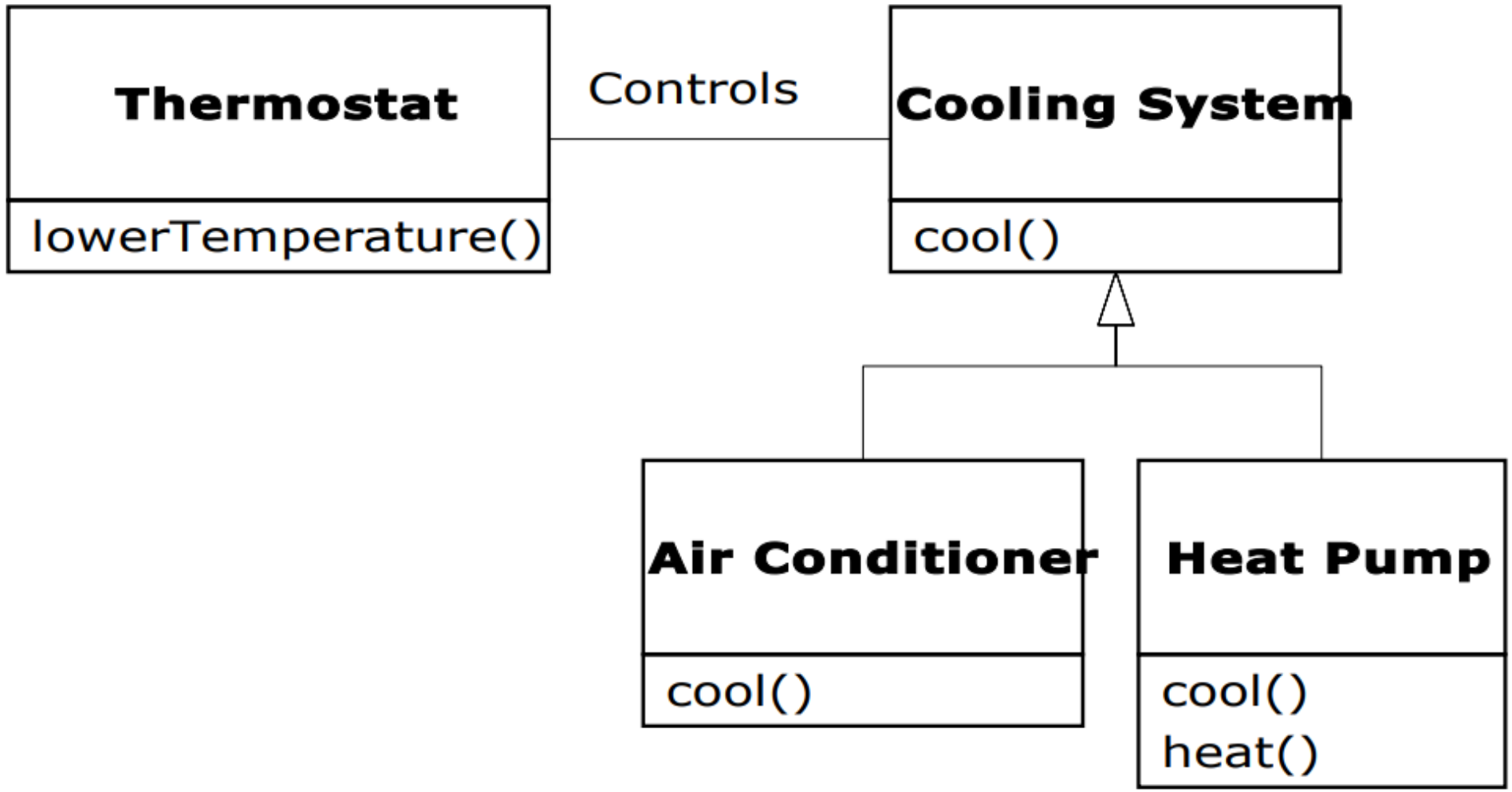
The second and more important way to differentiate your new class is to change the behavior of an existing base-class function. This is referred to as overriding that function.

Substitution principle

- Should inheritance override *only* base-class functions (and not add new member functions that aren't in the base class)? This would mean that the derived type is **exactly the same type as the base class** since it has exactly the same interface.
- As a result, you can exactly **substitute** an object of the derived class for an object of the base class. This can be thought of as pure substitution, and it's often referred to as the ***substitution*** principle.
- In a sense, this is the ideal way to treat inheritance.

Is-a vs. is-like-a relationship

- We often refer to the relationship between the base class and derived classes in this case as an **is-a** relationship, because you can say **“a circle is a shape.”**
- when you extend an interface and create a new type(class) t the substitution isn’t perfect because your new functions are not accessible from the base type.
- This is known as ***is-like-a*** relationship


Thermostat
lowerTemperature()
Controls
Cooling System
cool()
Air Conditioner
cool()
Heat Pump
cool()
heat()

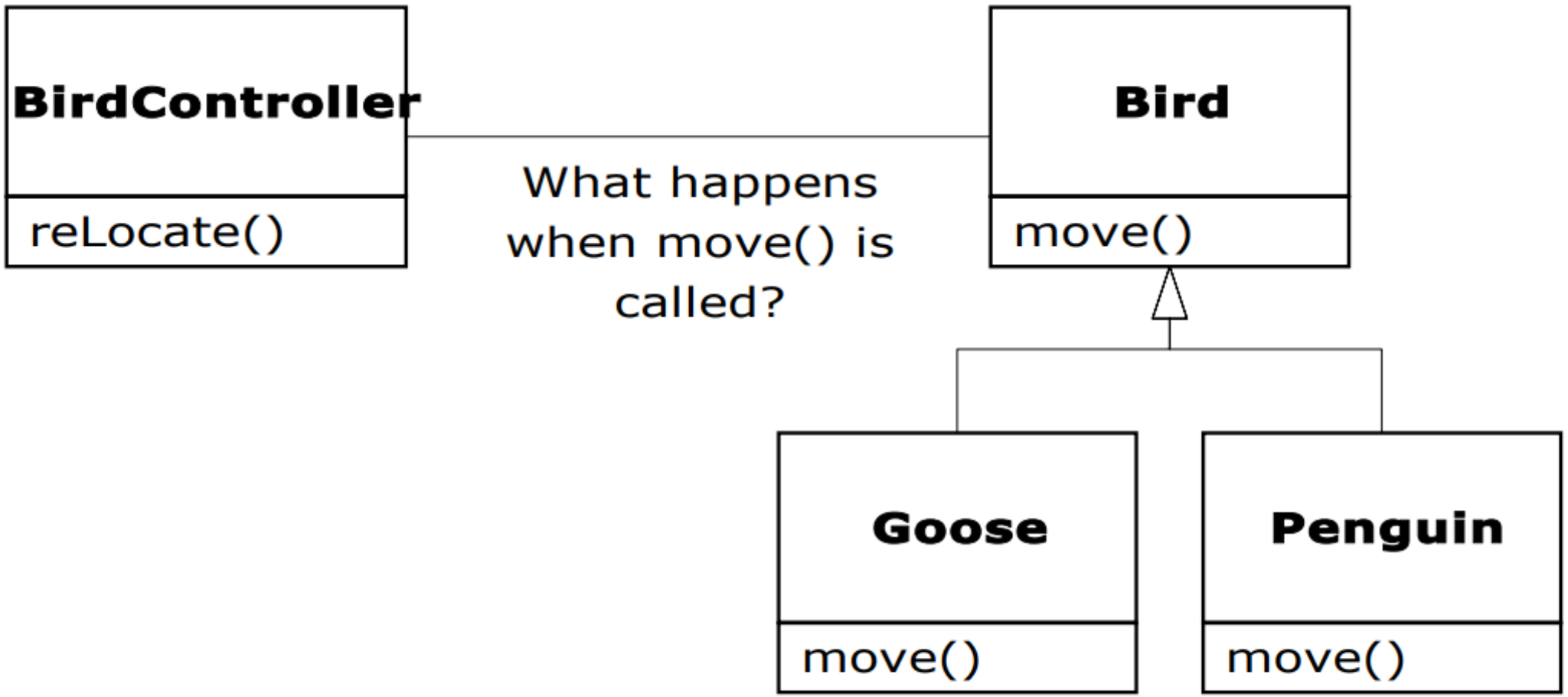
Interchangeable objects with polymorphism

1. **Class Hierarchies**:
    1. Imagine you have different classes of shapes: circles, squares, triangles, and more.
    2. These classes can be organized in a hierarchy. For instance, all shapes are related because they share common properties (like being able to be drawn, erased, and moved).
2. **Base Class vs. Specific Class**:
    1. When you work with these shapes in your code, you can treat them as either their specific class (like a circle or square) or as a more general base class (just “shape”).
    2. Treating them as the base class allows you to write code that doesn’t depend on the specific details of each shape.
3. **Why Use the Base Class?**:
    1. By treating shapes as the base class, you can create functions that work with any shape. These functions don’t need to know whether they’re dealing with a circle, square, or triangle.
    2. For example, a function to draw a shape can simply send a message to the shape object without worrying about the shape’s specific class.
4. **Adding New Classes**:
    1. Suppose you want to add a new shape, like a pentagon, to your program.
    2. If you’ve designed your code to work with the base class, you can easily add the pentagon subclass without changing the existing functions. They’ll still work!
5. **Benefits**:
    1. This flexibility to add new classes without breaking existing code is powerful. It improves software design and reduces maintenance costs.

- In summary, treating objects as their base class allows you to write more flexible and extensible code, making it easier to handle different situations in an object-oriented program.

1. **The Challenge**:
    1. When you write code that interacts with these generic base classes, the compiler doesn't know exactly which specific code will be executed.
    2. The whole point is that when you send a message (like asking a shape to draw itself), you don't want to worry about the specific type of object. You want the object to handle it correctly based on its own class.
2. **Adding New Subclasses**:
    1. Now, let's say you add a new subclass, like a "pentagon" shape. You don't want to change all your existing code.
    2. By treating shapes generically, you can add new subclasses without modifying existing function calls. The right behavior will still happen!
3. **The Compiler's Dilemma**:
    1. But here's the catch: How does the compiler know what code to execute?
    2. For instance, in the diagram(next slide), the "BirdController" works with generic "Bird" objects. It doesn't know their exact class (whether it's a goose or a penguin).
4. **Dynamic Behavior**:
    1. When you call a function like "move()" on a generic bird, the magic happens at runtime.
    2. The bird object itself knows its specific behavior. So, whether it's a goose (running, flying, or swimming) or a penguin (running or swimming), it handles it correctly.

- In summary, treating objects generically allows you to write flexible code that adapts to specific classes at runtime. It's like letting the birds decide how to move without the controller needing to know every detail!


BirdController
reLocate()
What happens when move() is called?
Bird
move()
Goose
move()
Penguin
move()

1. **Early Binding vs. Late Binding**:
    - In traditional programming, the compiler generates function calls directly. This is called **early binding**.
    - But in object-oriented programming (OOP), things get interesting. The compiler can't know the exact code to execute until runtime. This is where **late binding** comes into play.
2. **Early Binding (Static Binding)**:
    - Early binding happens during compilation.
    - The compiler knows the exact function to call based on the method name and arguments.
    - It stores this information in a virtual method table (v-table) inside the compiled program.
3. **Late Binding (Dynamic Binding)**:
    - With late binding, the compiler doesn't read enough information to verify the method exists or bind its slot in the v-table.
    - Instead, at runtime, the method is looked up by name.
    - The magic happens when the object calculates the address of the function body using stored information.
4. **Why Late Binding?**:
    - Late binding is essential for flexibility. When you send a message to an object, you don't want to worry about the exact code.
    - Each object behaves differently based on its specific type (thanks to that special bit of code).
5. **Virtual Functions**:
    - To achieve late binding in C++, use the **virtual** keyword
    - By default, member functions are not dynamically bound.
    - **Virtual** functions allow you to express differences in behavior among classes in the same family.

In summary, late binding lets objects decide what to do at runtime, making OOP powerful and flexible.

1. **Polymorphism and Base Classes**:
   - In object-oriented programming (OOP), **polymorphism** is a powerful concept.
   - Imagine a family of classes, all based on a common interface (like shapes: circles, squares, etc.).
   - Polymorphism allows us to write code that interacts with the **base class** without worrying about specific details of each derived type.
2. **The Goal**:
   - We want to write a single piece of code that talks only to the base class (like a generic “Shape”).
   - This code should be decoupled from type-specific information, making it simpler and easier to understand.
3. **Extensibility**:
   - If we add a new type (say, a “Hexagon”) through inheritance, the existing code should work seamlessly for the new shape.
   - This extensibility is a key benefit of polymorphism.

- If you write a function in C++

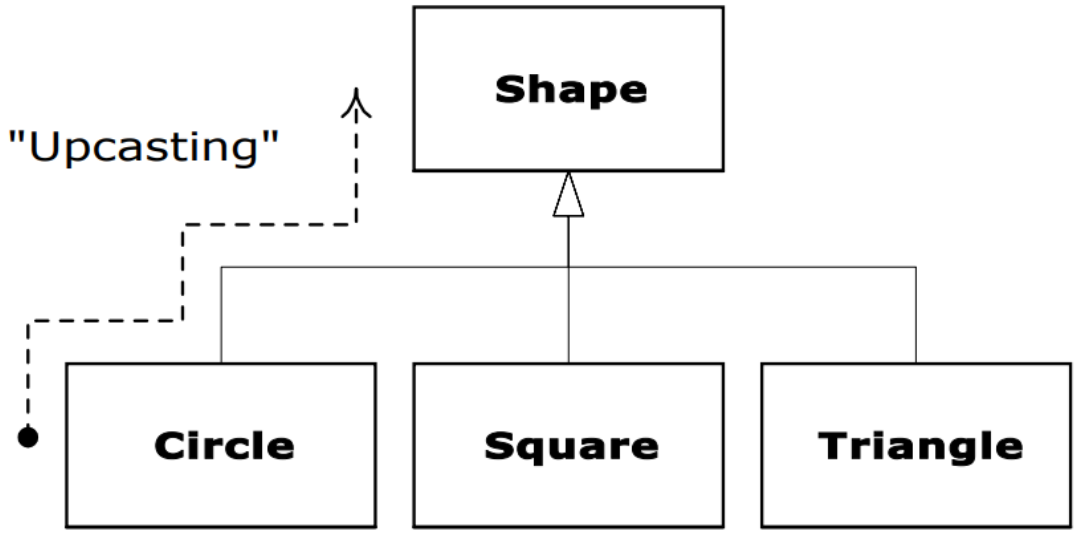
```
void doStuff(Shape& s) {
  s.erase();
  // ...
  s.draw();
}
```

- This function speaks to any Shape, so it is independent of the specific type of object that it's drawing and erasing
- If in some other part of the program we use the doStuff( ) function:

```
Circle c;
Triangle t;
Line l;
doStuff(c);
doStuff(t);
doStuff(l);
```

The calls to **doStuff( )**automatically work right, regardless of the exact type of the object.

- What's happening here is that a **Circle** is being passed into a function that's expecting a **Shape**. Since a **Circle** *is* a **Shape** it can be treated as one by **doStuff**( ). That is, any message that **doStuff( )** can send to a **Shape**, a **Circle** can accept.
- So it is a completely safe and logical thing to do.
- This is called **upcasting**



1. **Coupling and Cohesion**:
    1. In software design, **coupling** refers to how interdependent different parts of a system are. If two classes are tightly coupled, changes in one may force changes in the other.
    2. On the other hand, **cohesion** measures whether an element's responsibilities form a meaningful unit. High cohesion means that an element's tasks are related and make sense together.
2. **Decoupling in Inheritance**:
    1. **Decoupling** aims to reduce dependencies between classes or modules. It allows us to separate object interaction from the specifics of inheritance.
    2. When we decouple, we create distinct layers of abstraction. These layers help us achieve polymorphism without tightly binding code to specific types.
3. **Example**:
    1. Imagine a family of shape classes (like circles, squares, and triangles) all based on a common interface (the base class "Shape").
    2. To demonstrate polymorphism, we want to write code that talks only to the base class ("Shape") without worrying about specific derived types.
    3. By decoupling, we ensure that our code remains independent of the exact type of object it's dealing with.
    4. If we later add a new shape (like a "Hexagon") through inheritance, our existing code will still work seamlessly.
4. **Benefits of Decoupling**:
    1. **Flexibility**: Decoupled systems are more flexible. They allow us to extend functionality without breaking existing code.
    2. **Maintainability**: High cohesion and loose coupling lead to easier maintenance.
    3. **Polymorphism**: Decoupling enables polymorphism, where objects can behave differently based on their specific type.

- Decoupling in inheritance helps us build extensible, maintainable, and flexible software by allowing us to treat objects generically while preserving their unique behaviors.