

Implementing `AddChoice()`

- The method will receive three parameters
 - As usual for a class method the `self` parameter is required
 - The text for the choice
 - A Boolean denoting if it is the correct choice or not
- It appends the text as a `_choice`, sets choice number to the `_answer` and calls the inherited `setAnswer()` method:

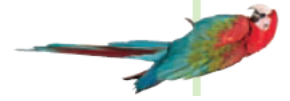
```
def addChoice(self, choice, correct) :  
    self._choices.append(choice)  
    if correct :  
        # Convert the length of the list to a string.  
        choiceString = str(len(self._choices))  
        self.setAnswer(choiceString)
```

Common Error 10.1 (1)



- Confusing Super- and Subclasses
- If you compare an object of type `ChoiceQuestion` with an object of type `Question`, you find that:
 - the `ChoiceQuestion` object is larger; it has an added instance variable, `_choices`,
 - the `ChoiceQuestion` object is more capable; it has an `addChoice()` method.

Common Error 10.1 (2)



- So why is `ChoiceQuestion` called the *subclass* and `Question` the *superclass*?
 - The *super/sub* terminology comes from set theory.
 - Look at the set of all questions.
 - Not all of them are `ChoiceQuestion` objects; some of them are other kinds of questions.
 - The more specialized objects in the subset have a richer state and more capabilities.

Self-Check

Self-Check

- A Road class simulates a road on which vehicles travel. Its add method requires an argument of type Vehicle. Rearrange the lines on the left to add appropriate objects to the road. You can add them in any order. Not all lines are useful.

```
road1 = Road()
```

```
truck1 = Truck()
```

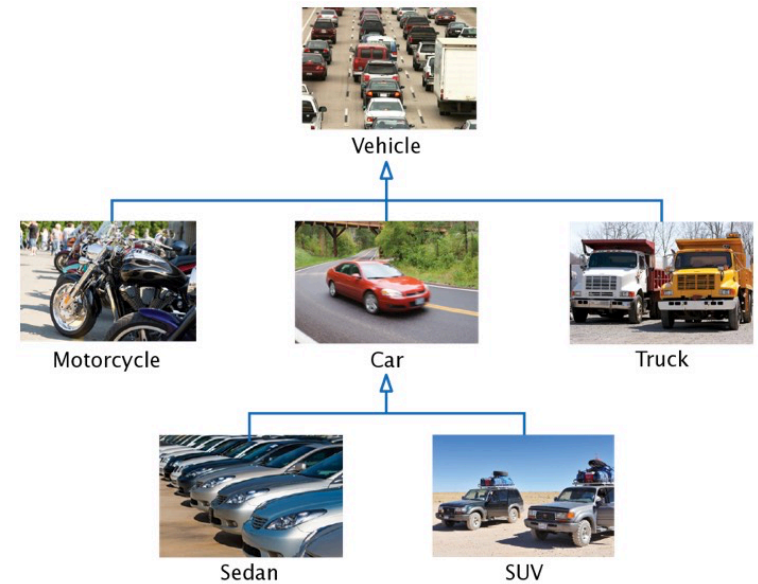
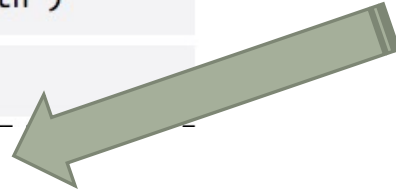
```
beetle = Sedan()
```

```
bmw = "BMW 540i"
```

```
harley = Motorcycle()
```

```
harry = Person("Harry Smith")
```

```
t11 = TrafficLight()
```



A. `road1.add(bmw)`

B. `road1.add(harley)`

C. `road1.add(t11)`

D. `road1.add("beetle")`

E. `road1.add(beetle)`

F. `road1.add(harry)`

G. `road1.add(truck1)`

10.3 Calling the Superclass Constructor (1)

- A subclass constructor can only define the instance variables of the subclass.
- But the superclass instance variables also need to be defined.
- The superclass is responsible for defining its own instance variables.
- Because this is done within its constructor, the constructor of the subclass must explicitly call the superclass constructor.

10.3 Calling the Superclass Constructor (2)

- ❑ To distinguish between super- and sub- class constructor use the `super()` function in place of the `self` reference when calling the constructor:

```
class ChoiceQuestion(Question) :  
    def __init__(self) :  
        super().__init__()  
        self._choices = []
```

- The superclass constructor should be called before the subclass defines its own instance variables.

10.3 Calling the Superclass Constructor (3)

- If a superclass constructor requires arguments, you must provide those arguments to the `__init__()` method.

```
class ChoiceQuestion(Question) :  
    def __init__(self, questionText) :  
        super().__init__(questionText)  
        self._choices = []
```


Syntax 10.2: Subclass Constructor

Syntax `class SubclassName(SuperclassName) :`
 `def __init__(self, parameterName1, parameterName2, . . .) :`
 `super().__init__(arguments)`
 constructor body

The super function
is used to refer to
the superclass.

```
class ChoiceQuestion(Question) :  
    def __init__(self, questionText) :
```

```
        super().__init__(questionText)
```

The superclass
constructor is
called first.

The subclass constructor
body can contain
additional statements.

```
        self._choices = []
```

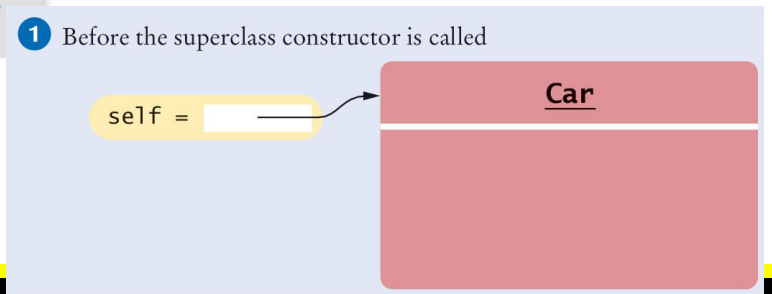
Example: Superclass Constructor (1)

- Suppose we have defined a `Vehicle` class and the constructor which requires an argument:

```
class Vehicle :  
    def __init__(self, numberOfTires) :  
        self._numberOfTires = numberOfTires  
        . . .
```

- We can extend the `Vehicle` class by defining a `Car` subclass:

```
class Car(Vehicle) :  
    def __init__(self) :      # 1
```

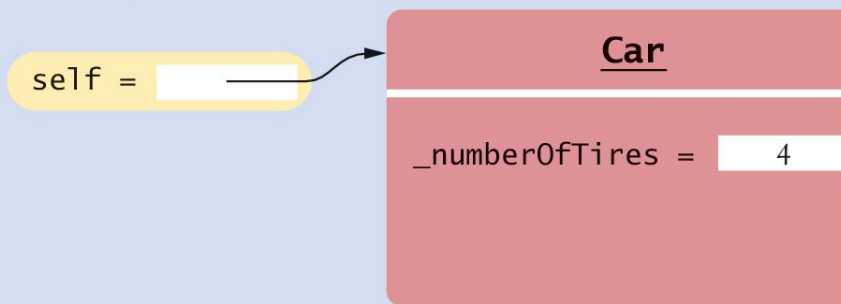


Example: Superclass Constructor (2)

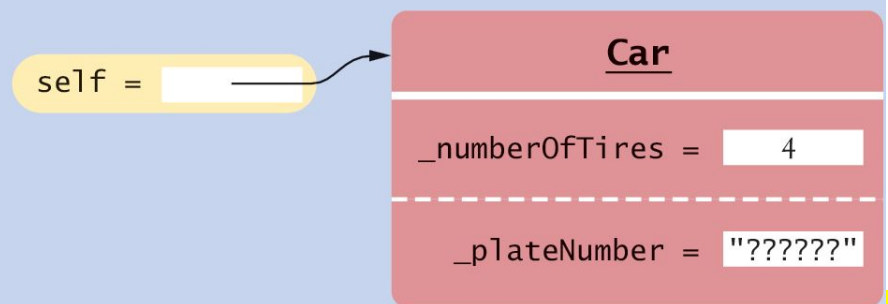
- Now as the subclass is defined, the parts of the object are added as attributes to the object:

```
# Call the superclass constructor to define its  
# instance variable.  
super().__init__(4)    # 2  
  
# This instance variable is added by the  
# subclass.  
self._plateNumber = "??????" # 3
```

2 After the superclass constructor returns



3 After the subclass instance variable is defined



10.4 Overriding Methods

- The `ChoiceQuestion` class needs a `display()` method that overrides the `display()` method of the `Question` class
- They are two different method implementations
- The two methods named `display` are:
 - `Question display()`
 - Displays the text of the private attribute of class `Question`
 - `ChoiceQuestion display()`
 - Overrides `Question display` method
 - Displays the instance variable text String
 - Displays the list of choices which is an attribute of `ChoiceQuestion`

Tasks Needed for Display(): 1


- Display the question text.
- Display the answer choices.



- The second part is easy because the answer choices are an instance variable of the subclass.


```
class ChoiceQuestion(Question) :  
    . . .  
    def display(self) :  
        # Display the question text.  
        . . .  
        # Display the answer choices.  
        for i in range(len(self._choices())) :  
            choiceNumber = i + 1  
            print("%d: %s" % (choiceNumber,  
                             self._choices[i]))
```

Tasks Needed for Display(): 2

- Display the question text. 
 - Display the answer choices.
-
- The first part is trickier!
 - You can't access the text variable of the superclass directly because it is private.
 - Call the display() method of the superclass, using the super() function:

```
def display(self) :  
    # Display the question text.  
    super().display() # OK  
    # Display the answer choices.
```

Tasks Needed for Display(): 3

- Display the question text. 
- Display the answer choices.
- The first part is trickier! (Continued)
 - If you use the self reference instead of the super() function, then the method will not work as intended.

```
def display(self) :  
    # Display the question text.  
    self.display()  
    # Error--invokes display() of ChoiceQuestion.  
    . . .
```

Questiondemo2.py (1)

```
1  ##
2  #  This program shows a simple quiz with two choice questions.
3  #
4
5  from questions import ChoiceQuestion
6
7  def main() :
8      first = ChoiceQuestion()
9      first.setText("In what year was the Python language first released?")
10     first.addChoice("1991", True)
11     first.addChoice("1995", False)
12     first.addChoice("1998", False)
13     first.addChoice("2000", False)
14
15     second = ChoiceQuestion()
16     second.setText("In which country was the inventor of Python born?")
17     second.addChoice("Australia", False)
18     second.addChoice("Canada", False)
19     second.addChoice("Netherlands", True)
20     second.addChoice("United States", False)
21
22     presentQuestion(first)
23     presentQuestion(second)
```

Creates two objects of the ChoiceQuestion class, uses new addChoice() method.

Calls presentQuestion() - next page

Questiondemo2.py (2)

```
25  ## Presents a question to the user and checks the response.
26  # @param q the question
27  #
28  def presentQuestion(q) :
29      q.display()
30      response = input("Your answer: ")
31      print(q.checkAnswer(response))
32
33  # Start the program.
34  main()
```

Uses ChoiceQuestion
(subclass) display()
method.

Questions.py (1)

```
37
38  ## A question with multiple choices.
39  #
40  class ChoiceQuestion(Question) :
41      # Constructs a choice question with no choices.
42      def __init__(self) :
43          super().__init__()
44          self._choices = []
45
46      def addChoice(self, choice, correct) :
47          self._choices.append(choice)
48          if correct :
49              # Convert len(choices) to string.
50              choiceString = str(len(self._choices))
51              self.setAnswer(choiceString)
52
53
54
55
56
```

Inherits from Question class.

New addChoice() method.

Questions.py (2)

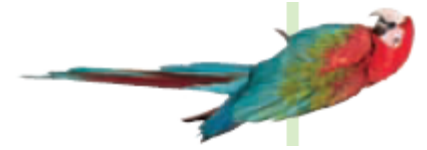
```
57 # Override Question.display().
58 def display(self) :
59     # Display the question text.
60     super().display()
61
62     # Display the answer choices.
63     for i in range(len(self._choices)) :
64         choiceNumber = i + 1
65         print("%d: %s" % (choiceNumber, self._choices[i]))
```

Overridden display()
method.

Program Run

```
In what year was the Python language first released?
1: 1991
2: 1995
3: 1998
4: 2000
Your answer: 2
False
In which country was the inventor of Python born?
1: Australia
2: Canada
3: Netherlands
4: United States
Your answer: 3
True
```

Common Error 10.2 (1)



- Extending the functionality of a superclass method but forgetting to call the `super()` method.
- For example, to compute the salary of a manager, get the salary of the underlying `Employee` object and add a bonus:

```
class Manager(Employee) :  
    . . .  
    def getSalary(self) :  
        base = self.getSalary()  
        # Error: should be super().getSalary()  
        return base + self._bonus
```

- Here `self` refers to an object of type `Manager` and there is a `getSalary()` method in the `Manager` class.

Common Error 10.2 (2)



- Whenever you call a superclass method from a subclass method with the same name, be sure to use the `super()` function in place of the `self` reference.

```
class Manager(Employee) :  
    . . .  
    def getSalary(self) :  
        base = super().getSalary()  
        return base + self._bonus
```

10.5 Polymorphism

- QuestionDemo2 passed two `ChoiceQuestion` objects to the `presentQuestion()` method
 - Can we write a `presentQuestion()` method that displays both `Question` and `ChoiceQuestion` types?
 - **With inheritance, this goal is very easy to realize!**
 - In order to present a question to the user, we need not know the exact type of the question.
 - We just display the question and check whether the user supplied the correct answer.

```
def presentQuestion(q) :  
    q.display()  
    response = input("Your answer: ")  
    print(q.checkAnswer(response))
```

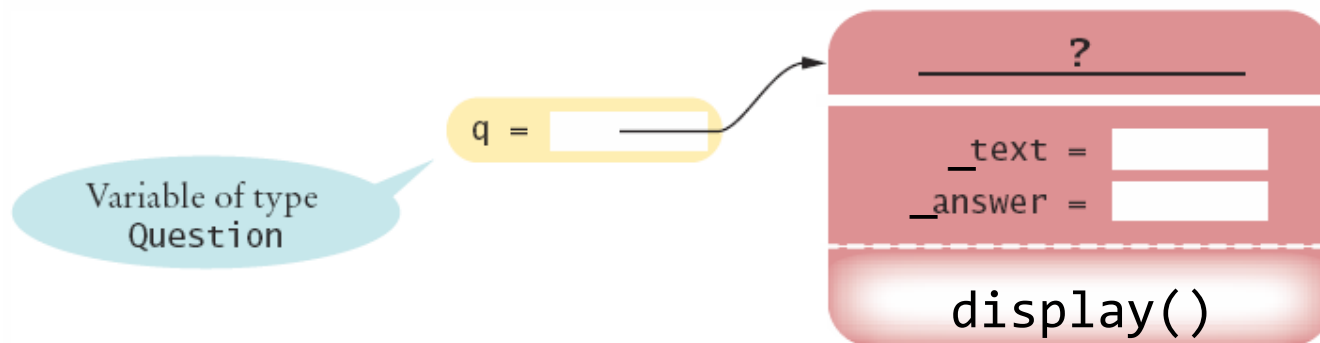
Which Display() method was called?

- `presentQuestion()` simply calls the `display()` method of whatever type is passed:

```
def presentQuestion(q) :  
    q.display()  
    . . .
```

- If passed an object of the Question class:
 - `Question display()`
- If passed an object of the ChoiceQuestion class:
 - `ChoiceQuestion display()`

- The variable `q` does not know the type of object to which it refers:



Why Does This Work?

- As discussed in Section 10.1, we can substitute a subclass object whenever a superclass object is expected:

```
second = ChoiceQuestion()  
presentQuestion(second)    # OK to pass a ChoiceQuestion
```

- Note however you cannot substitute a superclass object when a subclass object is expected.
 - An `AttributeError` exception will be raised.
 - The parent class has fewer capabilities than the child class (you cannot invoke a method on an object that has not been defined by that object's class).

Polymorphism Benefits

- In Python, method calls *are always determined by the type of the actual object*, **not** the type of the variable containing the object reference
 - This is called *dynamic method lookup*
 - Dynamic method lookup allows us to treat objects of different classes in a uniform way
- This feature is called **polymorphism**
- We ask multiple objects to carry out a task, and each object does so in its own way
- Polymorphism makes programs *easily extensible*

Questiondemo3.py (1)

```
1  ##
2  # This program shows a simple quiz with two question types.
3  #
4
5  from questions import Question, ChoiceQuestion
6
7  def main() :
8      first = Question()
9      first.setText("Who was the inventor of Python?")
10     first.setAnswer("Guido van Rossum")
11
12     second = ChoiceQuestion()
13     second.setText("In which country was the inventor of Python born?")
14     second.addChoice("Australia", False)
15     second.addChoice("Canada", False)
16     second.addChoice("Netherlands", True)
17     second.addChoice("United States", False)
18
19     presentQuestion(first)
20     presentQuestion(second)
21
```

Creates an object of the Question class

Creates an object of the ChoiceQuestion class, uses new addChoice() method.

Calls presentQuestion() - next page - passed both types of objects.

Questiondemo3.py (2)

```
22 ## Presents a question to the user and checks the response.
23 # @param q the question
24 #
25 def presentQuestion(q) :
26     q.display()    # Uses dynamic method lookup
27     response = input("Your answer: ")
28     print(q.checkAnswer(response))    # Uses dynamic method lookup
29
30 # Start the program.
31 main()
```

Receives a parameter of
the super-class type

Uses
appropriate
display
method.

Special Topic 10.2



- Subclasses and Instances:
 - You learned that the `isinstance()` function can be used to determine if an object is an instance of a specific class.
 - But the `isinstance()` function can also be used to determine if an object is an instance of a subclass.
 - For example, the function call:
`isinstance(q, Question)`
 - will return True if `q` is an instance of the `Question` class or of any subclass that extends the `Question` class,
 - otherwise, it returns False.

Use of Isinstance()

- ❑ A common use of the `isinstance()` function is to verify that the arguments passed to a function or method are of the correct type.

```
def presentQuestion(q) :  
    if not isinstance(q, Question) :  
        raise TypeError("The argument is not a Question or  
            one of its subclasses.")
```

Special Topic 10.3



- Dynamic Method Lookup
 - Suppose we move the `presentQuestion()` method to inside the `Question` class and call it as follows:

```
cq = ChoiceQuestion()
cq.setText("In which country was the inventor of Python born?")
. . .
cq.presentQuestion()
```

```
def presentQuestion(self) :
    self.display()
    response = input("Your answer: ")
    print(self.checkAnswer(response))
```

- Which `display()` and `checkAnswer()` methods will be called?



Dynamic Method Lookup

- If you look at the code of the `presentQuestion()` method, you can see that these methods are executed on the `self` reference parameter.
 - Because of dynamic method lookup, the `ChoiceQuestion` versions of the `display()` and `checkAnswer()` methods are called automatically.
 - This happens even though the `presentQuestion()` method is declared in the `Question` class, which has no knowledge of the `ChoiceQuestion` class.

```
class Question :  
    def presentQuestion(self) :  
        self.display()  
        response = input("Your answer: ")  
        print(self.checkAnswer(response))
```

Special Topic 10.4



- Abstract Classes and methods
 - If it is desirable to *force* subclasses to override a method of a base class, you can declare a method as **abstract**.
 - You cannot instantiate an object that has **abstract** methods
 - Therefore the class is considered **abstract** (it has 1+ abstract methods)
 - It's a tool to force programmers to create subclasses (avoids the trouble of coming up with useless default methods that others might inherit by accident).
 - In Python, there is no explicit way to specify that a method is an abstract method. Instead, the common practice among Python programmers is to have the method raise a `NotImplementedError` exception as its only statement:

```
class Account :  
    . . .  
    def deductFees(self) :  
        raise NotImplementedError
```


Common Error 10.3



- Don't Use Type Tests
 - Some programmers use specific type tests in order to implement behavior that varies with each class:

```
if isinstance(q, ChoiceQuestion) :    # Don't do this.  
    # Do the task the ChoiceQuestion way.  
elif isinstance(q, Question) :  
    # Do the task the Question way.
```

- This is a poor strategy.
- If a new class such as `NumericQuestion` is added, then you need to revise all parts of your program that make a type test, adding another case:

```
elif isinstance(q, NumericQuestion) :  
    # Do the task the NumericQuestion way.
```

Alternate to Type Tests

- Polymorphism
 - Whenever you find yourself trying to use type tests in a hierarchy of classes, reconsider and use polymorphism instead.
 - Declare a method `doTheTask()` in the superclass, override it in the subclasses, and call

```
q.doTheTask()
```

Steps to Using Inheritance

- As an example, we will consider a bank that offers customers the following account types:
 - 1) A savings account that earns interest. The interest compounds monthly and is based on the minimum monthly balance.
 - 2) A checking account that has no interest, gives you three free withdrawals per month, and charges a \$1 transaction fee for each additional withdrawal.
- The program will manage a set of accounts of both types
 - It should be structured so that other account types can be added without affecting the main processing loop.
- The menu: `D)eposit W)ithdraw M)onth end Q)uit`
 - For deposits and withdrawals, query the account number and amount. Print the balance of the account after each transaction.
 - In the “Month end” command, accumulate interest or clear the transaction counter, depending on the type of the bank account. Then print the balance of all accounts.

Steps to Using Inheritance

1) List the classes that are part of the hierarchy.

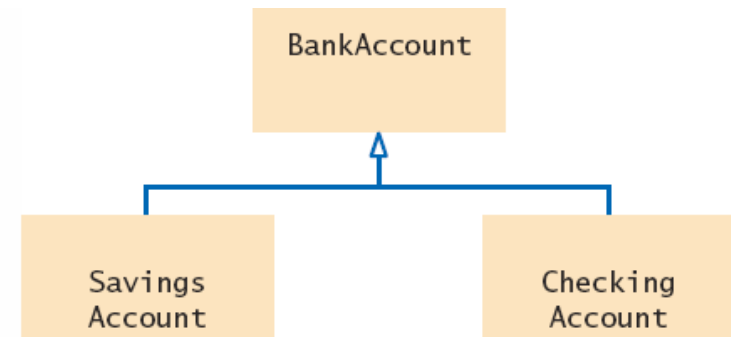
SavingsAccount

CheckingAccount

2) Organize the classes into an inheritance.

hierarchy

Base on superclass BankAccount



3) Determine the common responsibilities.

a. Write *Pseudocode* for each task

b. Find common tasks

Using Inheritance: Pseudocode

For each user command

 If it is a deposit or withdrawal

 Deposit or withdraw the amount from the specified account.

 Print the balance.

 If it is month end processing

 For each account

 Call month end processing.

 Print the balance.

Deposit money.

Withdraw money.

Get the balance.

Carry out month end processing.

Steps to Using Inheritance

4) Decide which methods are overridden in subclasses.

- For each subclass and each of the common responsibilities, decide whether the behavior can be inherited or whether it needs to be overridden

5) Declare the public interface of each subclass.

- Typically, subclasses have responsibilities other than those of the superclass. List those, as well as the methods that need to be overridden.
- You also need to specify how the objects of the subclasses should be constructed.

6) Identify instance variables.

- List the instance variables for each class. Place instance variables that are common to all classes in the base of the hierarchy.

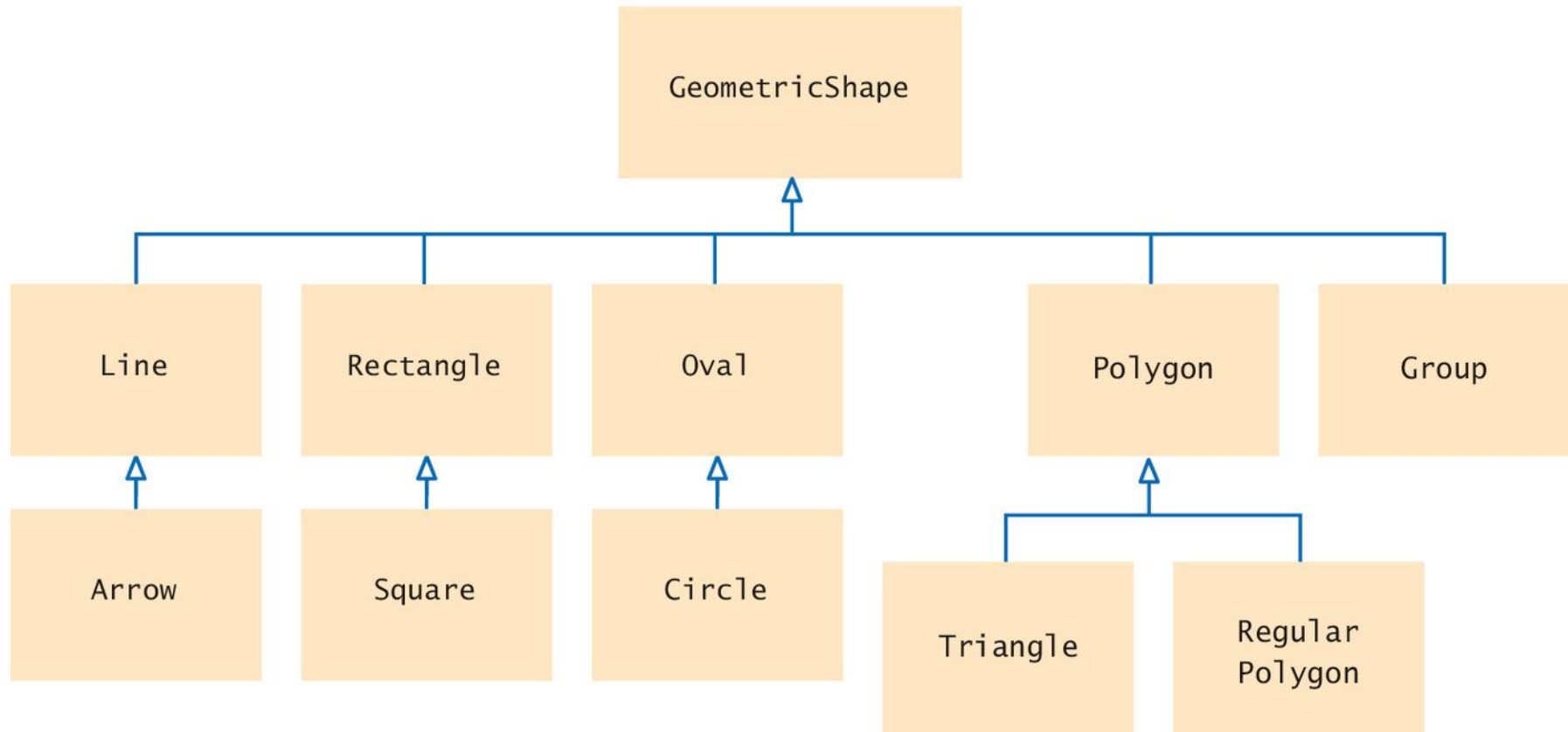
7) Implement constructors and methods.

8) Construct objects of different subclasses and process them.

10.6 Application

- Creating a geometric shape class hierarchy
 - To create complex scenes beyond the simple graphics introduced in Chapter 2, you may need a large number of shapes that vary in color, size, or location.
 - Rather than calling the various methods again and again, it would be useful to have classes that model the various geometric shapes.
 - Using shape classes, a programmer can create a shape object with specific characteristics, then use the same object to draw multiple instances of the shape with only minor changes.

Inheritance Diagram of Geometric Shapes



The Base Class

- The `GeometricShape` class should provide the functionality that is common among the various subclasses:
 - Setting the colors used to draw the shape.
 - Getting and setting the coordinates for the upper-left corner of a bounding box.
 - Computing the width and height of the shape (or the bounding box used to define the shape).
 - Drawing the shape on a canvas.
- Due to the amount of variation between shapes all subclasses will have to override the `draw()` method.