Chapter 12 Inheritance and Class Design

Objectives

- To develop a subclass from a superclass through inheritance (§12.2).
- To override methods in the subclass (§12.3).
- To explore the **object** class and its methods (§12.4).
- To understand polymorphism and dynamic binding (§12.5).
- To determine if an object is an instance of a class using the **isinstance** function (§12.6).
- To discover relationships among classes (§12.8).
- To design classes using composition and inheritance relationships (§§12.9-12.11).

Python Inheritance

- Inheritance allows us to define a class that inherits all the methods and properties from another class.
- **Parent class** is the class being inherited from, also called base class.
- **Child class** is the class that inherits from another class, also called derived class.

Base or Parent Class: Person

- Create a class named Person, with firstname and lastname properties, and a printname method:

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Child Class : Student

- To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class:

```
class Student(Person):
  pass
```

Use the pass keyword when you do not want to add any other properties or methods to the class.

- Use the Student class to create an object, and then execute the printname method:

```
x = Student("Mike", "Olsen")
x.printname()
```

Child Class : Student

- Add a property called graduationyear and a method called welcome to the Student class:

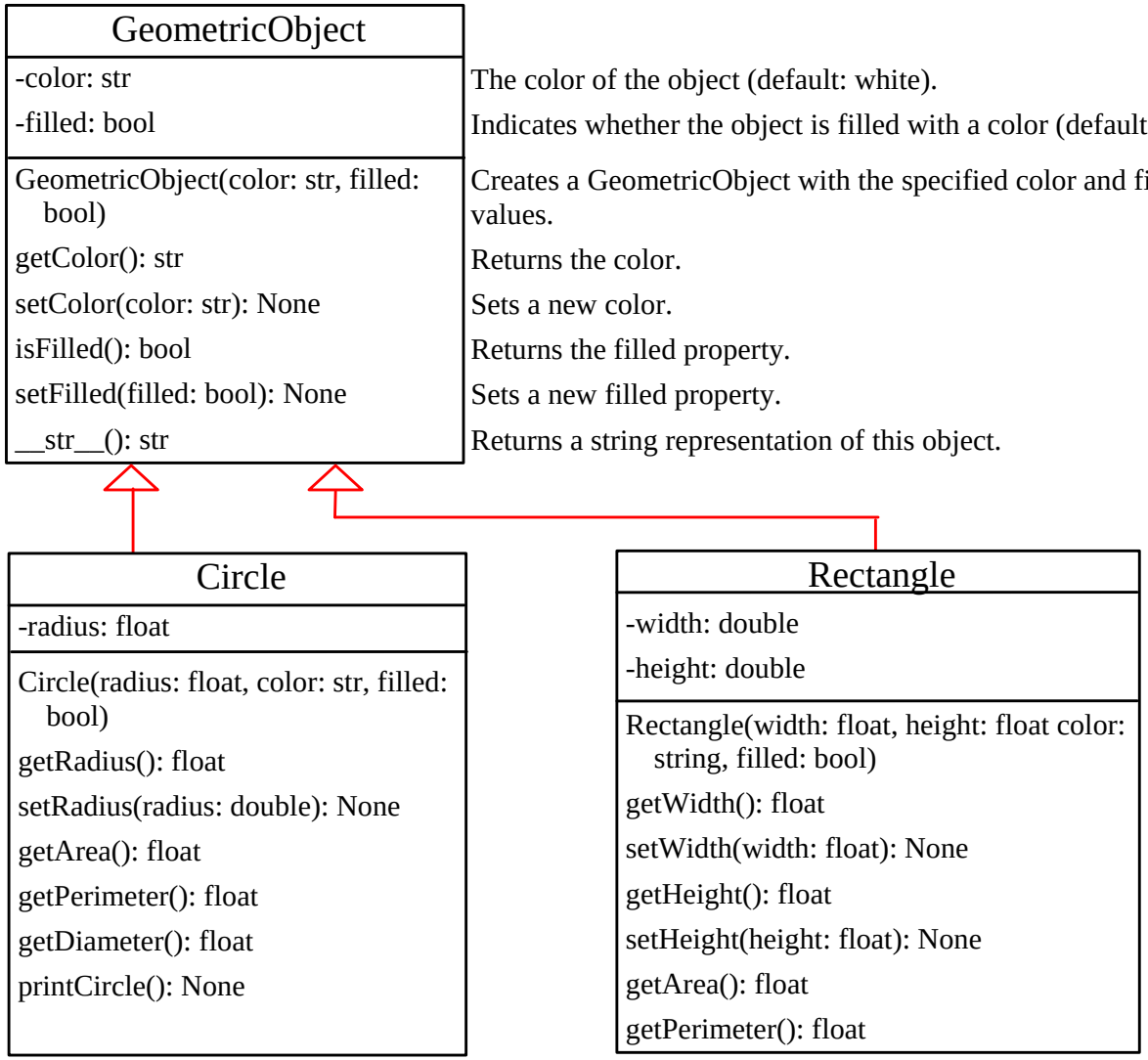
```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname, self.lastname, "to the class of", self.graduationyear)
```

super() function that will make the child class inherit all the methods and properties from its parent:

Example

Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy? The answer is to use inheritance.

Superclasses and Subclasses



GeometricObject

Circle

Rectangle

TestCircleRectangle

Run

Overriding Methods

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

```
class Circle(GeometricObject):
    # Other methods are omitted
    # Override the __str__ method defined in GeometricObject
    def __str__(self):
        return super().__str__() + " radius: " + str(radius)
```

The object Class

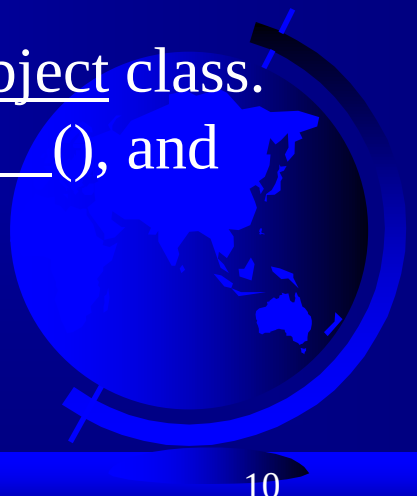
- Every class in Python is descended from the object class.
- If no inheritance is specified when a class is defined, the superclass of the class is object by default.

```
class ClassName:
    ...
```

Equivalent

```
class ClassName(object):
    ...
```

There are more than a dozen methods defined in the object class. We discuss four methods __new__(), __init__(), __str__(), and __eq__(other) here.



The __new__, __init__ Methods

All methods defined in the object class are special methods with two leading underscores and two trailing underscores.

- The __new__() method is automatically invoked when an object is constructed.
- This method then invokes the __init__() method to initialize the object.
- Normally you should only override the __init__() method to initialize the data fields defined in the new class.

The __str__ Method

The __str__() method returns a string representation for the object. By default, it returns a string consisting of a class name of which the object is an instance and the object's memory address in hexadecimal.

```
def __str__(self):
      return "color: " + self.__color + \
          " and filled: " + str(self.__filled)
```

The __eq__ Method

The __eq__(other) method returns True if two objects are the same. By default, x.__eq__(y) (i.e., x == y) returns False, but x.__eq__(x) is True. You can override this method to return True if two objects have the same contents.

Polymorphism

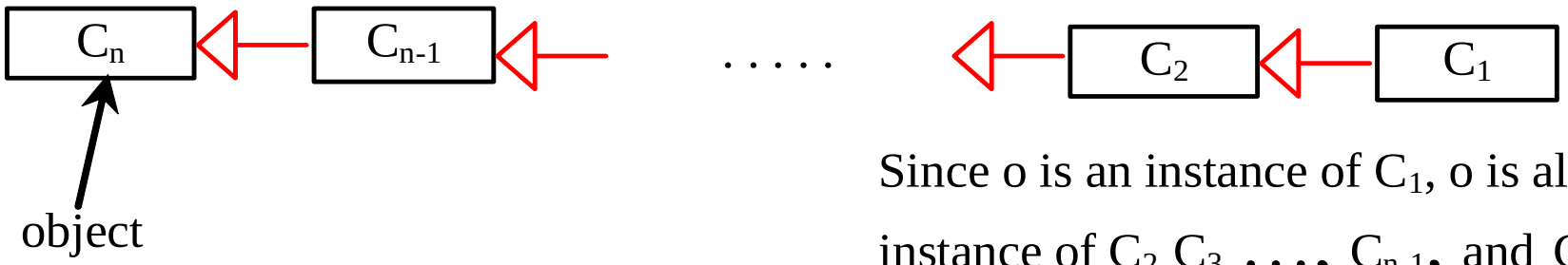
The three pillars of object-oriented programming are *encapsulation*, *inheritance*, and *polymorphism*.

- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.
- A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa.
- For example, every circle is a geometric object, but not every geometric object is a circle. Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.

PolymorphismDemo Run

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class. In Python, C_n is the object class. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



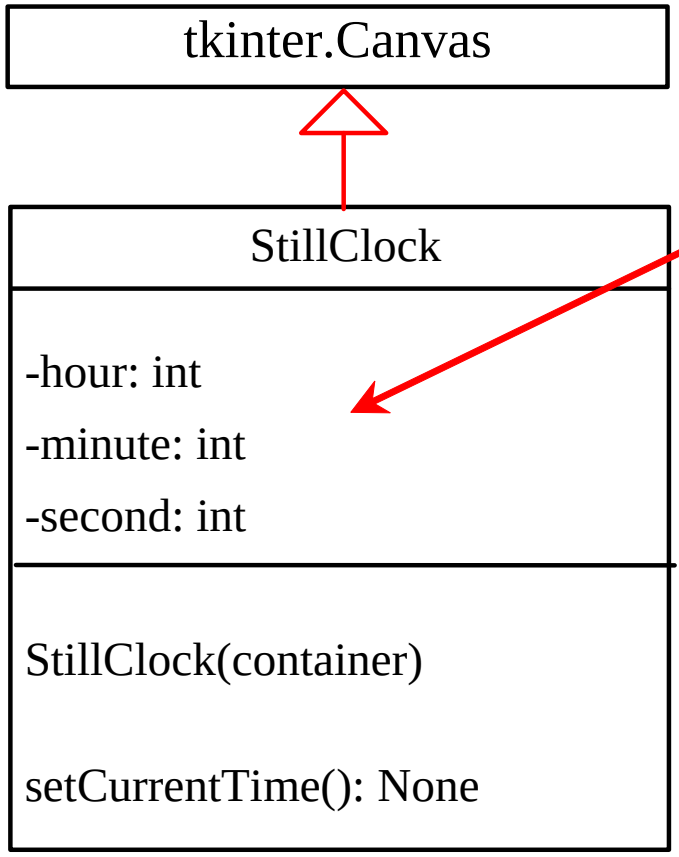
The isinstance Function

The isinstance function can be used to determine if an object is an instance of a class.

IsinstanceDemo

Run

Case Study: A Reusable Clock



The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

The hour in the clock.

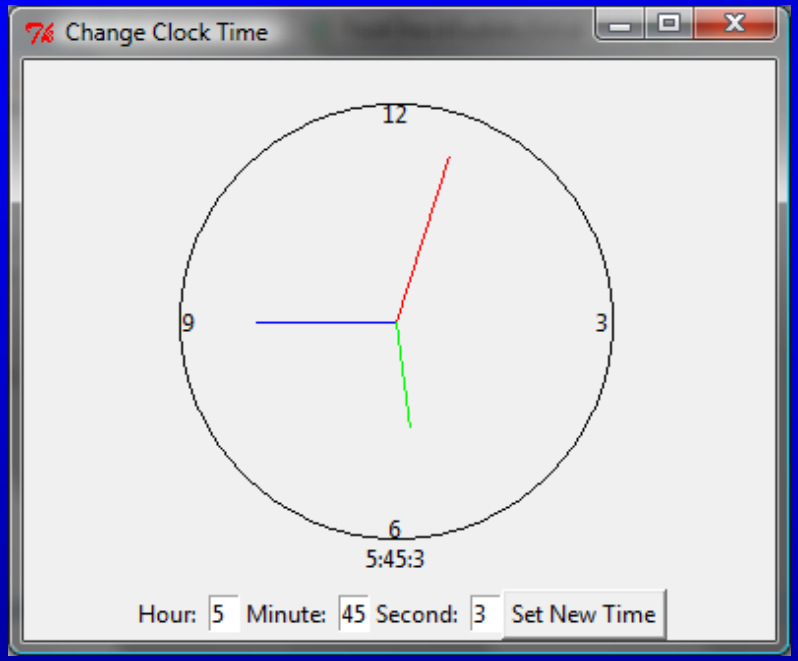
The minute in the clock.

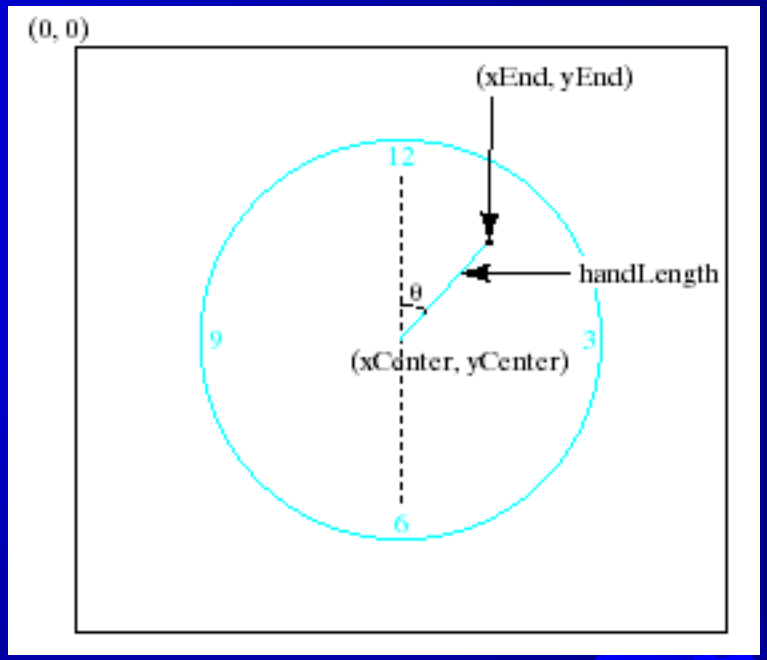
The second in the clock.

Constructs a default clock for the current time, placed inside a container.

Sets hour, minute, and second to current time.

Case Study: A Reusable Clock





Still Clock

DisplayClock

Run

Relationships among Classes

- Association
- Aggregation
- Composition
- Inheritance

Association

Association represents a general binary relationship that describes an activity between two classes.


```
class Student:
  def addCourse(self,
      course):
  # add course to a list
```

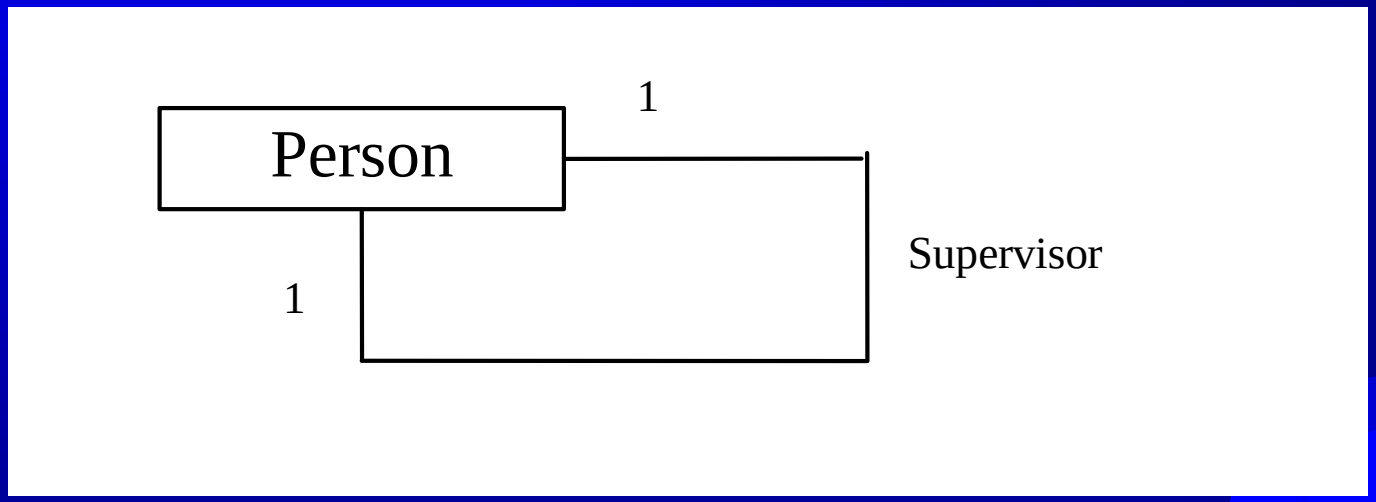
```
class Course:

  def addStudent(self,
      student):
    # add student to a list

  def setFaculty(self, faculty):
    # Code omitted
```

```
class Faculty:
  def addCourse(self,
     course):
     # add course to a list
```

An association is usually represented as a data field in the class.

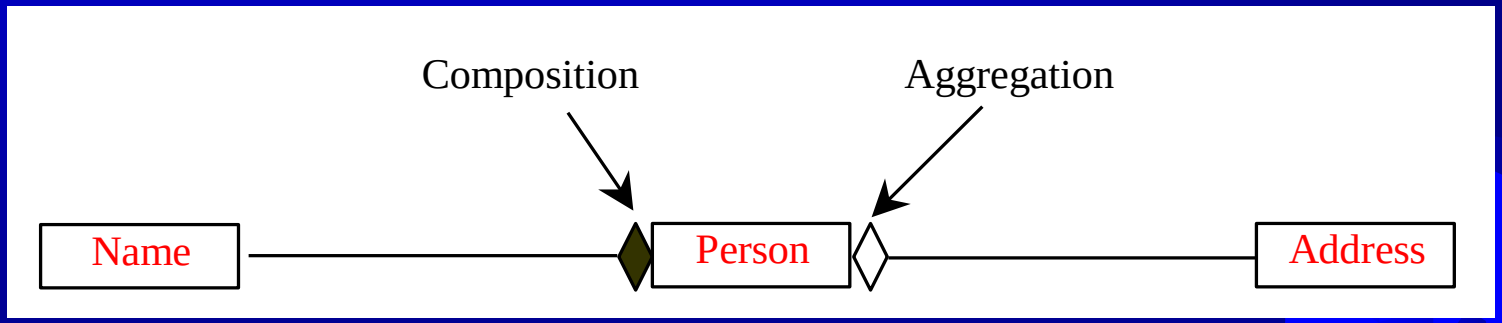
Association Between Same Class

Association may exist between objects of the same class. For example, a person may have a supervisor.



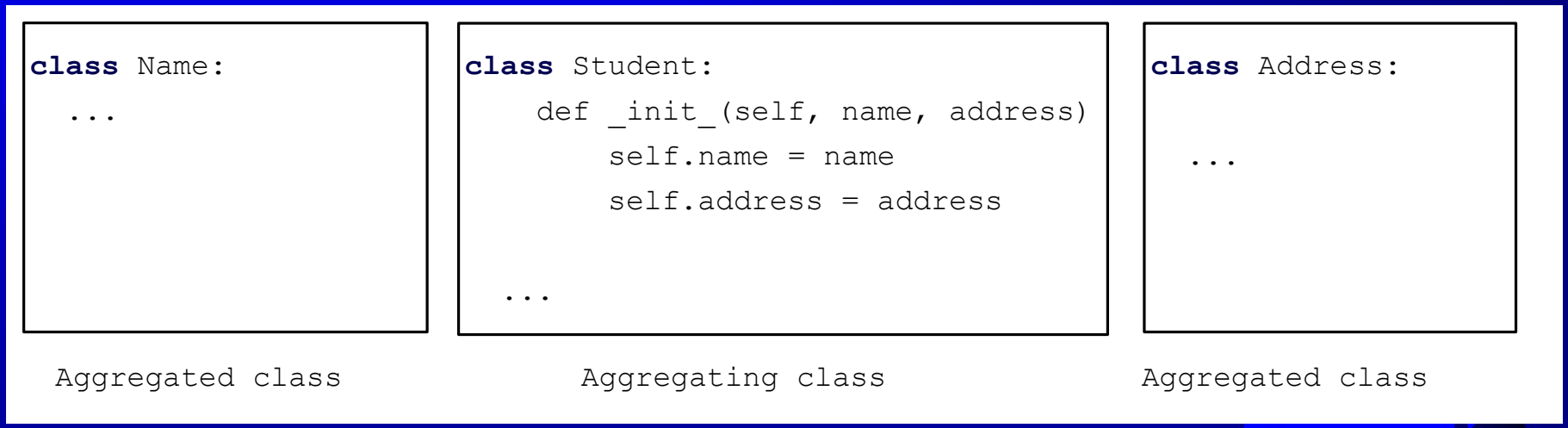
Aggregation and Composition

Aggregation is a special form of association, which represents an ownership relationship between two classes. Aggregation models the has-a relationship. If an object is exclusively owned by an aggregated object, the relationship between the object and its aggregated object is referred to as *composition*.



Representing Aggregation in Classes

An aggregation relationship is usually represented as a data field in the aggregated class.

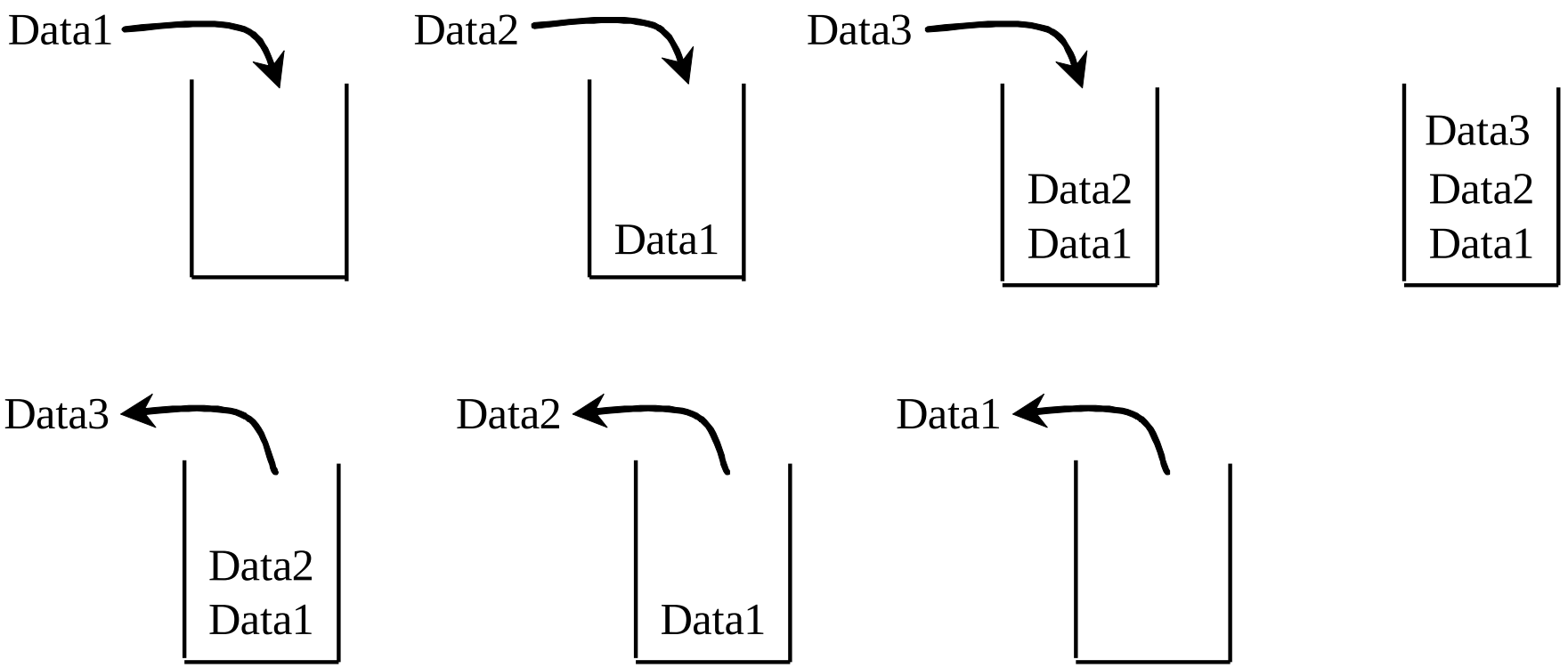


The Course Class

Course	
-courseName: str	The name of the course.
-students: list	An array to store the students for the course.
Course(courseName: str)	Creates a course with the specified name.
getCourseName(): str	Returns the course name.
addStudent(student: str): None	Adds a new student to the course.
dropStudent(student: str): None	Drops a student from the course.
getStudents(): list	Returns the students for the course.
getNumberOfStudents(): int	Returns the number of students for the course.

Course TestCourse Run

The Stack Class

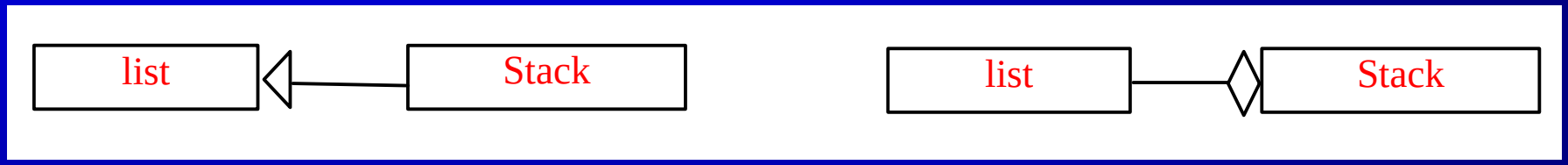


The Stack Class

You can define a class to model stacks. You can use a list to store the elements in a stack. There are two ways to design the stack and queue classes:

-Using inheritance: You can define a stack class by extending list.

-Using composition: You can create a list as a data field in the stack class.



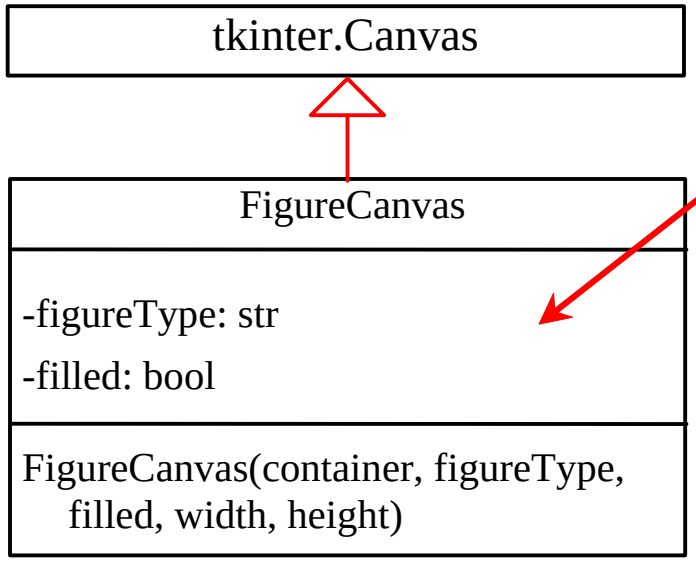
Both designs are fine, but using composition is better because it enables you to define a completely new stack class without inheriting the unnecessary and inappropriate methods from the list class.

The Stack Class

Stack	
-elements: list	A list to store elements in the stack.
+Stack()	Constructs an empty stack.
+isEmpty(): bool	Returns True if the stack is empty.
+peek(): object	Returns the element at the top of the stack without removing it from the stack.
+push(value: object): None	Stores an element into the top of the stack.
+pop(): object	Removes the element at the top of the stack and returns it.
+getSize(): int	Returns the number of elements in the stack.

Stack TestStack Run

The FigureCanvas Class



The get and set methods for these data fields are provided in the class, but omitted in the UML diagram for brevity.

Specifies the figure type (line, rectangle, oval, or arc).

Specifies whether the figure is filled (default: False).

Creates a figure canvas inside a container with the specified type, filled, width (default 200), and height (default 200).

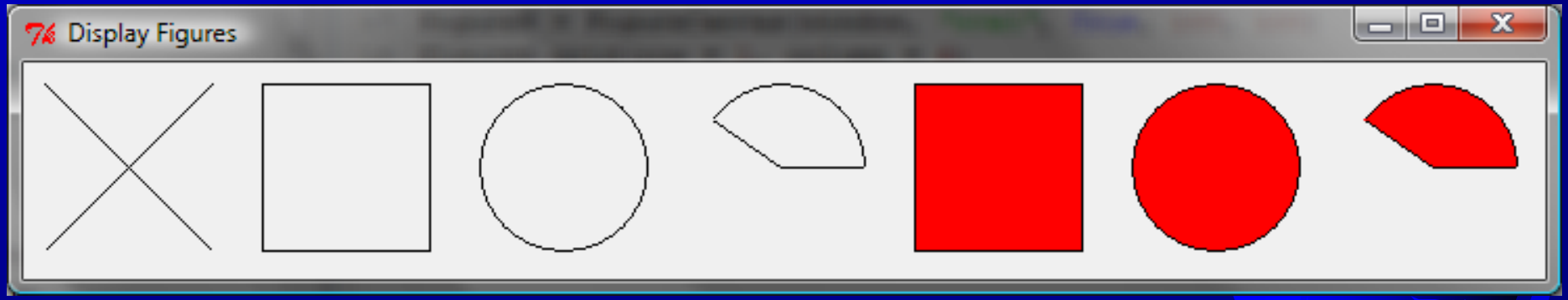


Figure Canvas DisplayFigures Run