

1

## Objects

object = state + behavior

“An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class.”

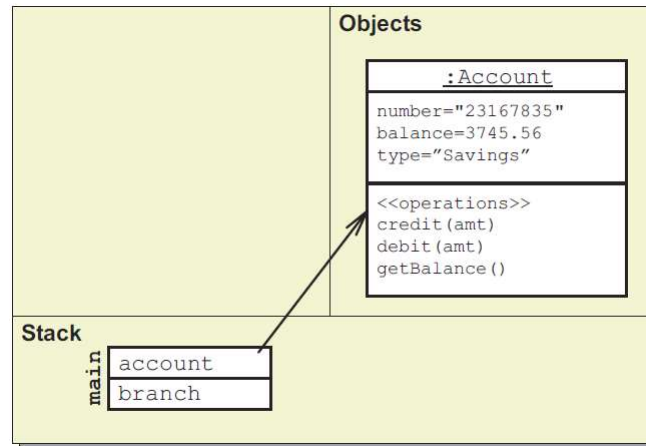
(Booch Object Solutions page 305)

Objects:

- > Have identity
- > Are an instance of only one class
- > Have attribute values that are unique to that object
- > Have methods that are common to the class

2

## Objects: Example



3

## Classes

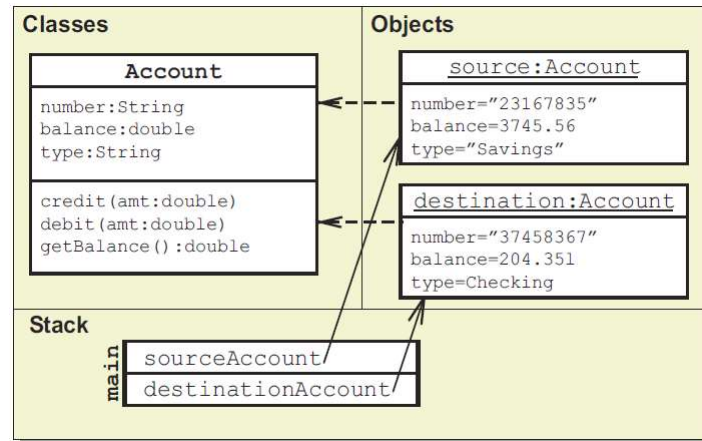
A class is a blueprint or prototype from which objects are created. (The Java™ Tutorials)

Classes provide:

- > The metadata for attributes
- > The signature for methods
- > The implementation of the methods (usually)
- > The constructors to initialize attributes at creation time

4

## Classes: Example



5

## Abstraction

In OO software, the concept of abstraction enables you to create a simplified, but relevant view of a real world object within the context of the problem and solution domains.

- > The abstraction object is a representation of the real world object with irrelevant (within the context of the system) behavior and data removed.
- > The abstraction object is a representation of the real world object with currently irrelevant (within the context of the view) behavior and data hidden.

6

## Abstraction: Example

Engineer	Engineer
fname:String lname:String salary:Money	fname:String lname:String salary:Money fingers:int toes:int hairColor:String politicalParty:String
increaseSalary(amt) designSoftware() implementCode()	increaseSalary(amt) designSoftware() implementCode() eatBreakfast() brushHair() vote()

7

## Encapsulation

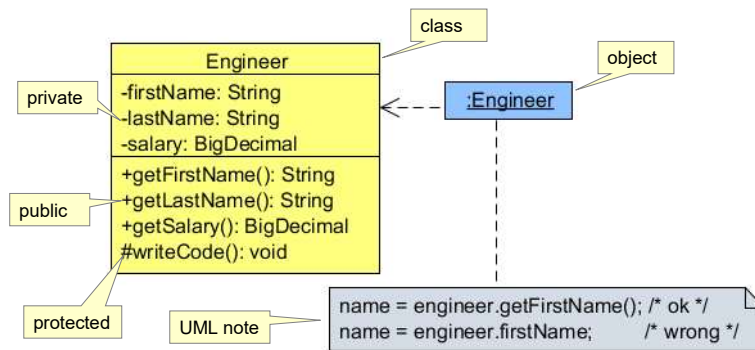
Encapsulation means “to enclose in or as if in a capsule” (Webster New Collegiate Dictionary)

- > Encapsulation is essential to an object. An object is a capsule that holds the object’s internal state within its boundary.
- > In most OO languages, the term encapsulation also includes *information hiding*, which can be defined as: “hide implementation details behind a set of non-private methods”.

8

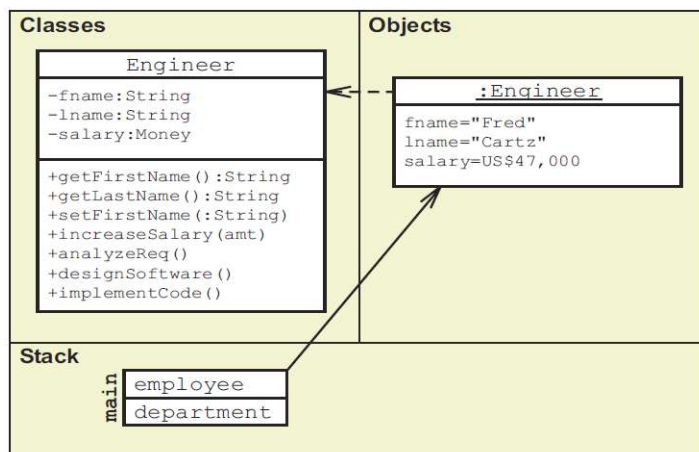
# Encapsulation

## > Black-Box Design



9

## Encapsulation: Example



✗ `name = employee.fname;`
✓ `name = employee.getFirstName();`  
✗ `employee.fname = "Samantha";`
✓ `employee.setFirstName("Samantha");`

10

## Defining a Class

```
class InsufficientBalance(Exception):
    def __init__(self, *args):
        if args:
            self.message = args[0]
            self.deficit = args[1]
        else:
            self.message = None
            self.deficit = 0.0

    def __str__(self):
        if self.message:
            return f"InsufficientBalance[ message: {self.message}, deficit: {self.deficit}"];
```

11

## Defining a Class

```
class Account:
    def __init__(self, iban, balance=100.0):
        self.iban = iban
        self.balance = balance

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > self.balance:
            raise InsufficientBalance("balance is less than amount.", amount-self.balance)
        self.balance -= amount

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        self.balance += amount

    def __str__(self):
        return f"Account[ iban: {self.iban}, balance: {self.balance}"];
```

12

## Defining a Class

```
try:
    acc = Account("TR1", 100000.0)
    print(acc)
    acc.withdraw(2500.0)
    print(acc)
    acc.deposit(1500.0)
    print(acc)
    acc.withdraw(100000.0)
    print(acc)
except ValueError as err:
    print(err)
except InsufficientBalance as err:
    print(err)
```

```
Account[ iban: TR1, balance: 100000.0]
Account[ iban: TR1, balance: 97500.0]
Account[ iban: TR1, balance: 99000.0]
InsufficientBalance[ message: balance is less than amount., deficit: 1000.0]
```

13

## \_\_init\_\_ Method with Default Parameter Values

```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour= hour
        self.minute= minute
        self.second= second
```

```
t1 = Time()
```

```
print(f'{t1.hour}h:{t1.minute}m:{t1.second}s')
```

```
0h:0m:0s
```

14

## Read-Write Property

```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self._hour = hour
        self._minute = minute
        self._second = second

    @property
    def hour(self):
        return self._hour

    @hour.setter
    def hour(self, hour):
        if not (0 <= hour < 24):
            raise ValueError(f'Hour ({hour}) must be 0-23')
        self._hour = hour

    @property
    def minute(self):
        return self._minute

    @property
    def second(self):
        return self._second
```

15

## Read-Write Property

In [3]: `wake_up = Time()`

In [4]: `wake_up.hour=10`

In [5]: `wake_up.hour`

Out[5]: 10

In [8]: `wake_up.second`

Out[8]: 0

In [9]: `wake_up.second = 20`

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-9-0fa505464fda> in <module>
----> 1 wake_up.second = 20

AttributeError: can't set attribute
```

16

## Special Method `__repr__`

```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self._hour = hour
        self._minute = minute
        self._second = second

    @property
    def hour(self):
        return self._hour

    @hour.setter
    def hour(self, hour):
        if not (0 <= hour < 24):
            raise ValueError(f'Hour ({hour}) must be 0-23')
        self._hour = hour

    @property
    def minute(self):
        return self._minute

    @property
    def second(self):
        return self._second

    def __repr__(self):
        return f'Time(hour={self.hour}, minute={self.minute}, second={self.second})'

    def __str__(self):
        return f'Time (hour={self.hour}, minute={self.minute}, second={self.second})'
```

17

## Information Hiding

```
class Account:
    def __init__(self, iban, balance=100.0):
        self.__iban = iban
        self.__balance = balance

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > self.__balance:
            raise InsufficientBalance("balance is less than amount.", amount-self.__balance)
        self.__balance -= amount

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        self.__balance += amount

    def __str__(self):
        return f"Account[ iban: {self.__iban}, balance: {self.__balance}]"
```

18

```

try:
    acc = Account("TR1", 100000.0)
    print(acc.__balance)
    print(acc.__iban)
    acc.withdraw(2500.0)
    print(acc)
    acc.balance = 10000000000
    acc.withdraw(100000.0)
    print(acc)
    acc.deposit(1500.0)
    print(acc)
except ValueError as err:
    print(err)
except InsufficientBalance as err:
    print(err)

```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-49-724a42ddf06c> in <module>
      1 try:
      2     acc = Account("TR1", 100000.0)
----> 3     print(acc.__balance)
      4     print(acc.__iban)
      5     acc.withdraw(2500.0)

AttributeError: 'Account' object has no attribute '__balance'

```

19

## Inheritance

Inheritance is “a mechanism whereby a class is defined in reference to others, adding all their features to its own.” (Meyer page 1197)

Features of inheritance:

- > Attributes and methods from the superclass are included in the subclass.
- > Subclass methods can override superclass methods.
- > The following conditions must be true for the inheritance relationship to be plausible:
  - A subclass object *is a (is a kind of)* the superclass object.
  - Inheritance should conform to Liskov’s Substitution Principle (LSP).

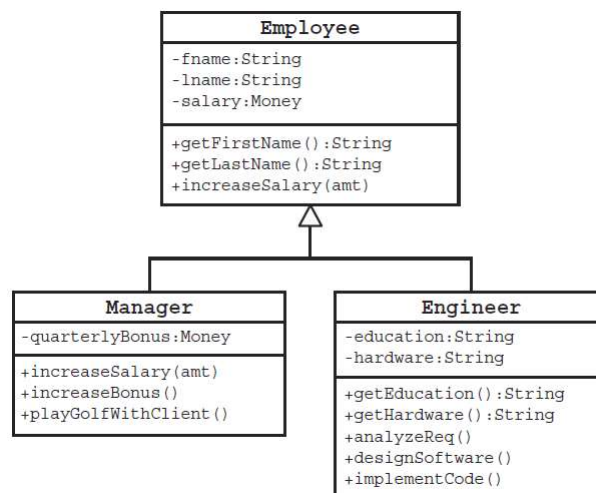
20

## Inheritance

- > Specific OO languages allow either of the following:
  - Single inheritance, which allows a class to directly inherit from only one superclass (for example, Java).
  - Multiple inheritance, which allows a class to directly inherit from one or more super-classes (for example, C++, Python).

21

## Inheritance: Example



22

## Inheritance: Example

```
class Account:
    def __init__(self, iban, balance=100.0):
        self.iban = iban
        self.balance = balance

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > self.balance:
            raise InsufficientBalance("balance is less than amount.", amount-self.balance)
        self.balance -= amount

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        self.balance += amount

    def __str__(self):
        return f"Account[ iban: {self.iban}, balance: {self.balance}];"
```

23

## Inheritance: Example

```
class CheckingAccount(Account):
    def __init__(self, iban, balance, overdraft_amount):
        super().__init__(iban, balance)
        self.overdraft_amount = overdraft_amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > (self.balance+self.overdraft_amount):
            raise InsufficientBalance("balance is less than amount.", amount-self.balance-self.overdraft_amount)
        self.balance -= amount
```

```
acc2 = CheckingAccount('TR2', 1000, 500)
```

```
acc2.withdraw(1250)
```

```
acc2.balance
```

```
-250
```

24

## Inheritance: Example

```
class CheckingAccount(Account):
    def __init__(self, iban, balance, overdraft_amount):
        super().__init__(iban, balance)
        self.overdraft_amount = overdraft_amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("amount must be positive.")
        if amount > (self.balance + self.overdraft_amount):
            raise InsufficientBalance("balance is less than amount.", amount - self.balance - self.overdraft_amount)
        self.balance -= amount

acc2 = CheckingAccount('TR2', 1000, 500)

acc2.withdraw(1250)

acc2.balance
```

25

## Testing the “is a” Relationship

- > Python provides two built-in functions—**issubclass** and **isinstance**—for testing “is a” relationships.
- > Function **issubclass** determines whether one class is derived from another
- > Function **isinstance** determines whether an object has an “is a” relationship with a specific type.

26

## Testing the “is a” Relationship

```
In [41]: acc1 = Account("TR1", 2000)

In [42]: isinstance(acc1, Account)
Out[42]: True

In [43]: isinstance(acc2, Account)
Out[43]: True

In [44]: isinstance(acc1, CheckingAccount)
Out[44]: False

In [46]: isinstance(CheckingAccount, Account)
Out[46]: True
```

27

## Polymorphism

```
accounts = [ Account("TR1", 1000.0),
              CheckingAccount("TR2", 2000.0, 500),
              Account("TR3", 3000.0),
              CheckingAccount("TR4", 4000.0, 1000) ]
```

```
for account in accounts:
    print(account)
    account.withdraw(100)
    print(account)
```

```
Account[ iban: TR1, balance: 1000.0]
CheckingAccount::withdraw
Account[ iban: TR1, balance: 900.0]
CheckingAccount(iban=TR2, balance=2000.0, overdraft_amount=500)
CheckingAccount::withdraw
CheckingAccount(iban=TR2, balance=1900.0, overdraft_amount=500)
Account[ iban: TR3, balance: 3000.0]
CheckingAccount::withdraw
Account[ iban: TR3, balance: 2900.0]
CheckingAccount(iban=TR4, balance=4000.0, overdraft_amount=1000)
CheckingAccount::withdraw
CheckingAccount(iban=TR4, balance=3900.0, overdraft_amount=1000)
```

28

## Operator Overloading

- > Method-call notation can be cumbersome for certain kinds of operations, such as arithmetic.
- > In these cases, it would be more convenient to use Python's rich set of built-in operators.

29

## Restrictions on operator overloading (1/2)

- > There are some restrictions on operator overloading:
  - The precedence of an operator cannot be changed by overloading.
    - However, parentheses can be used to force evaluation order in an expression.
  - The left-to-right or right-to-left grouping of an operator cannot be changed by overloading.
  - The “arity” of an operator—that is, whether it's a unary or binary operator—cannot be changed.
  - You cannot create new operators—only existing operators can be overloaded.
  - The meaning of how an operator works on objects of built-in types cannot be changed. You cannot, for example, change + so that it subtracts two integers.

30

## Restrictions on operator overloading (2/2)

- > There are some restrictions on operator overloading:
  - Operator overloading works only with objects of custom classes or with a mixture of an object of a custom class and an object of a built-in type.

31

## Operator Overloading Example: Fraction

```
import math
class fraction:
    def __init__(self, nominator : int , denominator : int):
        scale = math.gcd(nominator,denominator)
        self.nominator = int(nominator / scale)
        self.denominator = int(denominator / scale)

    def __add__(self, right):
        nominator = self.nominator * right.denominator + self.denominator * right.nominator
        denominator = self.denominator * right.denominator
        return fraction(nominator, denominator)

    def __sub__(self, right):
        nominator = self.nominator * right.denominator - self.denominator * right.nominator
        denominator = self.denominator * right.denominator
        return fraction(nominator, denominator)

    def __mul__(self, right):
        nominator = self.nominator * right.nominator
        denominator = self.denominator * right.denominator
        return fraction(nominator, denominator )
```

32

## Operator Overloading Example: **Fraction**

```
def __floordiv__(self, right):
    nominator = self.nominator * right.denominator
    denominator = self.denominator * right.nominator
    return fraction(nominator, denominator )

def __truediv__(self, right):
    nominator = self.nominator * right.denominator
    denominator = self.denominator * right.nominator
    return fraction(nominator, denominator )

def __str__(self):
    return f'Fraction({self.nominator}/{self.denominator})'
```

33

## Operator Overloading Example: **Fraction**

```
x = fraction(1,2)
y = fraction(3,4)
```

```
z = x + y
```

```
print(z)
```

```
Fraction(5/4)
```

```
z = x * y
```

```
print(z)
```

```
Fraction(3/8)
```

```
z = x - y
```

```
print(z)
```

```
Fraction(-1/4)
```

34

## Operator Overloading Example: **Fraction**

```
z = x // y      __floordiv__
```

```
print(z)
```

```
Fraction(2/3)
```

```
z = x / y      __truediv__
```

```
print(z)
```

```
Fraction(2/3)
```

35



36

## Objectives

Upon completion of this module, you should be able to:

- > Describe the important object-oriented (OO) concepts
- > Describe the fundamental OO terminology

37

## Examining Object Orientation

OO concepts affect the whole development process:

- > Humans think in terms of nouns (objects) and verbs (behaviors of objects).
- > With OOSD, both problem and solution domains are modeled using OO concepts.
- > The *Unified Modeling Language* (UML) is a de facto standard for modeling OO software.
- > OO languages bring the implementation closer to the language of mental models. The UML is a good bridge between mental models and implementation.

38

## Examining Object Orientation

*“Software systems perform certain actions on objects of certain types; to obtain flexible and reusable systems, it is better to base their structure on the objects types than on the actions.” (Meyer page vi)*

OO concepts affect the following issues:

- > Software complexity
- > Software decomposition
- > Software costs

39

## Software Complexity

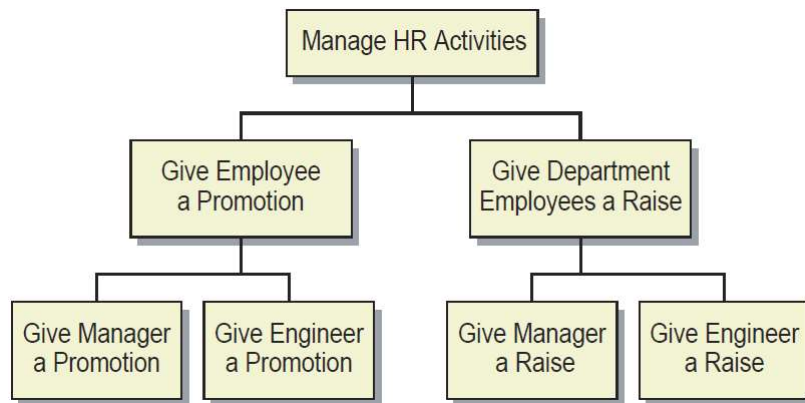
Complex systems have the following characteristics:

- > They have a **hierarchical structure**.
- > The choice of **which components are primitive** in the system is arbitrary.
- > A system can be split by intra- and inter-component relationships. This **separation of concerns** enables you to study each part in relative isolation.
- > Complex systems are usually composed of only a **few types of components in various combinations**.
- > A successful, complex system invariably **evolves from a simple working system**.

40

## Software Decomposition

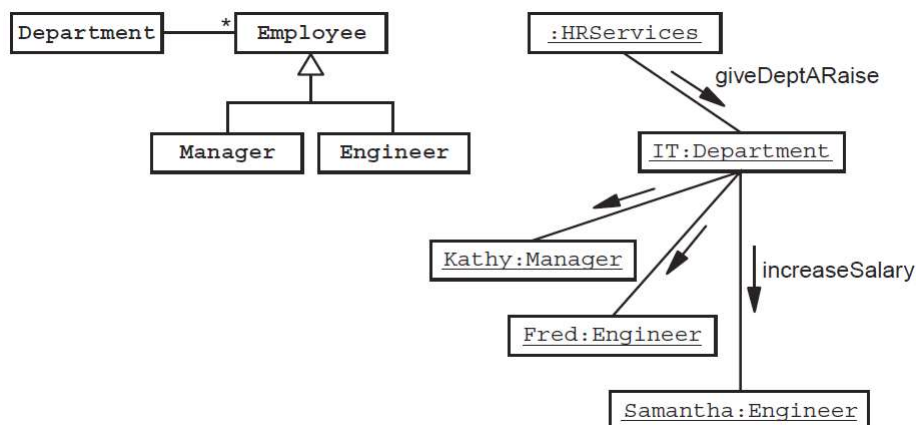
- > In the Procedural paradigm, software is decomposed into a hierarchy of procedures or tasks.



41

## Software Decomposition

- > In the OO paradigm, software is decomposed into a hierarchy of interacting components (usually objects).



42

## Software Costs

### Development:

- > OO principles provide a natural technique for modeling business entities and processes from the early stages of a project.
- > OO-modeled business entities and processes are easier to implement in an OO language.

### Maintenance:

- > Changeability, flexibility, and adaptability of software is important to keep software running for a long time.
- > OO-modeled business entities and processes can be adapted to new functional requirements.

43

## Surveying the Fundamental OO Concepts

- > Objects
- > Classes
- > Abstraction
- > Encapsulation
- > Inheritance
- > Interfaces
- > Polymorphism
- > Cohesion
- > Coupling
- > Class associations and object links
- > Delegation

44

# Objects

object = state + behavior

“An object has state, behavior, and identity; the structure and behavior of similar objects are defined in their common class.”

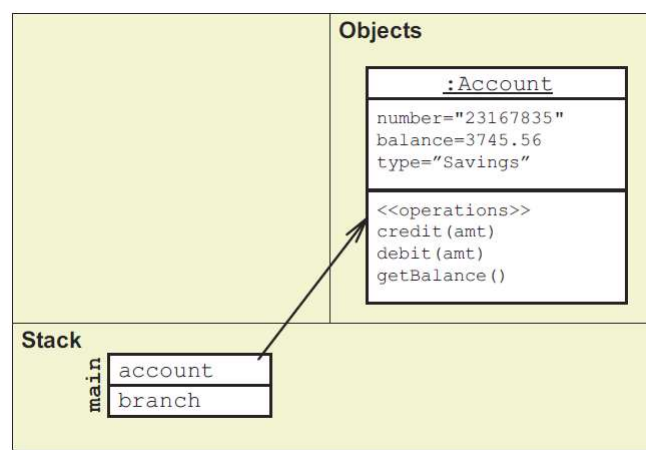
(Booch Object Solutions page 305)

Objects:

- > Have identity
- > Are an instance of only one class
- > Have attribute values that are unique to that object
- > Have methods that are common to the class

45

## Objects: Example



46

## Classes

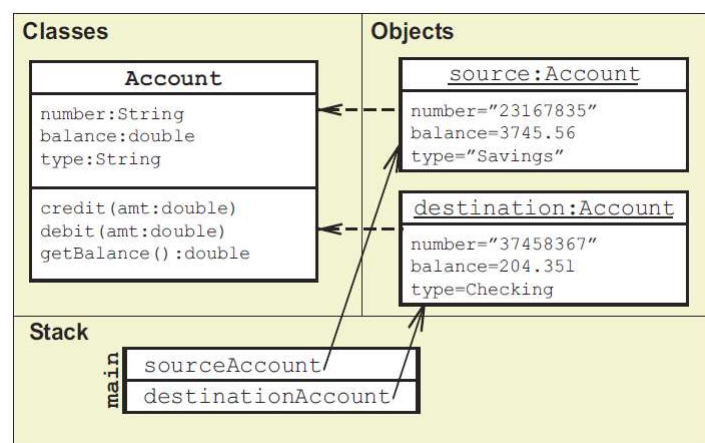
A class is a blueprint or prototype from which objects are created. (The Java™ Tutorials)

Classes provide:

- > The metadata for attributes
- > The signature for methods
- > The implementation of the methods (usually)
- > The constructors to initialize attributes at creation time

47

## Classes: Example



48

## Abstraction

In OO software, the concept of abstraction enables you to create a simplified, but relevant view of a real world object within the context of the problem and solution domains.

- > The abstraction object is a representation of the real world object with irrelevant (within the context of the system) behavior and data removed.
- > The abstraction object is a representation of the real world object with currently irrelevant (within the context of the view) behavior and data hidden.

49

## Abstraction: Example

Engineer	Engineer
fname:String lname:String salary:Money	fname:String lname:String salary:Money fingers:int toes:int hairColor:String politicalParty:String
increaseSalary(amt) designSoftware() implementCode()	increaseSalary(amt) designSoftware() implementCode() eatBreakfast() brushHair() vote()

50

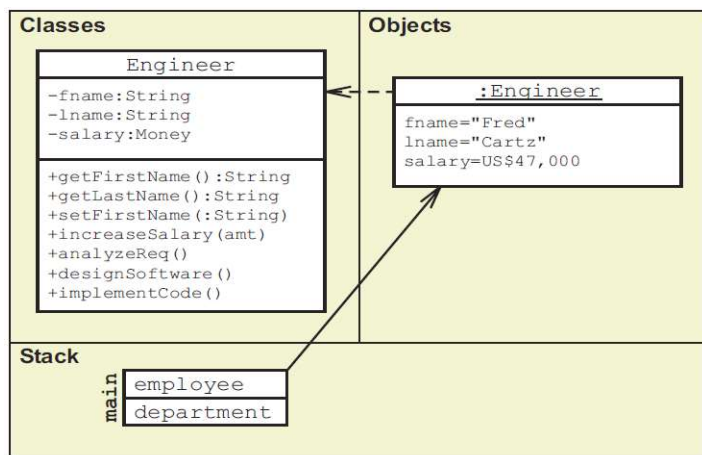
## Encapsulation

Encapsulation means “to enclose in or as if in a capsule” (Webster New Collegiate Dictionary)

- > Encapsulation is essential to an object. An object is a capsule that holds the object’s internal state within its boundary.
- > In most OO languages, the term encapsulation also includes *information hiding*, which can be defined as: “hide implementation details behind a set of non-private methods”.

51

## Encapsulation: Example

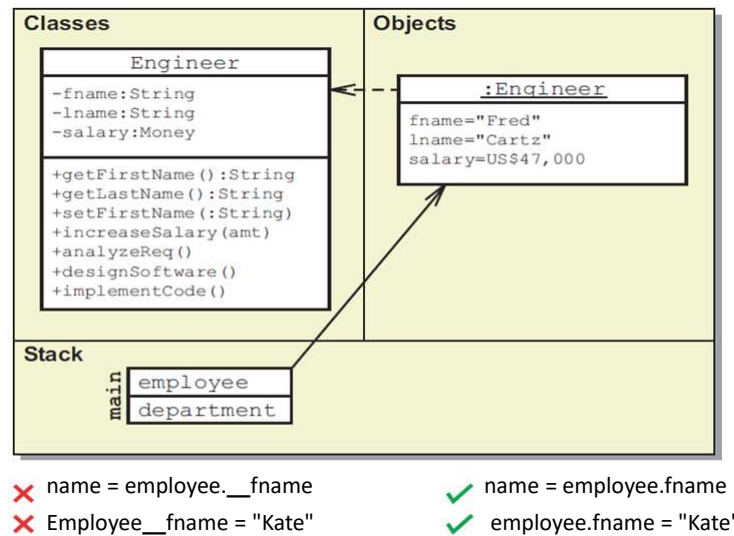


✗ `name = employee.fname;`  
✗ `employee.fname = "Samantha";`

✓ `name = employee.getFirstName();`  
✓ `employee.setFirstName("Samantha");`

52

## Encapsulation: Example



53

## Inheritance

Inheritance is “a mechanism whereby a class is defined in reference to others, adding all their features to its own.” (Meyer page 1197)

Features of inheritance:

- > Attributes and methods from the superclass are included in the subclass.
- > Subclass methods can override superclass methods.
- > The following conditions must be true for the inheritance relationship to be plausible:
  - A subclass object *is a (is a kind of)* the superclass object.
  - Inheritance should conform to Liskov’s Substitution Principle (LSP).

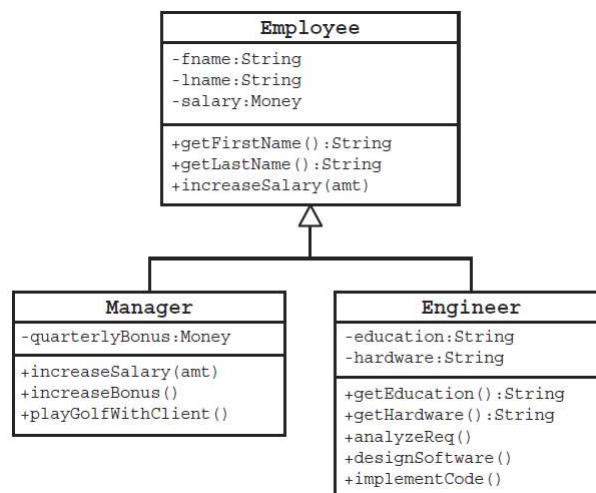
54

## Inheritance

- > Specific OO languages allow either of the following:
  - Single inheritance, which allows a class to directly inherit from only one superclass (for example, Java).
  - Multiple inheritance, which allows a class to directly inherit from one or more superclasses (for example, C++).

55

## Inheritance: Example



56

## Abstract Classes

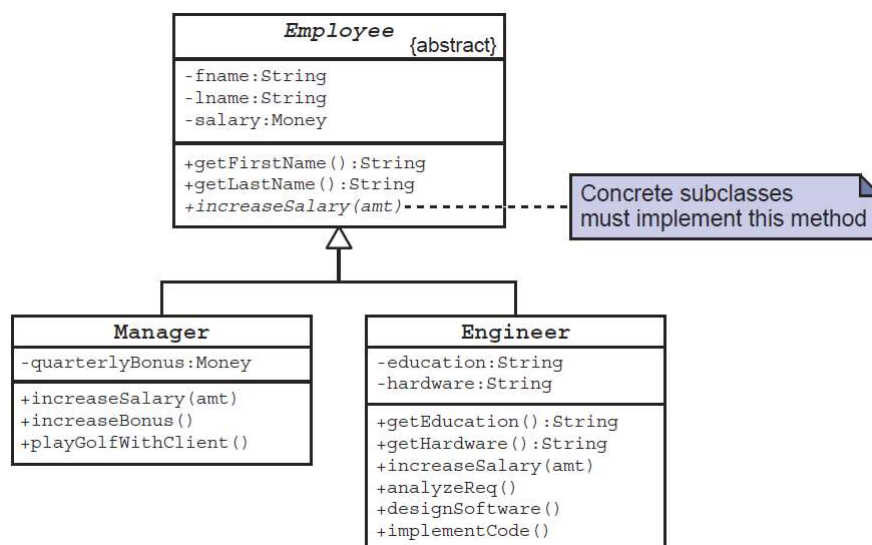
A class that contains one or more abstract methods, and therefore can never be instantiated. (Sun Glossary)

### Features of an abstract class:

- > Attributes are permitted.
- > Methods are permitted and some might be declared abstract.
- > Constructors are permitted, but no client may directly instantiate an abstract class.
- > Subclasses of abstract classes must provide implementations of all abstract methods; otherwise, the subclass must also be declared abstract.
- > In the UML, a method or a class is denoted as abstract by using italics, or by appending the method name or class name with **{abstract}**.

57

## Abstract Classes: Example



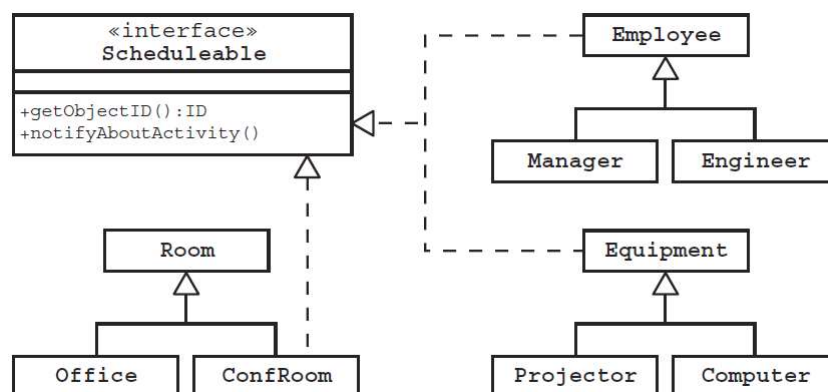
58

## Interfaces

- > Features of Java technology interfaces:
  - Attributes are not permitted (except constants).
  - Methods are permitted, but they must be abstract.
  - Constructors are not permitted.
  - Subinterfaces may be defined, forming an inheritance hierarchy of interfaces.
- > A class may implement one or more interfaces.

59

## Interfaces: Example



60

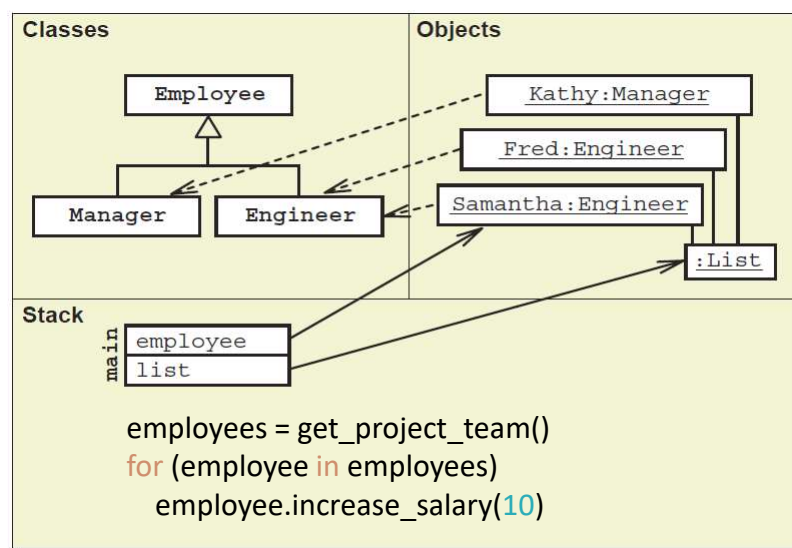
## Polymorphism

Polymorphism is “a concept in type theory, according to which a name (such as a variable declaration) may denote objects of many different classes that are related by some common superclass [type].” (Booch OOAD page 517)

- > Aspects of polymorphism:
- > A variable can be assigned different types of objects at runtime provided they are a subtype of the variable's type.
- > Method implementation is determined by the type of object, not the type of the declaration (dynamic binding).
- > Only method signatures defined by the variable type can be called without casting.

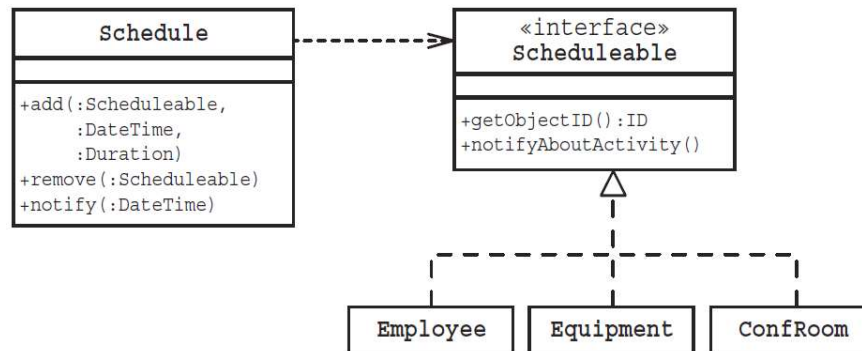
61

## Polymorphism: Example



62

## Polymorphism: Example



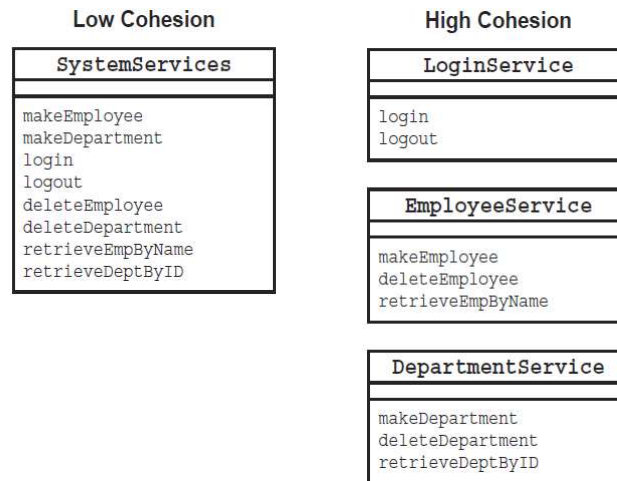
63

## Cohesion

- > In software, cohesion refers to how well a given component or method supports a single purpose.
  - Low cohesion occurs when a component is responsible for many unrelated features.
  - High cohesion occurs when a component is responsible for only one set of related features.
  - A component includes one or more classes. Therefore, cohesion applies to a class, a subsystem, and a system.
  - Cohesion also applies to other aspects including methods and packages.
  - Components that do everything are often described with the Anti-Pattern term of Blob components.

64

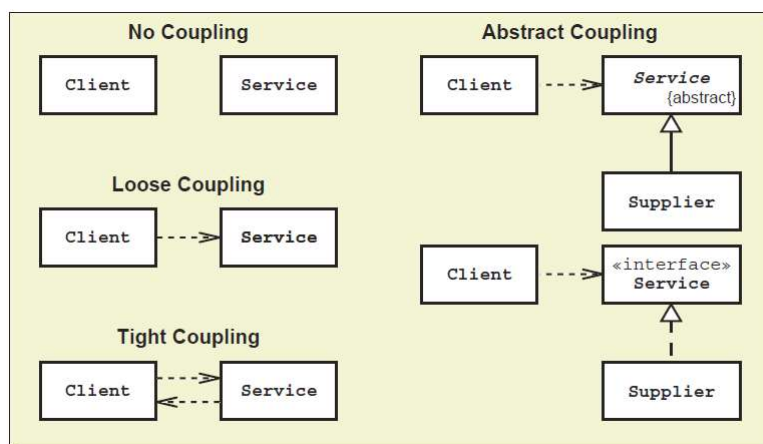
## Cohesion: Example



65

## Coupling

Coupling is “the degree to which classes within our system are dependent on each other.” (Knoernschild page 174)



66

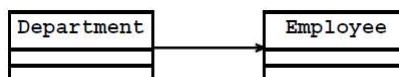
## Implementation Inheritance

- > Implementation inheritance is inheriting shared attributes and methods from a superclass
- > Advantages:
  - Avoids duplication of code that is common to subtypes
  - Organizes classes according to inheritance
- > Disadvantages:
  - Forces subclass to inherit everything from its superclass
  - Changes to the superclass might affect the subclass

67

## Composition

- > Builds complex objects from simpler objects
- > Forms a looser coupling than implementation inheritance



```
class department:
    def __init__(self, employee):
        self.__employee = employee

    @property
    def employee(self):
        return self.__employee
```

68

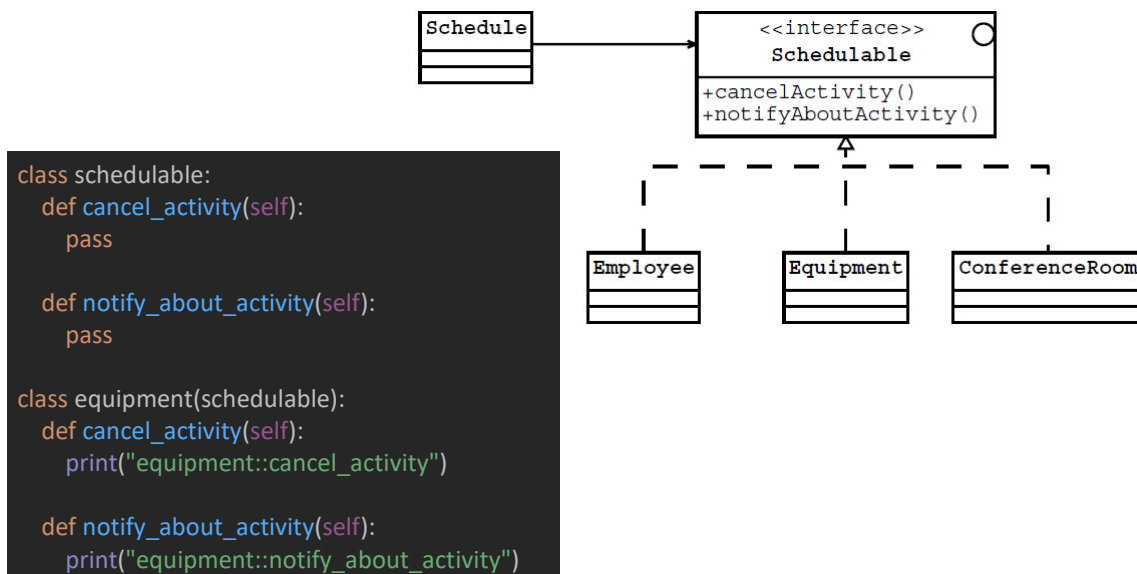
## Interface Inheritance

Interface inheritance is the separation of an interface definition from its implementation:

- > Similar to hardware devices that implement a common interface
- > Can make a class extensible without being modified
- > Java technology-based interfaces (Java interfaces) provide pure interface inheritance; abstract classes allow a mixture of implementation and interface inheritance
- > This course uses interfaces whenever possible and leaves it to the attendee to determine whether an abstract class or interface is more appropriate for them

69

## Interface Inheritance Example



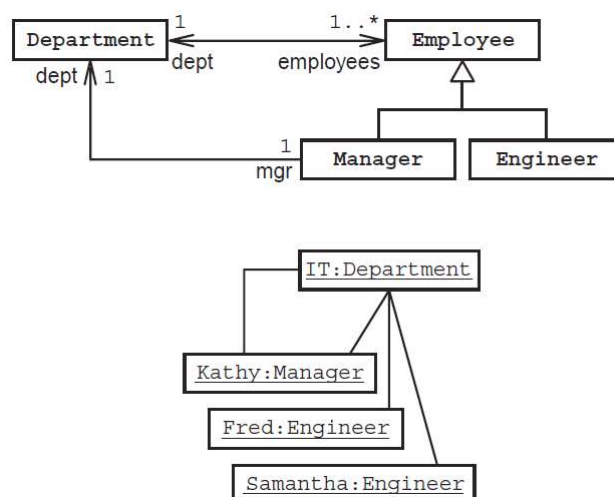
70

## Class Associations and Object Links

- > Dimensions of associations include:
  - The roles that each class plays
  - The multiplicity of each role
    - **1** denotes exactly one
    - **1..\*** denotes one or more
    - **0..\*** or **\*** denotes zero or more
  - The direction (or navigability) of the association
  - Object links:
    - Are instances of the class association
    - Are one-to-one relationships

71

## Class Associations and Object Links: Example



72

## Delegation

- > Many computing problems can be easily solved by delegation to a more cohesive component (one or more classes) or method.
- > Delegation is similar to how we humans behave.
  - A manager often delegates tasks to an employee with the appropriate skills.
  - You often delegate plumbing problems to a plumber.
  - A car delegates accelerate, brake, and steer messages to its subcomponents, who in turn delegate messages to their subcomponents. This delegation of messages eventually affects the engine, brakes, and wheel direction respectively.
- > OO paradigm frequently mimics the real world.

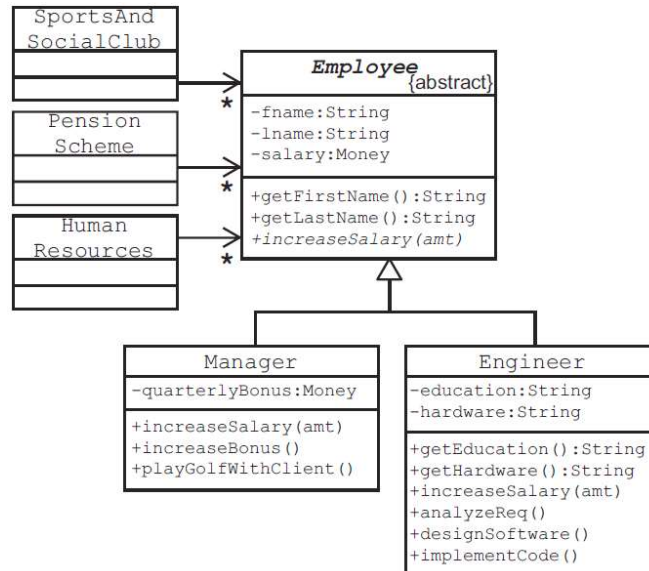
73

## Delegation

- > The ways you delegate in OO paradigm include delegating to:
  - A more cohesive linked object
  - A collection of cohesive linked objects
  - A method in a subclass
  - A method in a superclass
  - A method in the same class

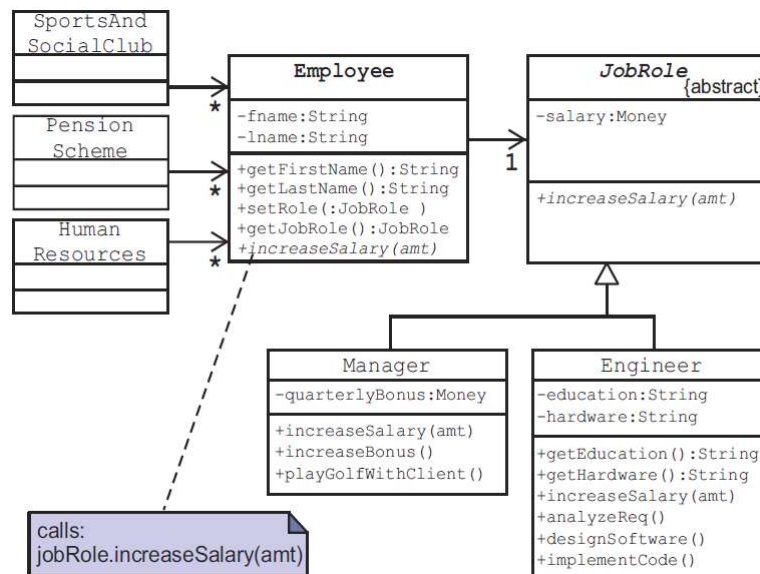
74

## Delegation: Example Problem



75

## Delegation: Example Solution



76

## Exploring Object-Oriented Design Principles

- > The Gang of Four book outlines three object-oriented design principles:
  - Favoring composition
  - Programming to an interface
  - Designing for change

77

## Favoring Composition

“Favor object composition over [implementation] inheritance.” (Gamma, Helm, Johnson, and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*)

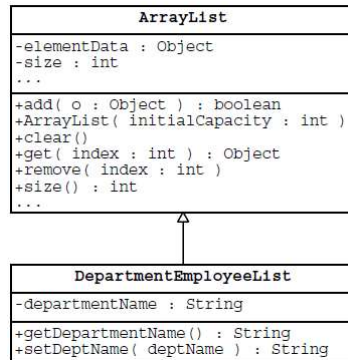
Reuse functionality through composition instead of implementation inheritance:

- > Implementation inheritance is white-box reuse
- > Composition is black-box reuse

78

## Favoring Composition

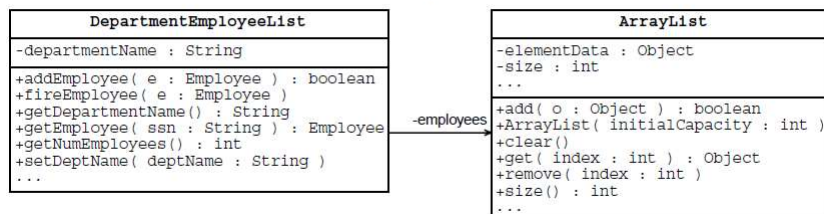
### Implementation Inheritance



79

## Favoring Composition

### Composition



80

## Programming to an Interface

“Program to an interface, not an implementation.”

(Gamma, Helm, Johnson, and Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*)



81



**SOLID**

OOP **PRINCIPLES**

82

## THE SINGLE RESPONSIBILITY PRINCIPLE (SRP)



83

### The Single-Responsibility Principle (SRP)

***A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE.***

- > This principle was described in the work of Tom DeMarco [1] and Meilir Page-Jones [2].
- > They called it **cohesion**, which they defined as the functional relatedness of the elements of a module.

**[1] Tom DeMarco**, *Structured Analysis and System Specification*, Yourdon Press Computing Series, 1979.

**[2] Meilir Page-Jones**, *The Practical Guide to Structured Systems Design*, 2d. ed., Yourdon Press Computing Series, 1988.

84

## The Single-Responsibility Principle (SRP)

- > **Why is it so important to separate responsibilities into separate classes?**
- > Each responsibility is an *axis of change*.
- > When the requirements change, that change will be manifest through a change in responsibility among the classes.
- > If a class assumes more than one responsibility, that class will have more than one reason to change.

85

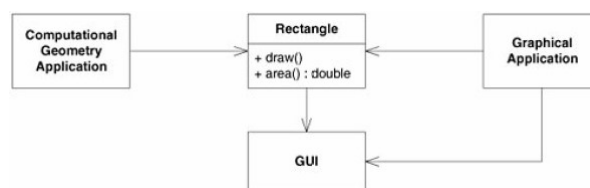
## The Single-Responsibility Principle (SRP)

- > If a class has more than one responsibility, the responsibilities become coupled.
- > Changes to one responsibility may impair or inhibit the class's ability to meet the others.
- > This kind of coupling leads to fragile designs that break in unexpected ways when changed.

86

## The Single-Responsibility Principle (SRP)

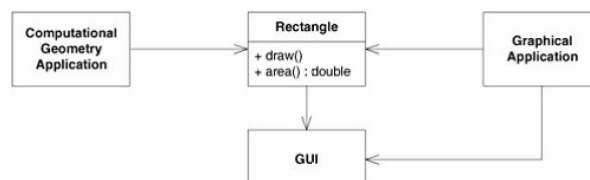
- > Two different applications use the **Rectangle** class.
- > One application does **computational geometry**. Using Rectangle to help it with the mathematics of geometric shapes but never drawing the rectangle on the screen.
- > The other application is graphical in nature and may also do some computational geometry, but it definitely **draws the rectangle** on the screen.



87

## The Single-Responsibility Principle (SRP)

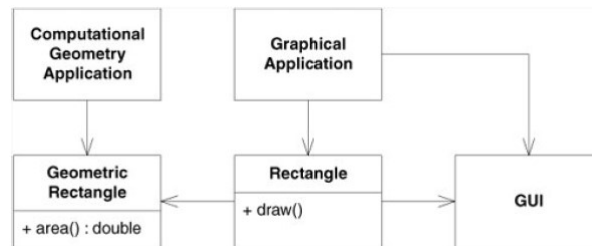
- > This design violates SRP.
- > The Rectangle class has two responsibilities.
  1. The first responsibility is to provide a mathematical model of the geometry of a rectangle.
  2. The second responsibility is to render the rectangle on a GUI.



88

## The Single-Responsibility Principle (SRP)

- > A better design is to separate the two responsibilities into two completely different classes



89

## Defining a Responsibility

- > In the context of the SRP, we define a responsibility to be *a reason for change*.
- > If you can think of more than one motive for changing a class, that class has more than one responsibility.
- > This is sometimes difficult to see.
- > We are accustomed to thinking of responsibility in groups.

90

## Defining a Responsibility: Example

```
class Modem:
    def dial(self, phone_number: str) -> None:
        pass

    def hangup(self) -> None:
        pass

    def send(self, data: bytes) -> None:
        pass

    def receive(self) -> bytes:
        pass
```

91

## Defining a Responsibility: Example

```
class Modem:
    def dial(self, phone_number: str) -> None:
        pass

    def hangup(self) -> None:
        pass

    def send(self, data: bytes) -> None:
        pass

    def receive(self) -> bytes:
        pass
```

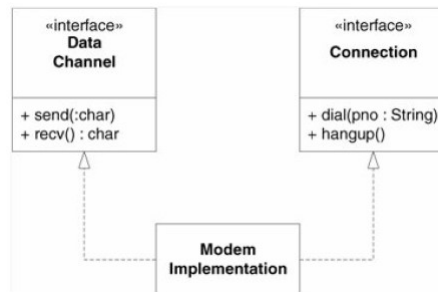
Two responsibilities are being shown here:

- > The first responsibility is **connection management**.
- > The second is **data communication**.
- > **dial** and **hangup** functions manage the connection; **send** and **recv** functions communicate data.

92

## Defining a Responsibility: Example

> Should these two responsibilities be separated?



93

## Defining a Responsibility: Example

> Should these two responsibilities be separated?

- That depends on how the application is changing.
- If the application changes in ways that affect the signature of the connection functions, the design will smell of rigidity, because the classes that call send and read will have to be recompiled and redeployed more often than we like. In that case, the two responsibilities should be separated.
- This keeps the client applications from coupling the two responsibilities.

94

## Persistence

- > The following shows a common violation of SRP.



- > The Employee class contains business rules and persistence control. These two responsibilities should almost never be mixed.
- > Business rules tend to change frequently, and although persistence may not change as frequently, it changes for completely different reasons.
- > Binding business rules to the persistence subsystem is asking for trouble.

95

## Conclusion

- > The Single-Responsibility Principle is one of the simplest of the principles but one of the most difficult to get right.
- > Conjoining responsibilities is something that we do naturally.
- > Finding and separating those responsibilities is much of what software design is really about.
- > Indeed, the rest of the principles we discuss come back to this issue in one way or another.

96

## THE OPEN/CLOSED PRINCIPLE (OCP)



97

### The Open/Closed Principle

- > As Ivar Jacobson has said, "All systems change during their life cycles. This must be borne in mind when developing systems expected to last longer than the first version." [1]
- > How can we create designs that are stable in the face of change and that will last longer than the first version?
- > Bertrand Meyer [2] gave us guidance as long ago as 1988 when he coined the now-famous open/closed principle.

98

## The Open/Closed Principle (OCP)

- > *Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.*
- > When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity.
- > OCP advises us to refactor the system so that further changes of that kind will not cause more modifications.
- > If OCP is applied well, further changes of that kind are achieved by adding new code, not by changing old code that already works.

99

## Description of OCP

- > They are *open for extension*. This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.
- > They are *closed for modification*. Extending the behavior of a module does not result in changes to the source, or binary, code of the module. The binary executable version of the module remains untouched.

100

## Description of OCP

- > It would seem that these two attributes are at odds.
- > The normal way to extend the behavior of a module is to make changes to the source code of that module.
- > A module that cannot be changed is normally thought to have a fixed behavior.
- > How is it possible that the behaviors of a module can be modified without changing its source code?
- > Without changing the module, how can we change what a module does?

101

## Description of OCP

- > The answer is abstraction.
- > In any object-oriented programming language (OOPL), it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors.
- > The abstractions are abstract base classes, and the unbounded group of possible behaviors are represented by all the possible derivative classes.

102

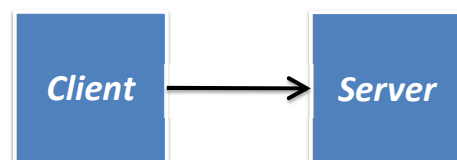
## Description of OCP

- > It is possible for a module to manipulate an abstraction.
  - Such a module can be closed for modification, since it depends on an abstraction that is fixed.
  - Yet the behavior of that module can be extended by creating new derivatives of the abstraction.

103

## Description of OCP

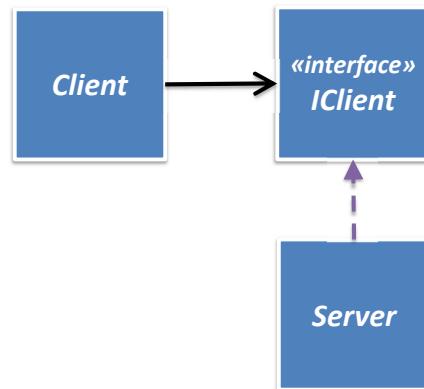
- > Both the *Client* and *Server* classes are concrete.
- > The *Client* class uses the *Server* class.
- > If we want for a *Client* object to use a different server object, *Client* class must be changed to name new server class.



104

## Description of OCP

- > If we want Client objects to use a different server class, a new derivative of the *ClientInterface* class can be created.
- > The Client class can remain unchanged.



105

## Description of OCP

- > You may wonder why I named **IClient** the way I did.
- > Why didn't I call it **AbstractServer** instead?
- > Abstract classes are more closely associated to their clients than to the classes that implement them.

106



## THE LISKOV SUBSTITUTION PRINCIPLE (LSP)

107

### The Liskov Substitution Principle

- > The primary mechanisms behind the Open/Closed Principle are abstraction and polymorphism.
- > In statically typed languages, one of the key mechanisms that supports abstraction and polymorphism is inheritance.
- > By using inheritance, we can create derived classes that implement abstract methods in base classes.
- > What are the design rules that govern this particular use of inheritance?
- > What are the characteristics of the best inheritance hierarchies?
- > What are the traps that will cause us to create hierarchies that do not conform to OCP?

108

## The Liskov Substitution Principle

- > New classes should be logical, consistent extensions of their superclasses, but what does it mean to be *logical* and *consistent*?
- > A Java compiler will ensure a certain level of consistency, but many principles of consistency will elude a compiler.
- > One principle you should consider in your designs is the *Liskov Substitution Principle* (LSP).
- > This principle, documented in Liskov (1987), can be paraphrased: An instance of a class should function as an instance of its superclass.

109

## The Liskov Substitution Principle

In short:

- > if **X** IS-A **Y**, then anywhere you could use a **Y**, you can substitute an **X** and things should just work.
- > So, if every Manager IS-AN Employee, every piece of code that operates on Employees can operate on Managers and just work.

110

## LSP Example

> Consider the following Rectangle class:

```
class Rectangle:
    def __init__(self, width: float, height: float):
        self.__width = width
        self.__height = height

    @property
    def width(self) -> float:
        return self.__width

    @width.setter
    def width(self, value) -> None:
        self.__width = value

    @property
    def height(self) -> float:
        return self.__height

    @height.setter
    def height(self, value) -> None:
        self.__height = value

    def area(self) -> float:
        return self.width * self.height
```

111

## LSP Example

> Here's the Square class:

```
class Square(Rectangle):
    def __init__(self, edge: float):
        super().__init__(edge, edge)

    @property
    def width(self) -> float:
        return self.__width

    @width.setter
    def width(self, value) -> None:
        self.__width = value
        self.__height = value

    @property
    def height(self) -> float:
        return self.__height

    @height.setter
    def height(self, value) -> None:
        self.__width = value
        self.__height = value
```

112

## LSP Example

- > Now, had about a Square class? Clearly, a square is a rectangle, so the Square class should be derived from the Rectangle class, right? Let's see!
- > Observations:
  - A square does not need both a width and a height as attributes, but it will inherit them from Rectangle anyway. So, each Square object wastes a little memory, but this is not a major concern.
  - The inherited `setWidth()` and `setHeight()` methods are not really appropriate for a Square, since the width and height of a square are identical. So we'll need to override `setWidth()` and `setHeight()`.

113

## LSP Example

- > Everything looks good. But check this out!

```
public class TestRectangle {  
    public static void testLSP(Rectangle r){  
        r.setWidth(4.0);  
        r.setHeight(5.0);  
        if (r.area() == 20.0)  
            System.out.println("Looking good!\n");  
        else  
            System.out.println("What kind of " +  
                               "rectangle is this??\n");  
    }  
}
```

114

## LSP Example

```
def test_lsp(rectangle: Rectangle) -> None:
    rectangle.width = 4.0
    rectangle.height = 5.0
    if rectangle.area() == 20.0:
        print("Looking good!\n");
    else:
        print("What kind of rectangle is this?\n")

r = Rectangle(4, 5)
test_lsp(r)
r = Square(10)
test_lsp(r)
```

115

## LSP Example

- > A mathematical square might be a rectangle, but a Square object is not a Rectangle object
- > A Square object is not consistent with the behavior of a Rectangle object!
- > Behaviorally, a Square is *not* a Rectangle!
- > A Square object is not polymorphic with a Rectangle object.

116

## LSP Example

```
class Shape:
    def area(self) -> float:
        pass
```

117

## LSP Example

```
> class Rectangle(Shape):
    def __init__(self, width: float, height: float):
        self.__width = width
        self.__height = height

    @property
    def width(self) -> float:
        return self.__width

    @width.setter
    def width(self, value) -> None:
        self.__width = value

    @property
    def height(self) -> float:
        return self.__height

    @height.setter
    def height(self, value) -> None:
        self.__height = value

    def area(self) -> float:
        return self.width * self.height
```

118

## LSP Example

```
class Square(Shape):
    def __init__(self, edge: float):
        self.__edge = edge

    @property
    def edge(self) -> float:
        return self.__edge

    @edge.setter
    def edge(self, value) -> None:
        self.__edge = value

    def area(self) -> float:
        return self.edge ** 2

shapes: list[Shape] = [Rectangle(5, 10), Square(30), Rectangle(20, 30), Square(50)]
for shape in shapes:
    print(shape.area())
```

119

## Liskov Substitution Principle

- > The Liskov Substitution Principle (LSP) makes it clear that **IS-A** relationship is all about behavior
- > In order for the LSP to hold (and with it the Open-Closed Principle) all subclasses must conform to the behavior that clients expect of the base classes they use
- > A subtype must have no more constraints than its base type, since the subtype must be usable anywhere the base type is usable

120

## Liskov Substitution Principle

- > If the subtype has more constraints than the base type, there would be uses that would be valid for the base type, but that would violate one of the extra constraints of the subtype and thus violate the LSP!
- > The guarantee of the LSP is that a subclass can always be used wherever its base class is used!

121

## PRINCIPLE #4

### THE INTERFACE SEGREGATION PRINCIPLE (ISP)

*Clients should not be forced to depend on methods they do not use.*

122

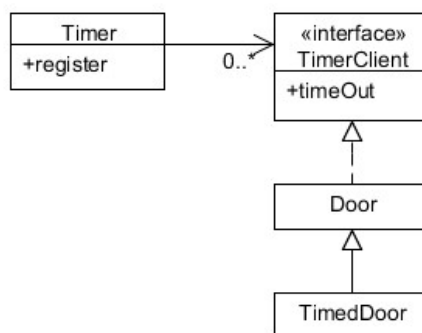
## Fat Interfaces

- > "Fat" interfaces
  - Classes whose interfaces are not cohesive have "fat" interfaces.
  - The interfaces of the class can be broken up into groups of methods.
  - Each group serves a different set of clients. Thus, some clients use one group of methods, and other clients use the other groups.
- > It suggests
  - Clients should not know about them as a single class.
  - Clients should know about abstract base classes that have cohesive interfaces.

123

## Interface Pollution

- > *Clients should not be forced to depend on methods they do not use.*



124

```

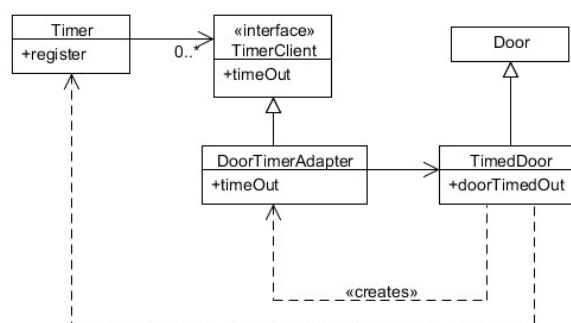
class TimerClient:
    def timeOut(self, timeout_id: int) -> None:
        pass

class Timer:
    def register(self, timeout: int, timeout_id: int, client: TimerClient):
        # provide an implementation
        pass

```

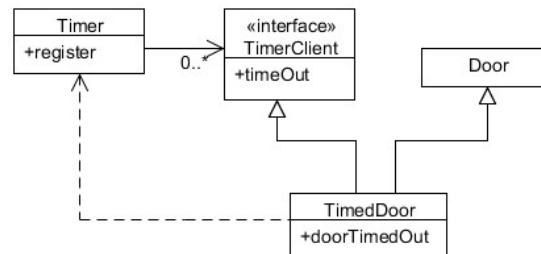
125

## Separation Through Delegation



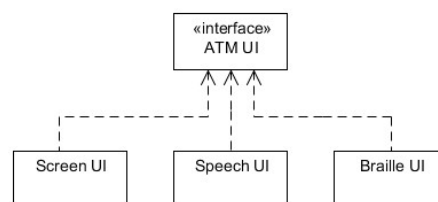
126

## Separation Through Multiple Inheritance



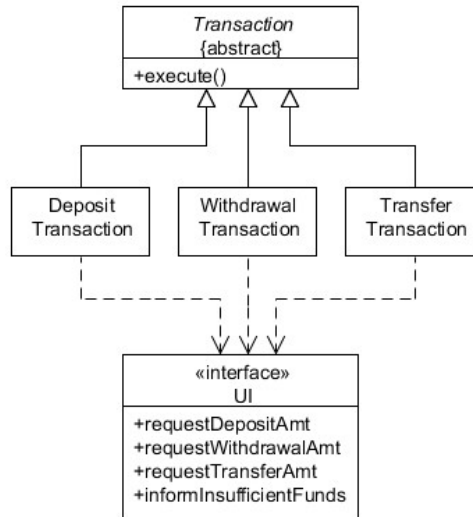
127

## The ATM User Interface Example



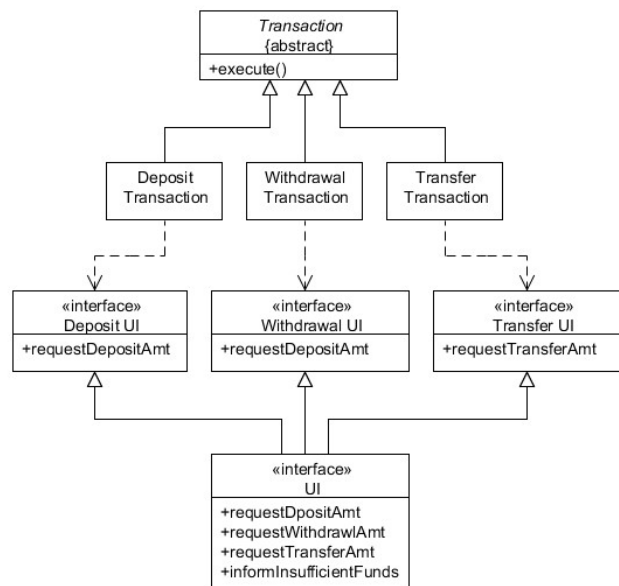
128

## ATM transaction hierarchy



129

## Segregated ATM UI interface



130

## DEPENDENCY INVERSION PRINCIPLE (DIP)

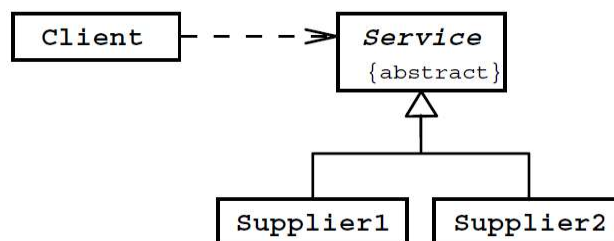
131

### Dependency Inversion Principle

“Depend on abstractions. Do not depend on concretions.”

(Knoernschild page 12)

> An abstraction can be an abstract class:



132

## Dependency Inversion Principle

- > An abstraction can be a Java technology interface:

