C++ Programming

Instructor: Rita Kuo
Office: CS 520E
Phone: Ext. 4405
E-mail: rita.kuo@uvu.edu

Mapping zyBooks Chapters

Topic	zyBooks Chapter
UML	12.7, 12.8
Inheritance	12.1, 12.2
Overriding	12.3
Polymorphism	12.4, 12.5, 12.6
Wrap Up	12.7

UML

https://www.tutorialspoint.com/uml/index.htm

Unified Modeling Language (UML)

- Usage
  - A standard language for specifying, visualizing, constructing and documenting the artifacts of software systems
- Conceptual Model
  - Can be defined as a model which is made of concepts and their relationships
  - Is the first step before drawing a UML diagram
  - Helps to understand the entities in the real world and how they interact with each other
- Major Elements
  - UML building blocks
  - Rules to connect the building blocks
  - Common mechanisms of UML

Unified Modeling Language (UML)

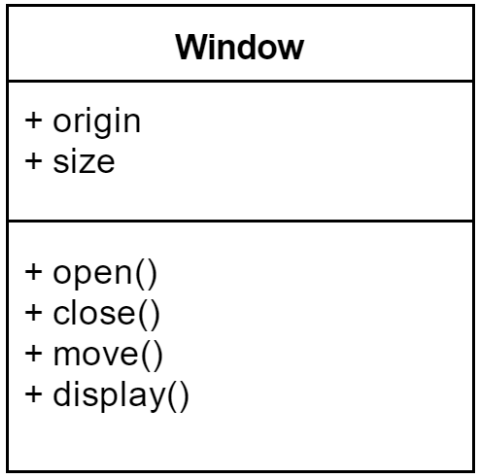
- UML Building Blocks
  - Things
    - Are the most important building blocks of UML
    - Can be structural, behavioral, grouping, or annotational
  - Relationships
    - Shows how elements are associated with each other
    - Describe the functionality of an application
  - Diagrams
    - Are the ultimate output of the entire discussion.
    - All the elements, relationships are used to make a complete UML diagram and the diagram represents a system

UML - Things

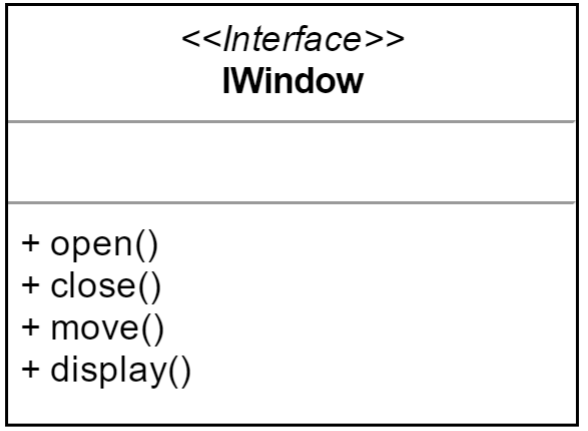
- Types
  - Structural
    - Define the static part of the model
    - Present physical and conceptual elements
  - Behavioral
    - Consist of the dynamic parts of UML model
  - Grouping
    - Can be defined as a mechanism to group elements of a UML model together
  - Annotational
    - Can be defined as a mechanism to capture remarks, descriptions, and comments of UML model elements.

UML - Structure Things

- Class
  - A set of objects having similar responsibilities



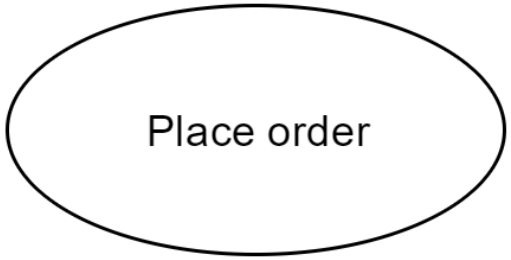
- Interface
  - A set of operations which specify the responsibility of a class



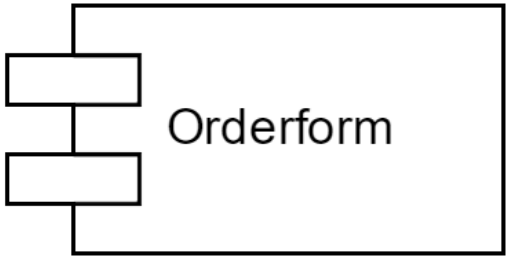
UML - Structure Things

- Collaboration
  - Defines interaction between elements

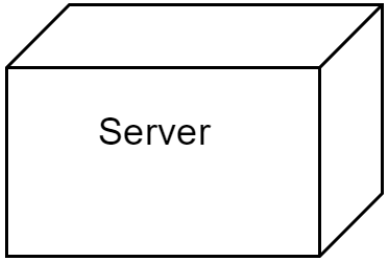
- Use Case
  - Represents a set of actions performed by a system for a specific goal



- Component
  - Describe physical part of a system

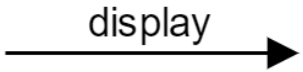


- Node
  - Can be defined as a physical element that exists at run time

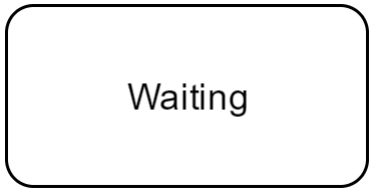


UML - Behavior Things

- Interaction
  - Is defined as a behavior that consists of a group of messages exchanged among elements to accomplish a specific task

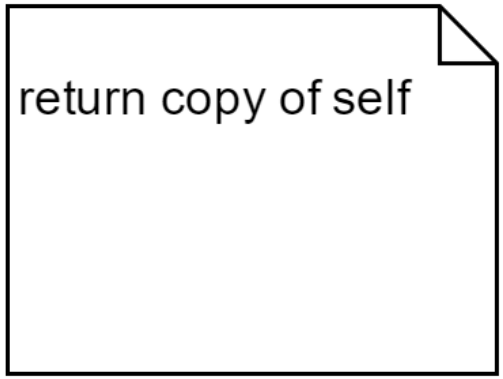


- State Machine
  - Is useful when the state of an object in its life cycle is important
  - Defines the sequence of state an object goes through in response to events



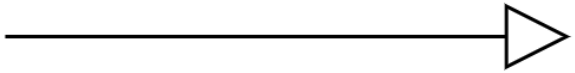
UML - Grouping and Annotational Things

- Grouping Things
  - Package: Is the only grouping thing available for gathering structural and behavior things
- Annotational Things
  - Note: Is used to render comments, constraints etc of an UML element

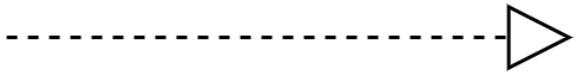


UML - Relationship

- Generalization
  - A relationship which connects a specialized element with a generalized element.
  - Basically describes inheritance relationship.

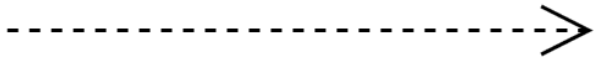


- Realization
  - A relationship in which two elements are connected.
  - One element describes some responsibility which is not implemented and the other one implements them
  - Exists in case of interfaces

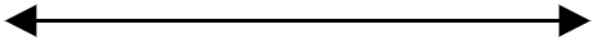


UML - Relationship

- Dependency
  - A relationship between two things in which change in one element also affects the other one



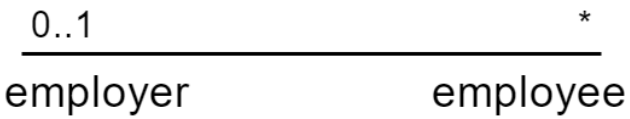
- Association
  - A set of links that connects elements of an UML model
  - Describes how many objects are taking part in that relationship



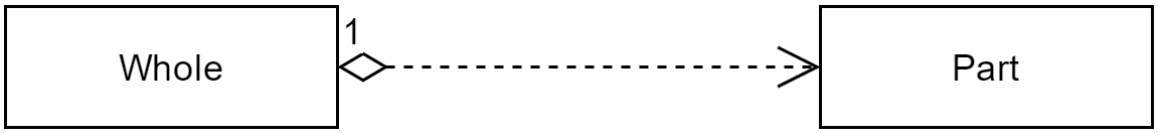
  - Role: when a class participates in an association, it has a specific role that it plays in that relationship

UML - Relationship

- Association
  - Multiplicity: state how many objects may be connected across an instance of an association



  - Aggregation: model a “whole/part” relationship, in which one class represents a larger thing (the “whole”), which consists of smaller things (the “parts”)

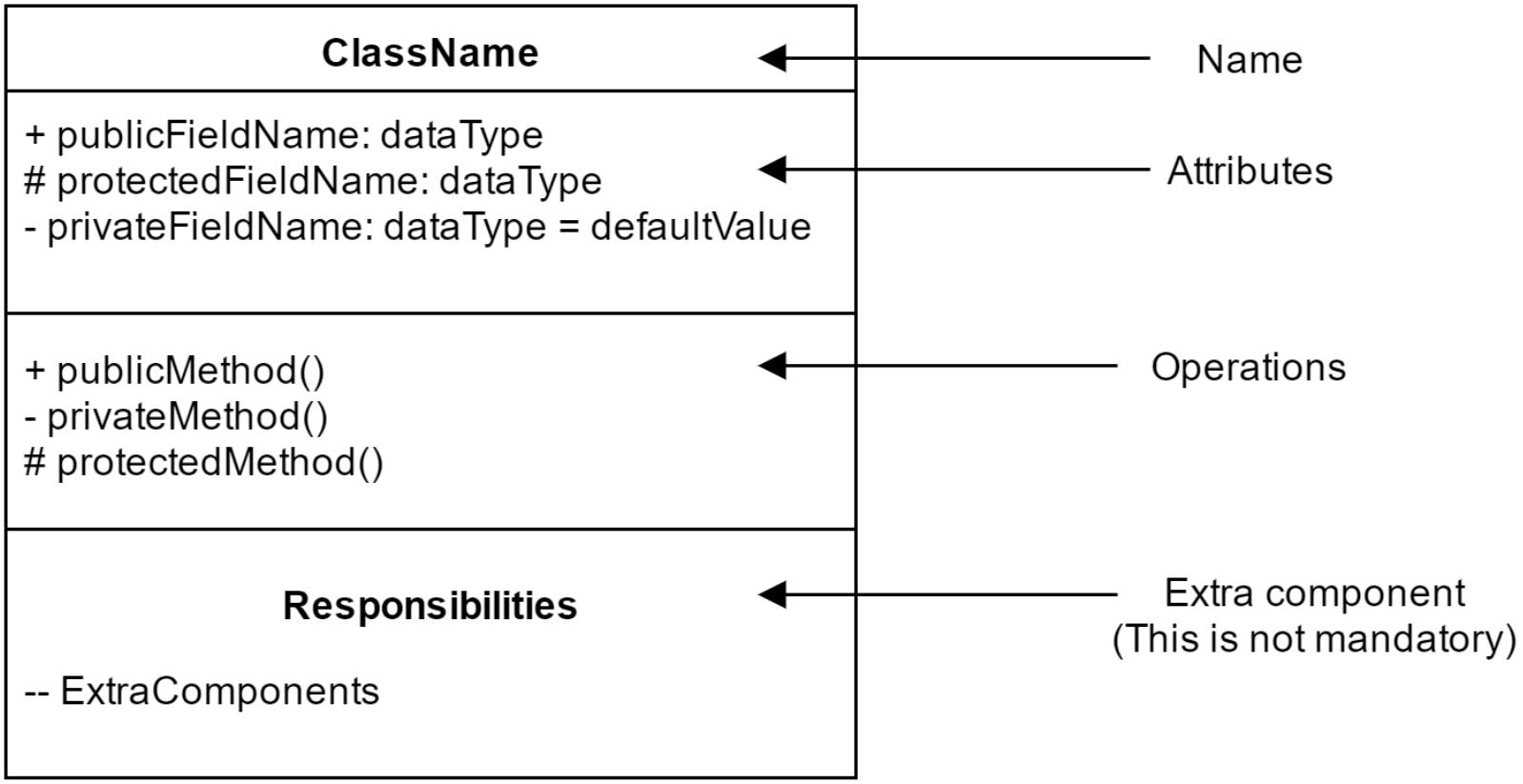


UML Diagrams

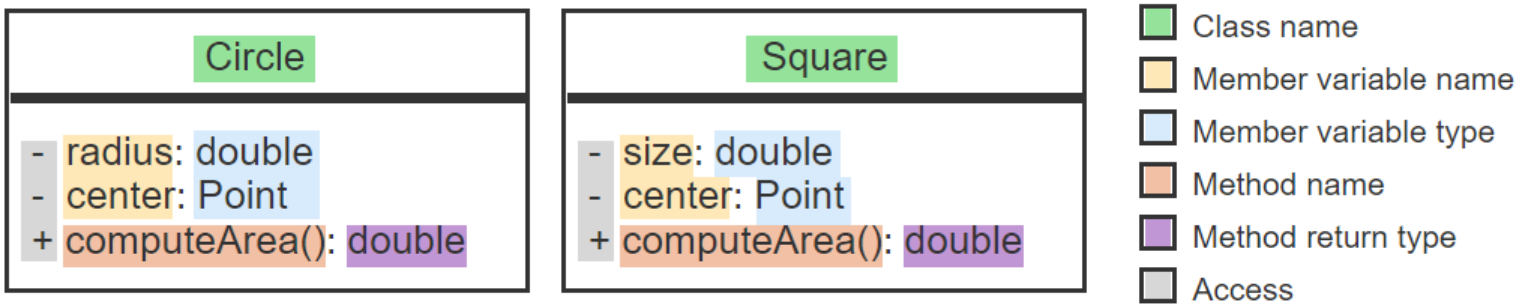
- Types
  - Class Diagram
  - Object Diagram
  - Use Case Diagram
  - Sequence Diagram
  - Collaboration Diagram
  - Activity Diagram
  - Statechart Diagram
  - Deployment Diagram
  - Component Diagram
- Tools for Drawing UML
  - http://www.draw.io
  - Dia (http://dia-installer.de/)
  - Microsoft Visio
  - Software Ideas Modeler (https://www.softwareideas.net/)

Class Diagram

- Definition in UML



Class Diagram

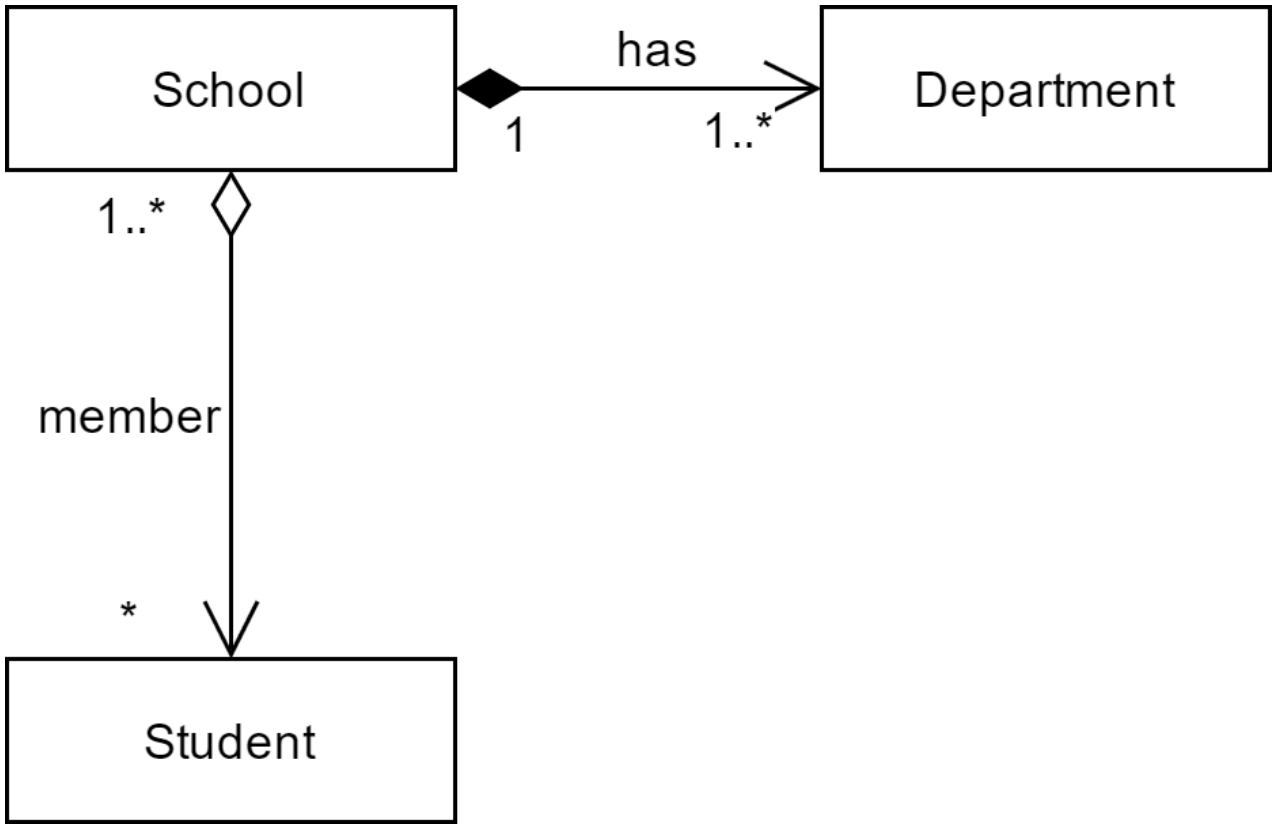



```
class Circle {
  public:
    double computeArea();
  private:
    double radius;
    Point center;
};
```

```
class Square {
  public:
    double computeArea();
  private:
    double size;
    Point center;
};
```

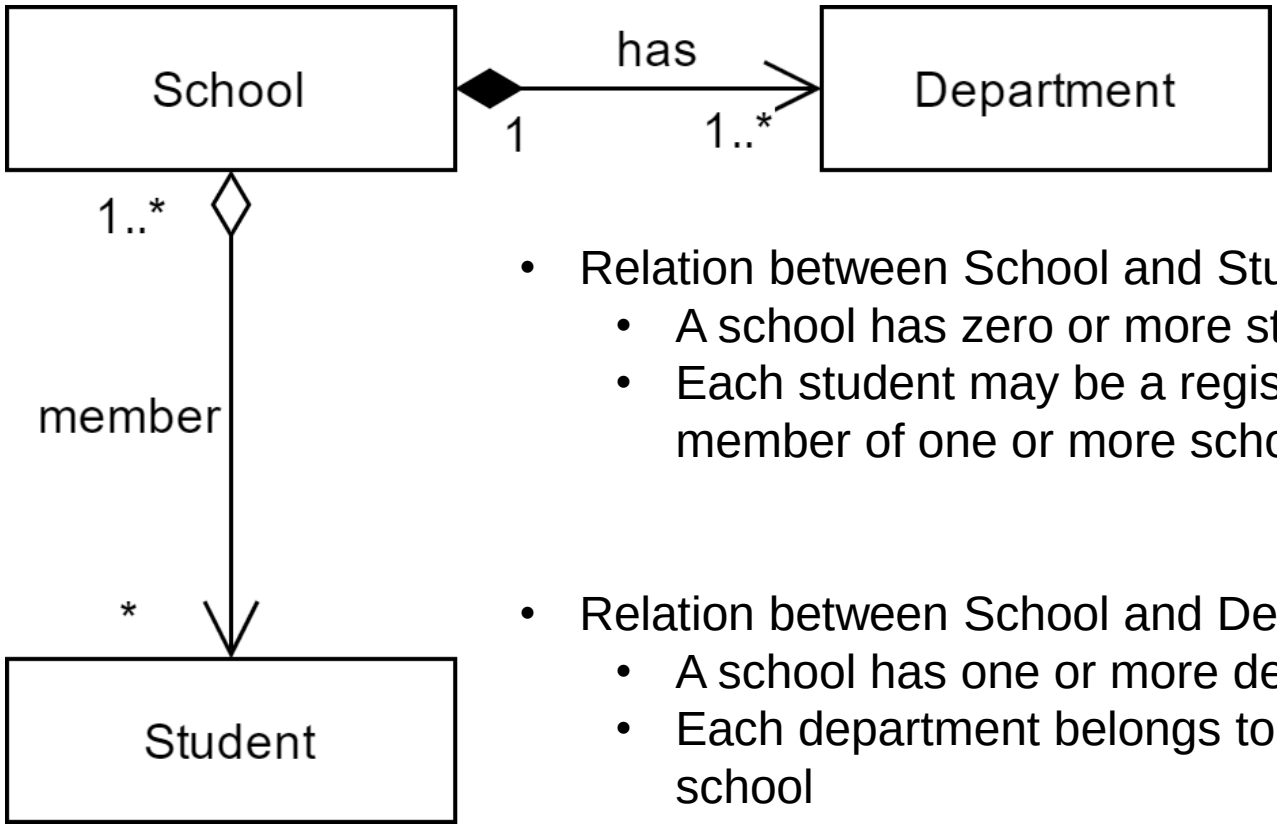
Aggregation Relation

- Model a "whole/part" relationship
- Example



Aggregation Relation

- Model a "whole/part" relationship
- Example

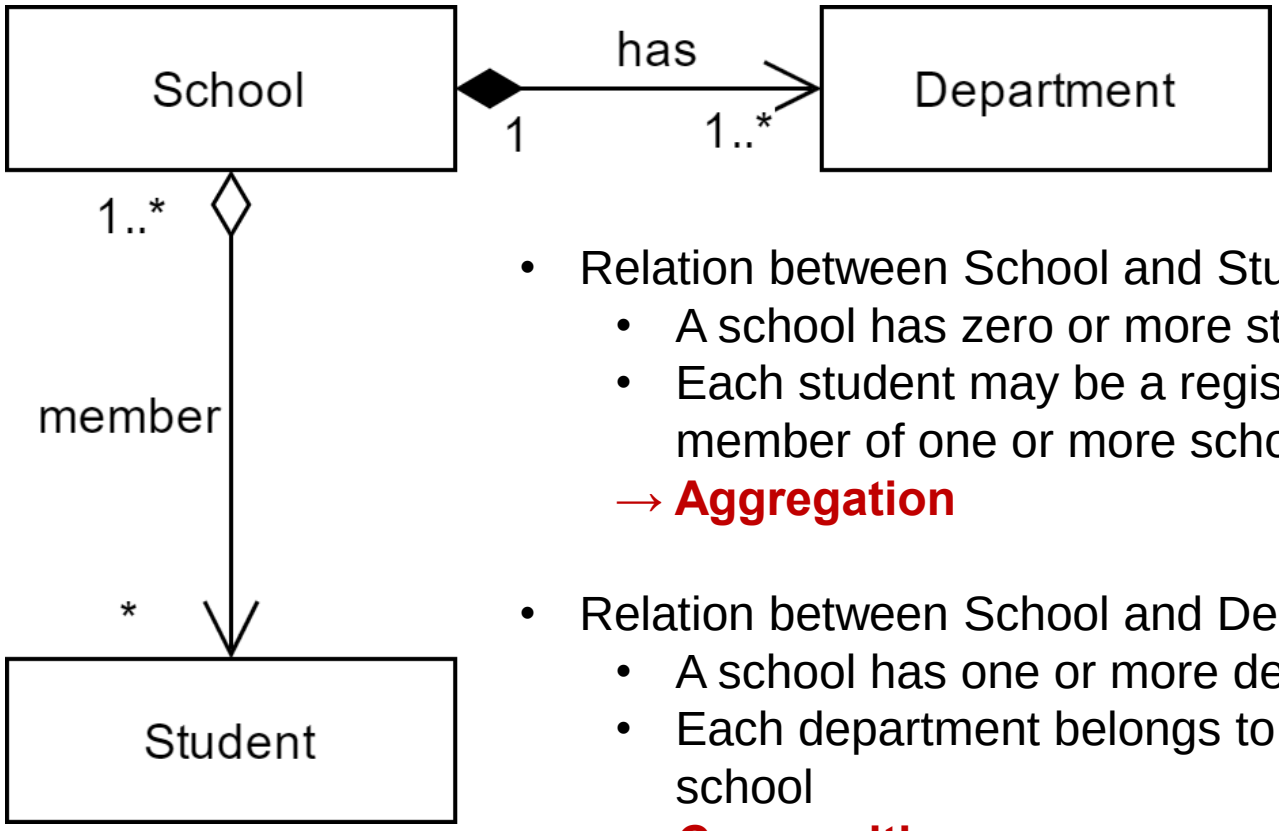


- Relation between School and Student
  - A school has zero or more students
  - Each student may be a registered member of one or more schools

- Relation between School and Department
  - A school has one or more departments
  - Each department belongs to exactly one school

Aggregation Relation

- Model a "whole/part" relationship
- Example



- Relation between School and Student
  - A school has zero or more students
  - Each student may be a registered member of one or more schools

  → **Aggregation**

- Relation between School and Department
  - A school has one or more departments
  - Each department belongs to exactly one school

  → **Composition**

Composition Example

- The 'has-a' relationship
  - A `MotherInfo` object 'has a' string object and 'has a' vector of `ChildInfo` objects

```
class ChildInfo {
   string firstName;
   string birthDate;
   string schoolName;

   ...
};

class MotherInfo {
   string firstname;
   string birthDate;
   string spouseName;
   vector<ChildInfo> childrenData;

   ...
};
```

Inheritance

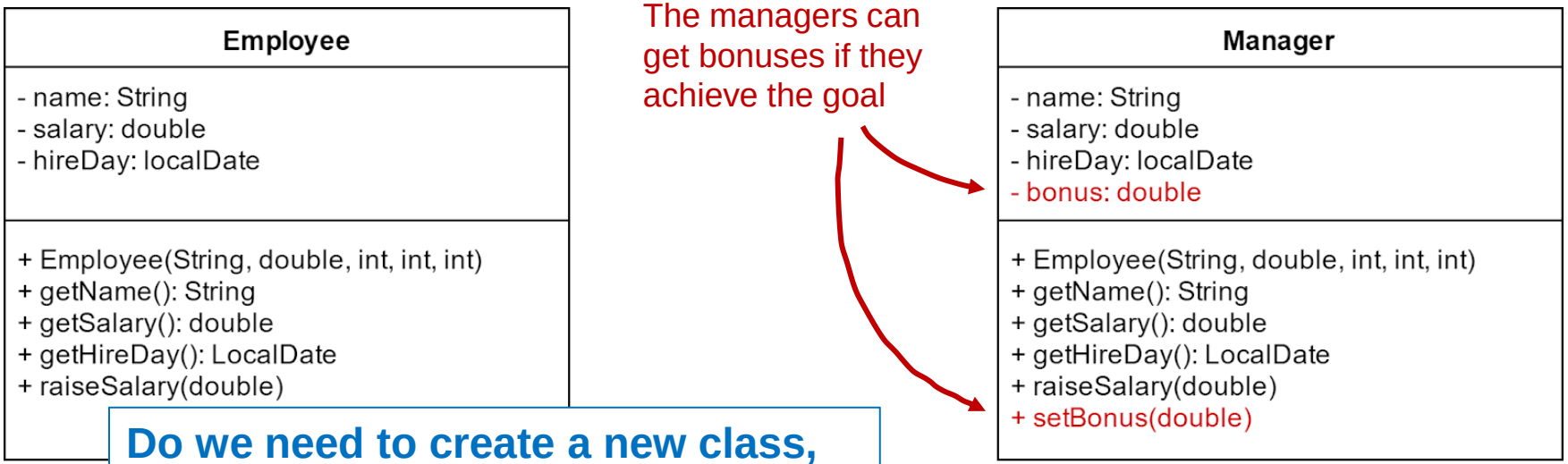
Review - Object-Oriented Programming Major Concepts

- Encapsulation
  - Restrict access to methods and attributes in a class.
  - Hide the complex details from the users, and prevent data being modified by accident
- Inheritance
  - Define a class that inherits all the methods and attributes from another class
  - Makes the OOP code more modular, easier to reuse and build a relationship between classes
- Polymorphism
  - Use a single interface with different underlying forms such as data types or classes

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter07.03-Inheritance-Encapsulation-and-Polymorphism.html

Inheritance

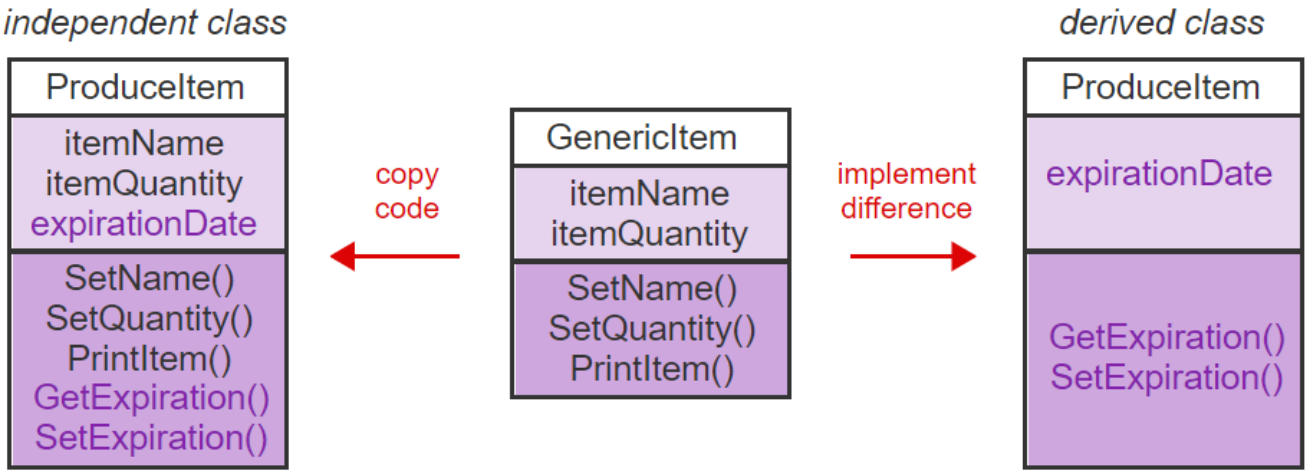
- Definition
  - Create new classes that are built on existing classes
- When you inherit from an existing class, you can
  - Reuse (or inherit) its methods
  - Add new methods and fields
- Example



Do we need to create a new class, Manager, and implement all fields and methods in Employee class?

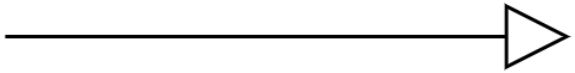
Inheritance

- Example

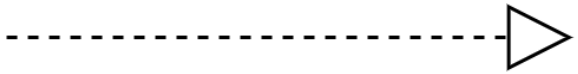


Review - UML - Relationship

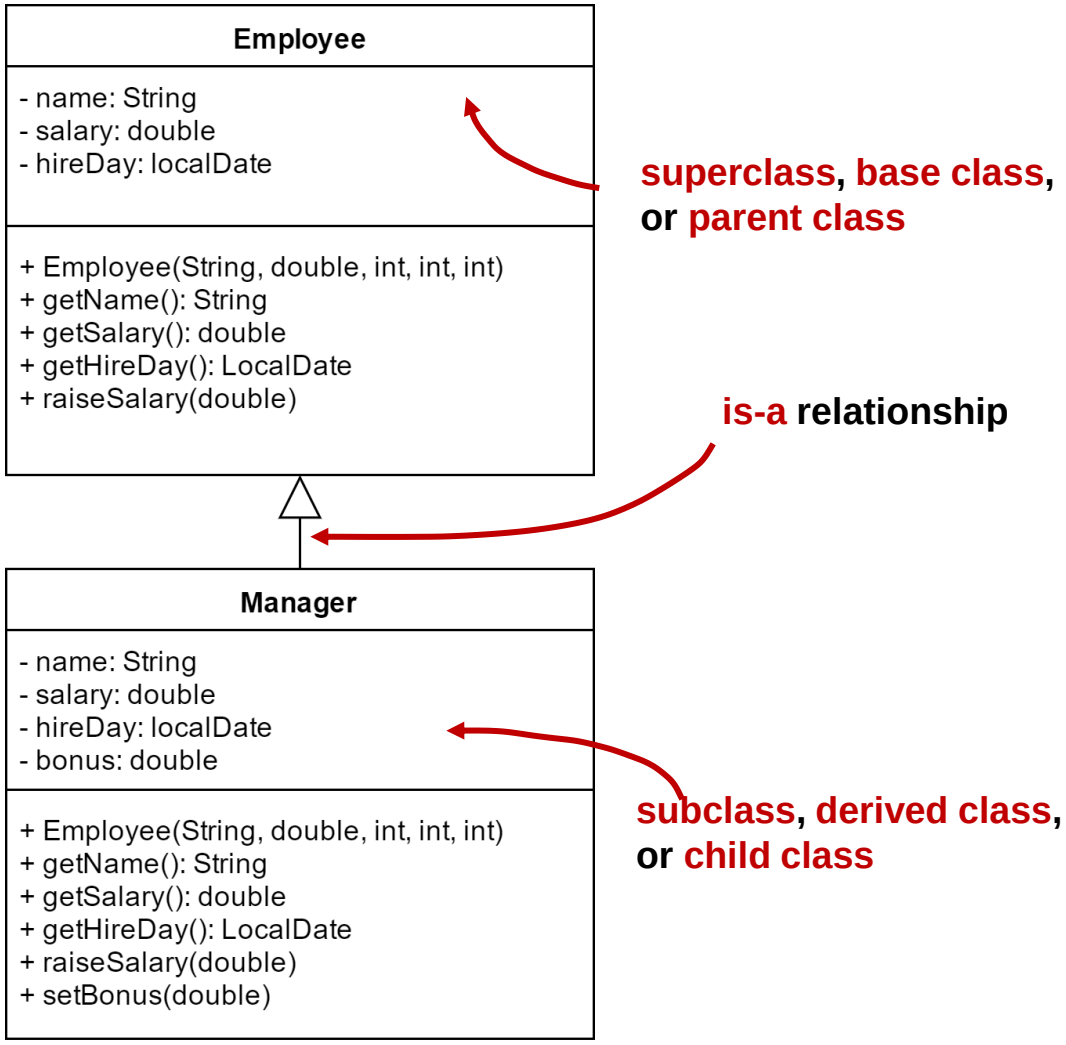
- **Generalization**
  - A relationship which connects a specialized element with a generalized element.
  - Basically describes inheritance relationship.



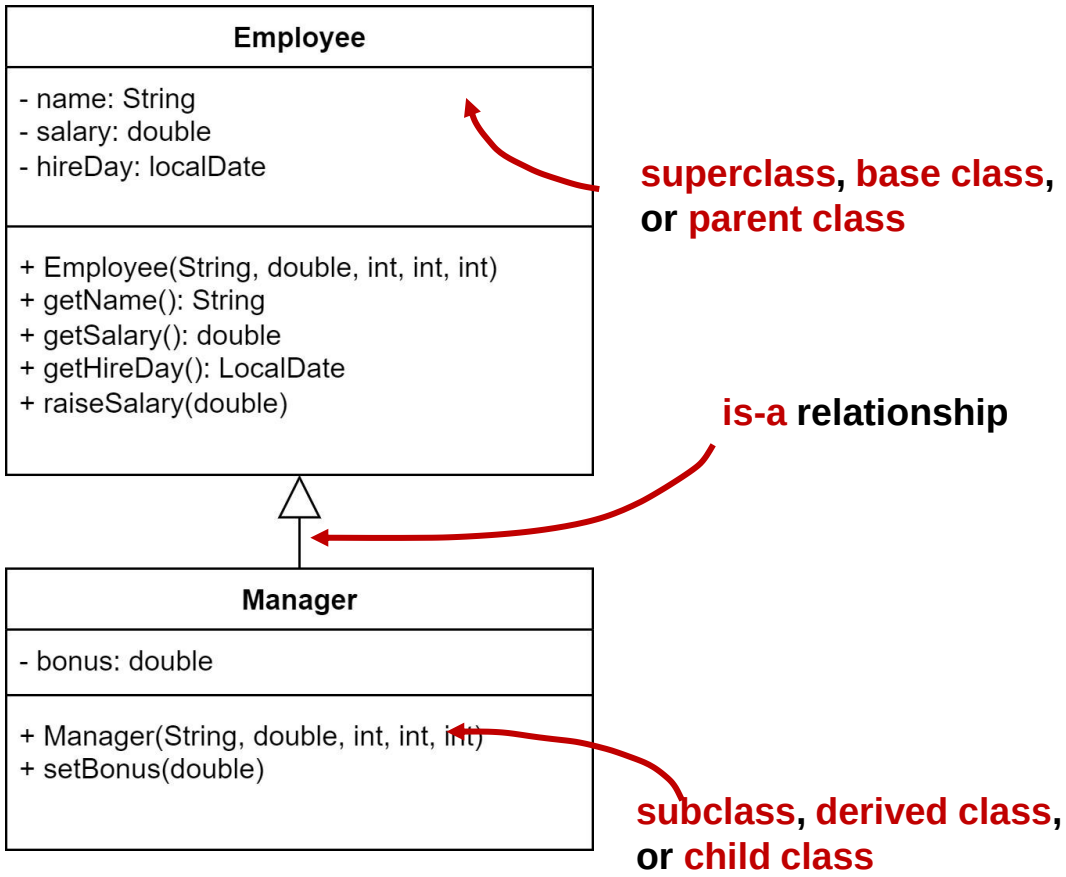
- Realization
  - A relationship in which two elements are connected.
  - One element describes some responsibility which is not implemented and the other one implements them
  - Exists in case of interfaces



Inheritance – UML Example



Inheritance – UML Example



Inheritance in C++

- Define the base class

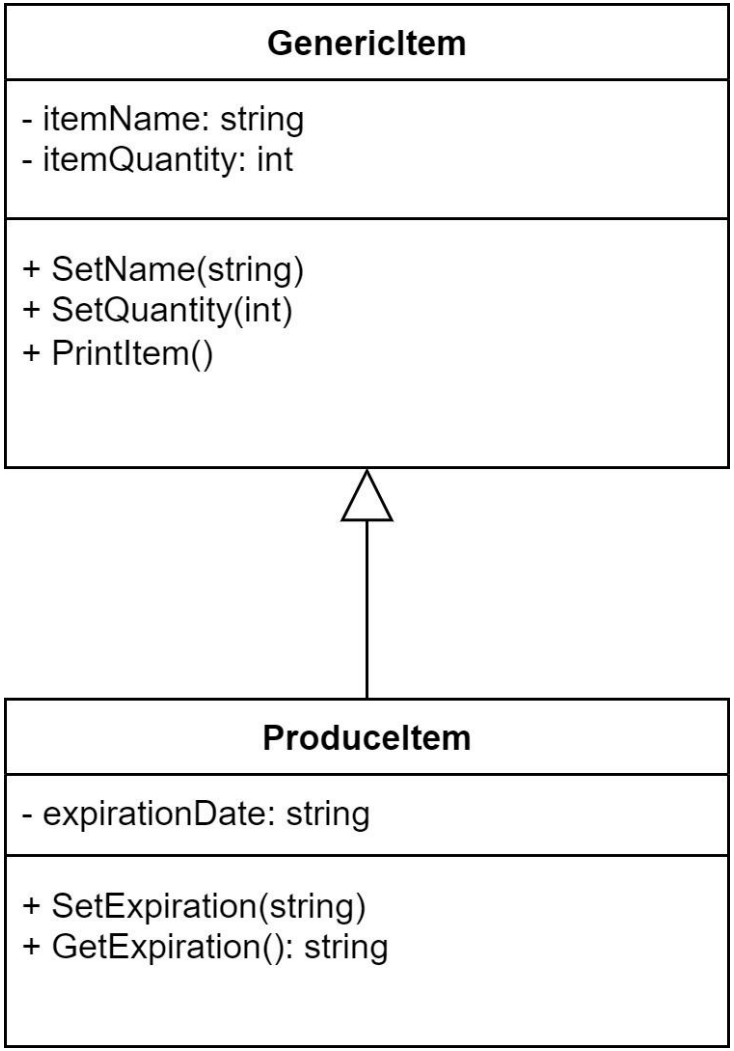
```
class GenericItem {
   public:
      void SetName(string newName) {
          itemName = newName;
      }

      void SetQuantity(int newQty) {
          itemQuantity = newQty;
      }

      void PrintItem() {
          cout << itemName << " "
               << itemQuantity << endl;
      }

   private:
      string itemName;
      int itemQuantity;
};
```

- Define the derived class

```
class ProduceItem : public GenericItem {
   public:
      void SetExpiration(string newDate) {
          expirationDate = newDate;
      }

      string GetExpiration() {
          return expirationDate;
      }

   private:
      string expirationDate;
};
```

Inheritance in C++

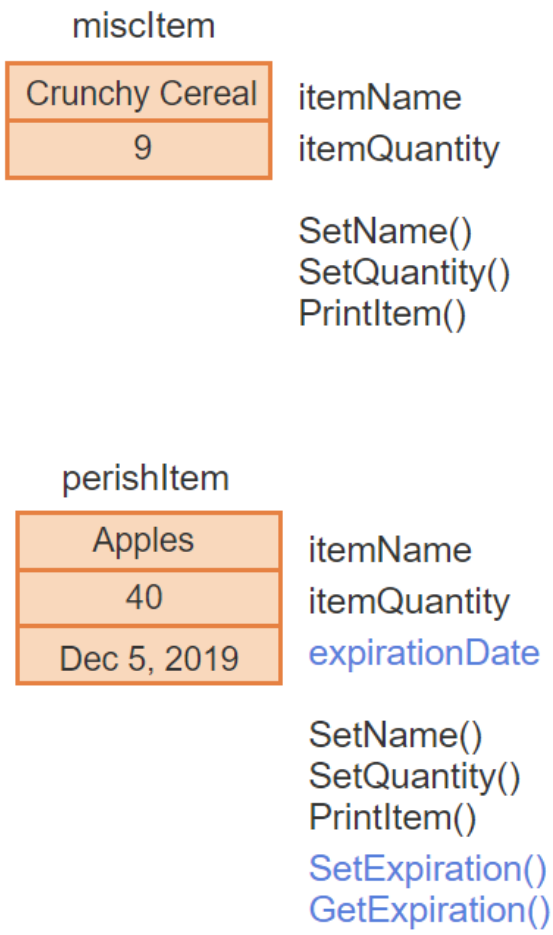
- UML diagram



Inheritance in C++

- Using `GenericItem` and `ProduceItem` objects

```
#include <iostream>
#include <string>
using namespace std;

// See figure above for class details
class GenericItem { ... };
class ProduceItem : public GenericItem { ... };

int main() {
   GenericItem miscItem;
   ProduceItem perishItem;

   miscItem.SetName("Crunchy Cereal");
   miscItem.SetQuantity(9);
   miscItem.PrintItem();

   perishItem.SetName("Apples");
   perishItem.SetQuantity(40);
   perishItem.SetExpiration("Dec 5, 2019");
   perishItem.PrintItem();
   cout << "  (Expires: " << perishItem.GetExpiration()
        << ")" << endl;

   return 0;
}
```

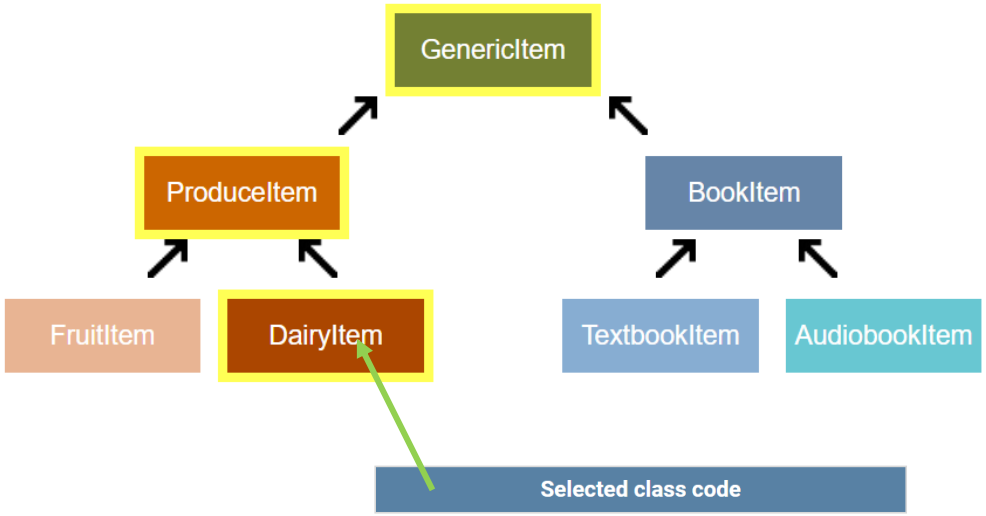


Inheritance Scenarios in C++

- A derived class can serve as a base class for another class.
  - Ex: `class FruitItem: public ProduceItem {...}` creates a derived class `FruitItem` from `ProduceItem`, which was derived from `GenericItem`.
- A class can serve as a base class for multiple derived classes.
  - Ex: `class FrozenFoodItem: public GenericItem {...}` creates a derived class `FrozenFoodItem` that inherits from `GenericItem`, just as `ProduceItem` inherits from `GenericItem`.
- A class may be derived from multiple classes.
  - Ex: `class House: public Dwelling, public Property {...}` creates a derived class `House` that inherits from base classes `Dwelling` and `Property`.

Inheritance Tree

Inheritance tree	Selected class pseudocode



Selected class code

```
class DairyItem : public ProduceItem {
   public:
      void SetPercentageFat(int newPercent)
         { percentageFat = newPercent; };
      int GetPercentageFat()
         { return percentageFat; };
   private:
      int percentageFat;
};
```

```
public:
    void SetName(string newName)
    void SetQuantity(int newQty)
    void PrintItem()
    void SetExpiration(string newDate)
    string GetExpiration()
    void SetPercentageFat(int newPercent)
    int GetPercentageFat()
private:
    string itemName;
    int itemQuantity;
    string expirationDate;
    int percentFat;
```

Example: Business and Restaurant

- Define the base class and derived class

```
#include <iostream>
#include <string>
using namespace std;

class Business {
   public:
      void SetName(string busName) {
         name = busName;
      }

      void SetAddress(string busAddress) {
         address = busAddress;
      }

      string GetDescription() const {
         return name + " -- " + address;
      }

   private:
      string name;
      string address;
};
```

```
class Restaurant : public Business {
   public:
      void SetRating(int userRating) {
         rating = userRating;
      }

      int GetRating() const {
         return rating;
      }

   private:
      int rating;
};
```

Example: Business and Restaurant

- Calling public functions defined in the base class

```
int main() {
   Business someBusiness;
   Restaurant favoritePlace;

   someBusiness.SetName("ACME");
   someBusiness.SetAddress("4 Main St");

   favoritePlace.SetName("Friends Cafe");
   favoritePlace.SetAddress("500 W 2nd Ave");
   favoritePlace.SetRating(5);

   cout << someBusiness.GetDescription() << endl;
   cout << favoritePlace.GetDescription() << endl;
   cout << "  Rating: " << favoritePlace.GetRating() << endl;

   return 0;
}
```

Example: Business and Restaurant

- Access the private data members in the derived class

```
#include <iostream>
#include <string>
using namespace std;

class Business {
   public:
      void SetName(string busName) {
         name = busName;
      }

      void SetAddress(string busAddress) {
         address = busAddress;
      }

      string GetDescription() const {
         return name + " -- " + address;
      }

   private:
      string name;
      string address;
};
```

```
class Restaurant : public Business {
   public:
      void SetRating(int userRating) {
         rating = userRating;
      }

      int GetRating() const {
         return rating;
      }

      void DisplayRestaurant() {
         cout << name << "-" << address
              << "-" << rating << endl;
      }

   private:
      int rating;
};
```

Example: Business and Restaurant

- Access the private data members in the derived class

```
int main() {
   Business someBusiness;
   Restaurant favoritePlace;

   someBusiness.SetName("ACME");
   someBusiness.SetAddress("4 Main St");

   favoritePlace.SetName("Friends Cafe");
   favoritePlace.SetAddress("500 W 2nd Ave");
   favoritePlace.SetRating(5);

   cout << someBusiness.GetDescription() << endl;
   favoritePlace.DisplayRestaurant();

   return 0;
}
```

Example: Business and Restaurant

- Define the base class and derived class

```
#include <iostream>
#include <string>
using namespace std;

class Business {
   public:
      void SetName(string busName) {
      string address;
};
```

```
class Restaurant : public Business {
   public:
      void SetRating(int userRating) {
         rating = userRating;
      }

      int GetRating() const {
```

```
main.cpp: In member function 'void Restaurant::DisplayRestaurant()':
main.cpp:35:18: error: 'std::string Business::name' is private within this context
   35 |          cout << name << "-" << address
      |                  ^~~~
main.cpp:20:14: note: declared private here
   20 |       string name;
      |              ^~~~
main.cpp:35:33: error: 'std::string Business::address' is private within this context
   35 |          cout << name << "-" << address
      |                                 ^~~~~~~
main.cpp:21:14: note: declared private here
   21 |       string address;
      |              ^~~~~~~
```

```
nt() {
-" << address
ting << endl;
```

Example: Business and Restaurant

- Modify the private data members to protected data members

```
#include <iostream>
#include <string>
using namespace std;

class Business {
   public:
      void SetName(string busName) {
         name = busName;
      }

      void SetAddress(string busAddress) {
         address = busAddress;
      }

      string GetDescription() const {
         return name + " -- " + address;
      }

   protected:
      string name;
      string address;
};
```

```
class Restaurant : public Business {
   public:
      void SetRating(int userRating) {
         rating = userRating;
      }

      int GetRating() const {
         return rating;
      }

      void DisplayRestaurant() {
         cout << name << "-" << address
              << "-" << rating << endl;
      }

   private:
      int rating;
};
```

Example: Business and Restaurant

- Modify the private data members to protected data members

```
#include <iostream>
#include <string>
using namespace std;

class Business {
   public:
      void SetName(string busName) {
         name = busName;
      }

      void S
         add
      }

      string
         ret
      }

   protected:
      string name;
      string address;
};
```

```
ACME -- 4 Main St
Friends Cafe-500 W 2nd Ave-5


...Program finished with exit code 0
Press ENTER to exit console.
```

```
class Restaurant : public Business {
   public:
      void SetRating(int userRating) {
         rating = userRating;
      }

      int GetRating() const {
         return rating;
      }

                                        {
                                  < address
                                  << endl;

      int rating;
};
```

Review - Access Specifiers

- `public`
  - Members are accessible from outside the class
- `private`
  - Members cannot be accessed (or viewed) from outside the class
  - Declaring data member with access specifier `private` is known as **data hiding** → name and rating are **encapsulated** (hidden) in the object
  - Have to implement member functions to access the private member data
- `protected`
  - Members cannot be accessed from outside the class, however, they can be accessed in inherited classes. (will be discussed later)

https://www.w3schools.com/cpp/cpp_access_specifiers.asp

Type of Inheritance

- When deriving a class from a base class, the base class may be inherited through `public` (most common), `protected`, and `private`

```
class Restaurant : public Business {
   public:
      void SetRating(int userRating) {
         rating = userRating;
      }

      int GetRating() const {
         return rating;
      }

      void DisplayRestaurant() {
         cout << name << "-" << address
              << "-" << rating << endl;
      }

   private:
      int rating;
};
```

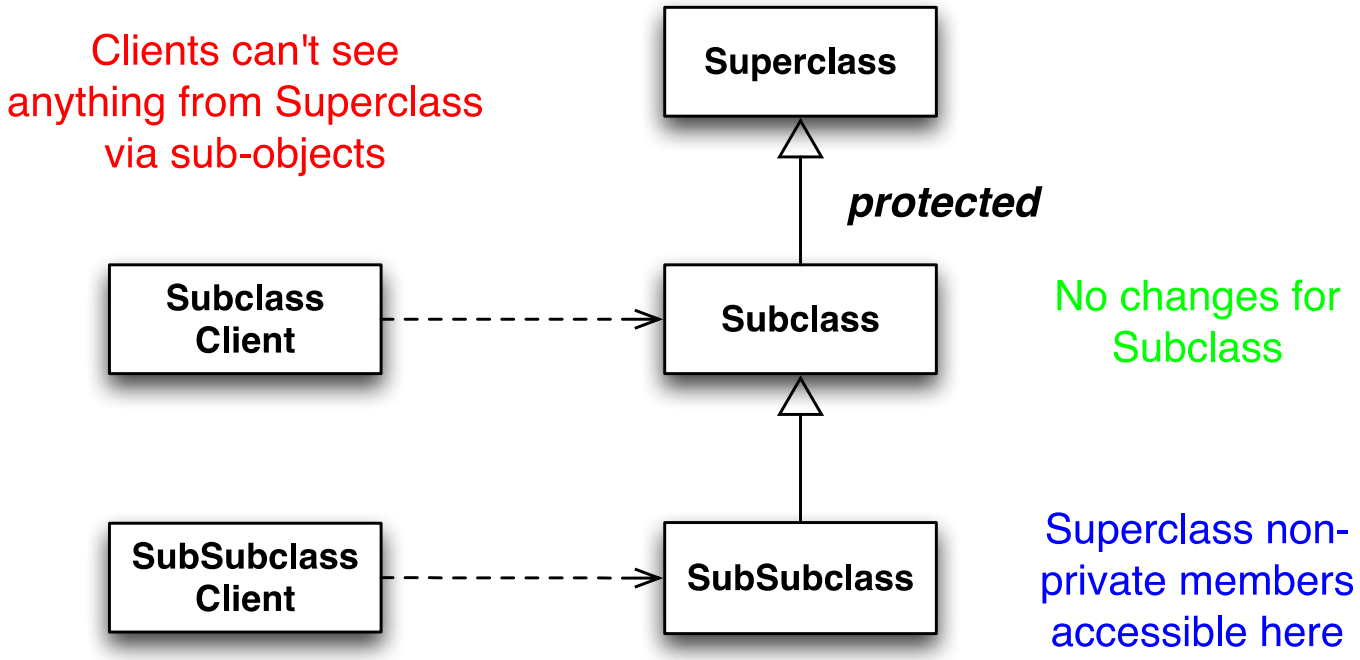
Type of Inheritance

- `public` inheritance
  - `public` members of the base class become `public` members of the derived class
  - `protected` members of the base class become `protected` member of the derived class

- `private` members of the base class are never accessible directly from a derived class, but can be accessed through calls to the `public` and `protected` members of the base class

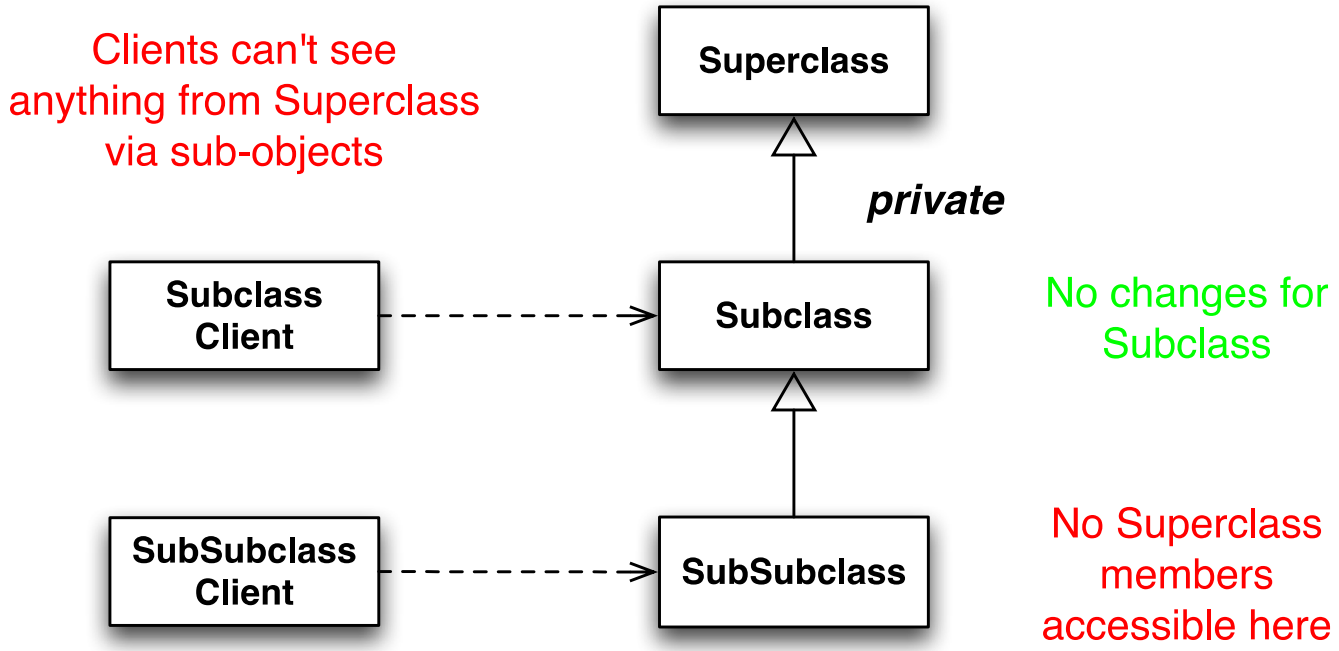
Type of Inheritance

- `protected` inheritance:
  - `public` and `protected` members of the base class become `protected` members of the derived class



Type of Inheritance

- `private` inheritance:
  - `public` and `protected` members of the base class become `private` members of the derived class



Type of Inheritance

- Summary

Base-class member-access specifier	Type of inheritance		
	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	`public` in derived class. Can be accessed directly by member functions, `friend` functions and nonmember functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`protected`	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`protected` in derived class. Can be accessed directly by member functions and `friend` functions.	`private` in derived class. Can be accessed directly by member functions and `friend` functions.
`private`	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.	Hidden in derived class. Can be accessed by member functions and `friend` functions through `public` or `protected` member functions of the base class.

Overriding

Example: Business and Restaurant

- Add the `GetDescription` method in the derived class

```
#include <iostream>
#include <string>
using namespace std;

class Business {
   public:
      ...
      string GetDescription() const {
         return name + " -- " + address;
      }
};
```

```
class Restaurant : public Business {
   public:
      ...
      string GetDescription() const {
         return name + " -- " + address +
            "\n  Rating: " + to_string(rating);
      }
};
```

```
int main() {
   Business someBusiness;
   Restaurant favoritePlace;
   ...
   cout << someBusiness.GetDescription() << endl;
   cout << favoritePlace.GetDescription() << endl;

   return 0;
}
```

  - Which `GetDescription` method will be called?

When a derived class defines a member function that has the same name and parameters as a base class's function, the member function is said to **override** the base class's function

Example: Business and Restaurant

- Calling the base class function in the derived class

```
class Restaurant : public Business {
   public:
      ...
      string GetDescription() const {
         return GetDescription() +
            "\n  Rating: " + to_string(rating);
      }
};
```

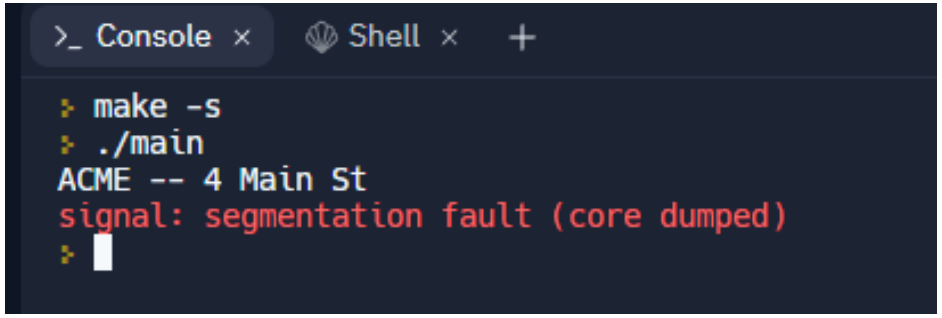
Example: Business and Restaurant

- Calling the base class function in the derived class

```
class Restaurant : public Business {
   public:
      ...
      string GetDescription() const {
         return GetDescription() +
            "\n  Rating: " + to_string(rating);
      }
};
```

Call itself

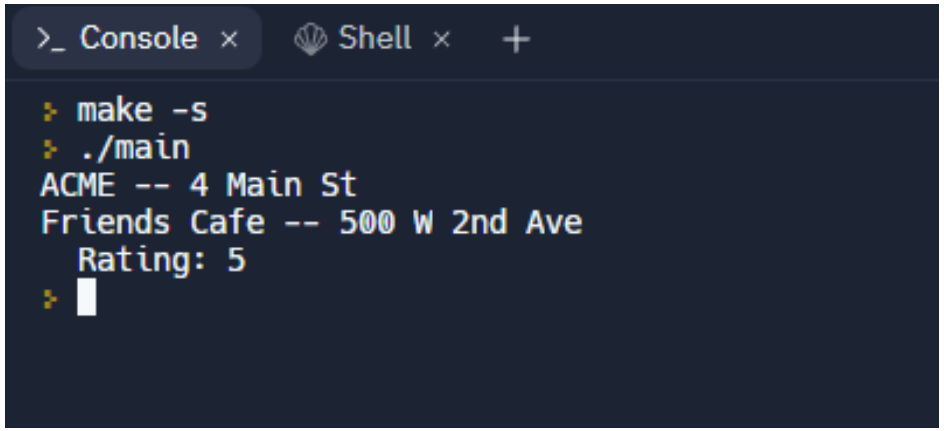
Solution: indicate that we want to call the `GetDescription` method of the `Business` base class



Example: Business and Restaurant

- Calling the base class function in the derived class

```
class Restaurant : public Business {
   public:
      ...
      string GetDescription() const {
         return Business::GetDescription() +
            "\n  Rating: " + to_string(rating);
      }
};
```



Overriding vs. Overloading

- Overriding
  - A derived class member function must have the same parameter types, number of parameters, and return value as the base class member function with the same name.
- Overloading
  - Functions with the same name must have different parameter types, number of parameters, or return values.
  - Is performed if derived and base member functions have different parameter types; the member function of the derived class does not hide the member function of the base class.
    → Unintentionally overloading
- Preventing unintentionally overloading
  - Use the `override` keyword: to explicitly state that a member function in a derived class overrides a virtual function in a base class

Will be discussed later

Polymorphism

Review - Object-Oriented Programming Major Concepts

- Encapsulation
  - Restrict access to methods and attributes in a class.
  - Hide the complex details from the users, and prevent data being modified by accident
- Inheritance
  - Define a class that inherits all the methods and attributes from another class
  - Makes the OOP code more modular, easier to reuse and build a relationship between classes
- **Polymorphism**
  - Use a single interface with different underlying forms such as data types or classes

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter07.03-Inheritance-Encapsulation-and-Polymorphism.html

Example: Business and Restaurant

- Define a `DriveTo` method as follow

```
void DriveTo(Business *businessPtr) {
  cout << "Driving to " << businessPtr->GetDescription() << endl;
}
```

- Is the following statement legal?

```
int main() {
   Business someBusiness;
   Restaurant favoritePlace;

   someBusiness.SetName("ACME");
   someBusiness.SetAddress("4 Main St");

   favoritePlace.SetName("Friends Cafe");
   favoritePlace.SetAddress("500 W 2nd Ave");
   favoritePlace.SetRating(5);

   DriveTo(&favoritePlace);

   return 0;
}
```

Example: Business and Restaurant

- Define a `DriveTo` method as follow

```
void DriveTo(Business *businessPtr) {
  cout << "Driving to " << businessPtr->GetDescription() << endl;
}
```

- Which `GetDescription` method the `businessPtr` will call?

```
int main() {
   Business someBusiness;
   Restaurant favoritePlace;

   someBusiness.SetName("ACME");
   someBusiness.SetAddress("4 Main St");

   favoritePlace.SetName("Friends Cafe");
   favoritePlace.SetAddress("500 W 2nd Ave");
   favoritePlace.SetRating(5);

   DriveTo(&favoritePlace);

   return 0;
}
```

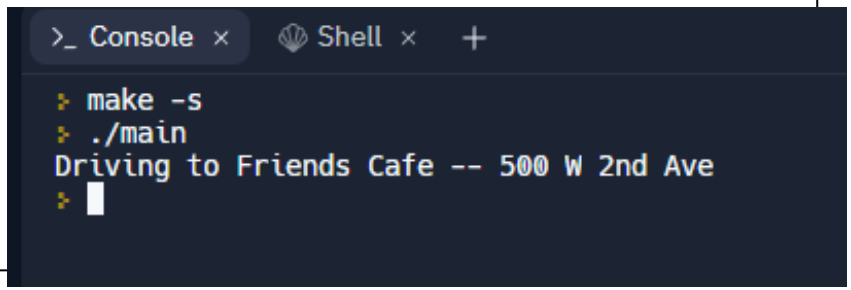
Example: Business and Restaurant

- Define a `DriveTo` method as follow

```
void DriveTo(Business *businessPtr) {
  cout << "Driving to " << businessPtr->GetDescription() << endl;
}
```

- Which `GetDescription` method the `businessPtr` will call?

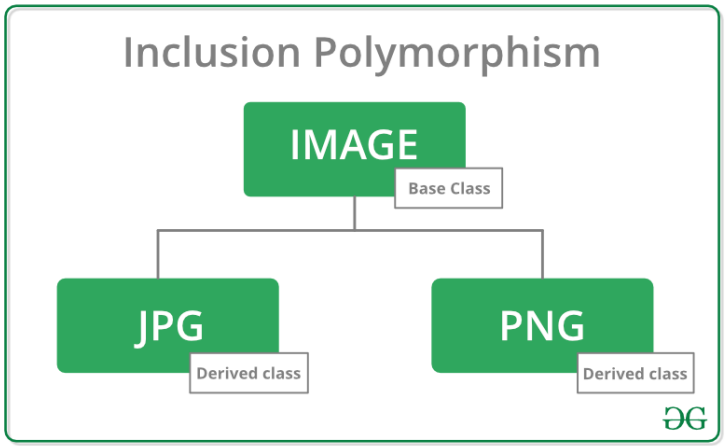
```
int main() {
   Business someBusiness;
   Restaurant favoritePlace;

   someBusiness.SetName("ACME");
   someBusiness.SetAddress("4 Main St");

   favoritePlace.SetName("Friends Cafe");
   favoritePlace.SetAddress("500 W 2nd Ave");
   favoritePlace.SetRating(5);

   DriveTo(&favoritePlace);

   return 0;
}
```



Polymorphism

- Polymorphic
  - A word from the Greek meaning: many forms
  - To describe a wide variety of programming-language features
- Subtype (inclusion) polymorphism
  - A function or operator exhibits subtype polymorphism if one or more or its parameter types have subtypes

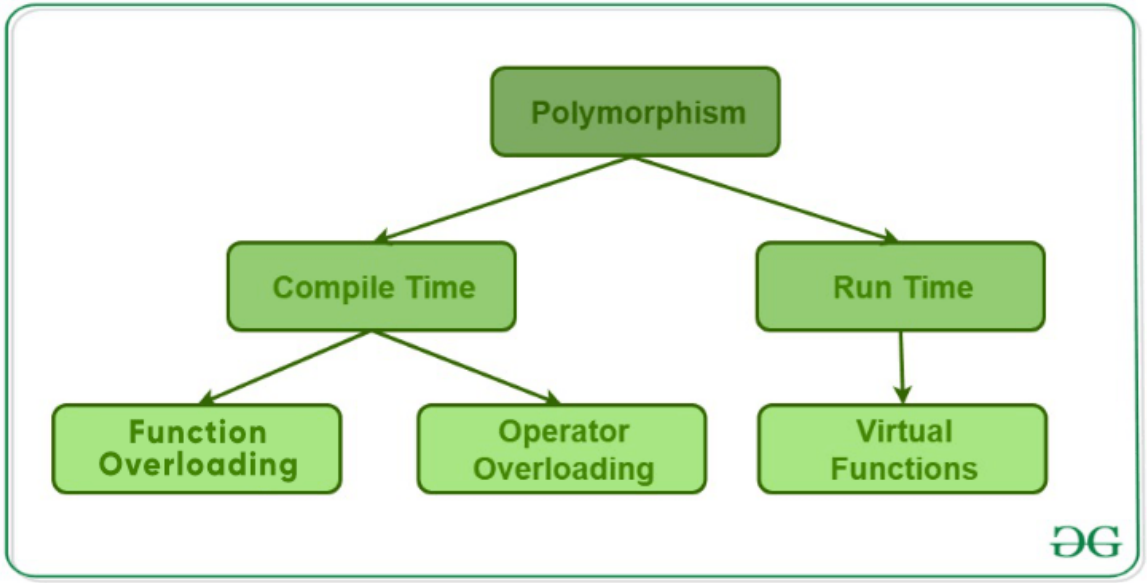
```
public class Car {
  public void brake() { ... }
}

public class ManualCar extends Car {
  public void clutch() { ... }
}

void g(Car z) {
  z.brake();
}

void f(Car x, ManualCar y) {
  g(x);
  g(y);
}
```



https://www.geeksforgeeks.org/ad-hoc-inclusion-parametric-coercion-polymorphisms/

Polymorphism

- Compile-time polymorphism
  - When the compiler determines which function to call at compile-time.
- Runtime polymorphism
  - The compiler is unable to determine which function to call at compile-time
  - The determination is made while the program is running.



https://www.geeksforgeeks.org/cpp-polymorphism/

Polymorphism

- Compile-time polymorphism

```
void DriveTo(string restaurant) {
   cout << "Driving to " << restaurant << endl;
}

void DriveTo(Restaurant restaurant) {
   cout << "Driving to " << restaurant.GetDescription() << endl;
}

int main() {
   DriveTo("Big Mac's");  // Call string version
}
```

Polymorphism

- Runtime polymorphism

```
void DriveTo(Business* businessPtr) {
   cout << "Driving to " << businessPtr->GetDescription() << endl;
}

int main() {
   int index;
   vector<Business*> businessList;
   Business* businessPtr;
   Restaurant* restaurantPtr;
   ...
   businessList.push_back(businessPtr);
   businessList.push_back(restaurantPtr);

   index = rand() % businessList.size();
   DriveTo(businessList.at(index));
}
```

Calls Restaurant's GetDescription() for Restaurant pointer

Virtual Functions

- Runtime polymorphism only works when an overridden member function in a base class is virtual
- Virtual Function
  - A member function that may be overridden in a derived class and is used for runtime polymorphism
  - Is declared by prepending the keyword `virtual`.
    Ex: `virtual string GetDescription()`
- At runtime, when a virtual function is called using a pointer, the correct function to call is dynamically determined based on the actual object type to which the pointer or reference refers
- Virtual table
  - The compiler creates a virtual table that allows the computer to quickly lookup which function to call at runtime.
  - Contains an entry for each virtual function with a function pointer that points to the most-derived function that is accessible to each class

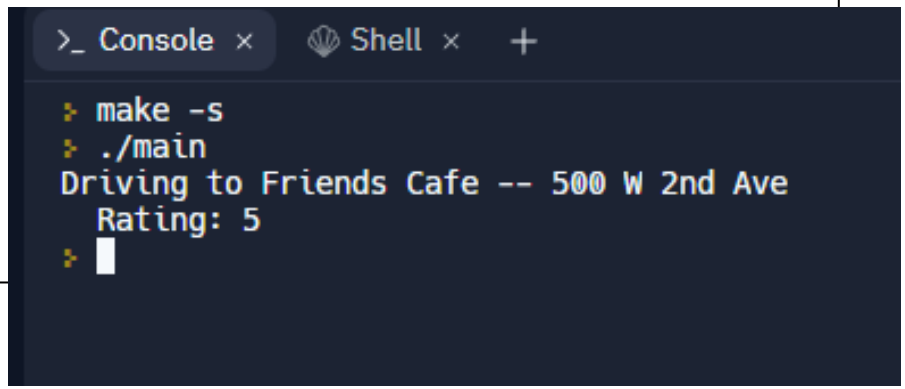
Example: Business and Restaurant

- Set `GetDescription` is a virtual function

```
class Business {
   public:
      ...
      virtual string GetDescription() const {
         return name + " -- " + address;
      }
};
```

- Execute the `main` function again

```
int main() {
   Business someBusiness;
   Restaurant favoritePlace;
   ...
   DriveTo(&favoritePlace);

   return 0;
}
```



Review - Overriding vs. Overloading

- Overriding
  - A derived class member function must have the same parameter types, number of parameters, and return value as the base class member function with the same name.
- Overloading
  - Functions with the same name must have different parameter types, number of parameters, or return values.
  - Is performed if derived and base member functions have different parameter types; the member function of the derived class does not hide the member function of the base class.
    → Unintentionally overloading
- Preventing unintentionally overloading
  - Use the `override` keyword: to explicitly state that a member function in a derived class overrides a virtual function in a base class

Will be discussed later

Example: Business and Restaurant

- Add `override` keyword on the `GetDescription` method in the derived class

```
class Restaurant : public Business {
   public:
      ...
      string GetDescription() const override {
         return name + " -- " + address +
            "\n  Rating: " + to_string(rating);
      }
};
```

- Add a parameter in the `GetDescription` method in the derived class to see if there is any error message

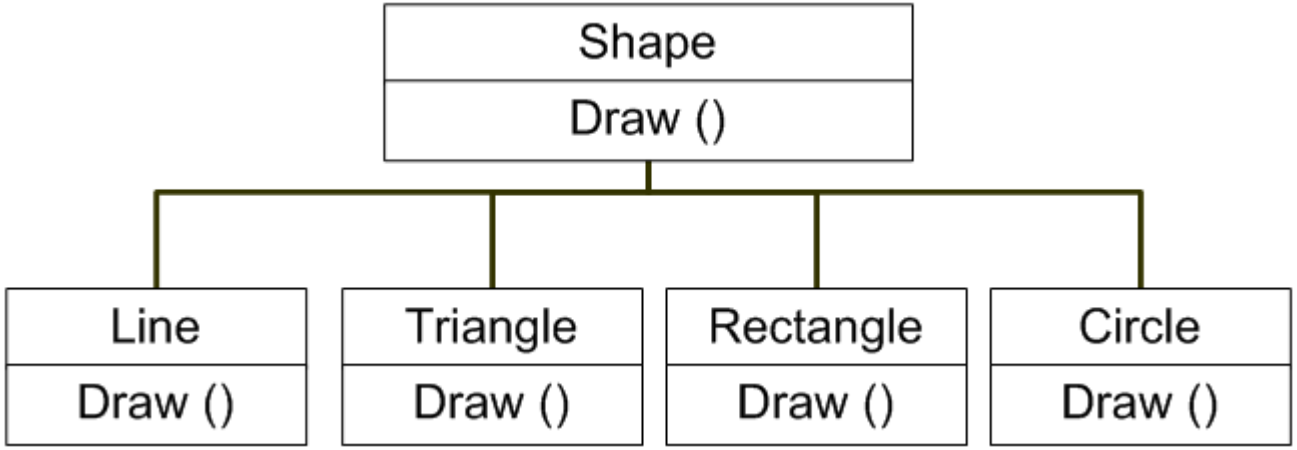
```
make -s
inheritance.cpp:35:45: error: non-virtual member function marked 'override' hides virtual me
function
      string GetDescription(string s) const override {
                                            ^

inheritance.cpp:16:22: note: hidden overloaded virtual function 'Business::GetDescription' declar
ed here: different number of parameters (0 vs 1)
      virtual string GetDescription() const {
                     ^

1 error generated.
make: *** [Makefile:17: inheritance.o] Error 1
exit status 2
```

Example: Business and Restaurant

```
int main() {
   unsigned int i;
   vector<Business*> businessList;
   Business* businessPtr;
   Restaurant* restaurantPtr;

   businessPtr = new Business;
   businessPtr->SetName("ACME");
   businessPtr->SetAddress("4 Main St");

   restaurantPtr = new Restaurant;
   restaurantPtr->SetName("Friends Cafe");
   restaurantPtr->SetAddress("500 2nd Ave");
   restaurantPtr->SetRating(5);

   businessList.push_back(businessPtr);
   businessList.push_back(restaurantPtr);

   for (i = 0; i < businessList.size(); ++i) {
      cout << businessList.at(i)->GetDescription() << endl;
   }

   return 0;
}
```

Pure Virtual Function

- Condition
  - A base class should not provide a definition for a member function
  - All derived classes must provide a definition



- Define a pure virtual function
  - Declare a virtual function with the virtual keyword and is assigned with 0
  - Example: `virtual string GetHours() const = 0;`

https://medium.com/@mathurk29/virtual-functions-bdb0ca95b1ca

Pure Virtual Function

- Example

```
class Business {
public:
   void SetName(string busName) {
      name = busName;
   }

   void SetAddress(string busAddress) {
      address = busAddress;
   }

   virtual string GetDescription() const {
      return name + " -- " + address;
   }

   virtual string GetHours() const = 0;     // pure virtual function

protected:
   string name;
   string address;
};
```

Pure Virtual Function

- Abstract base class
  - A class that has at least one pure virtual function
  - **Can not be instantiated**

```
class Business {
   public:
      ...
      virtual string GetHours() const = 0;
};
```

```
int main() {
   Business b;

   return 0;
}
```

```
Console × Shell × +

make -s
inheritance.cpp:56:13: error: variable type 'Business' is an abstract class
   Business b;
            ^

inheritance.cpp:20:22: note: unimplemented pure virtual method 'GetHours' in 'Business'
      virtual string GetHours() const = 0;
                     ^

1 error generated.
make: *** [Makefile:17: inheritance.o] Error 1
exit status 2
```

Pure Virtual Function

- Abstract base class
  - A class that has at least one pure virtual function
  - Can not be instantiated
  - Can be the reference of the derived class instances

```
class Restaurant : public Business {
   public:
      ...
      string GetHours() const {
        return "time";
      }

};
```

```
int main() {
   Restaurant r;
   Business *b = &r;

   return 0;
}
```

Pure Virtual Function

Virtual function	Pure virtual function
A virtual function is a member function of base class which can be redefined by derived class.	A pure virtual function is a member function of base class whose only declaration is provided in base class and should be defined in derived class otherwise derived class also becomes abstract.
Classes having virtual functions are not abstract.	Base class containing pure virtual function becomes abstract.

Syntax:
`virtual<func_type><func_name>()`
`{`
`  // code`
`}`
	Syntax:
	`virtual<func_type><func_name>()`
	`  = 0;`

Definition is given in base class.	No definition is given in base class.
Base class having virtual function can be instantiated i.e. its object can be made.	Base class having pure virtual function becomes abstract i.e. it cannot be instantiated.
If derived class do not redefine virtual function of base class, then it does not affect compilation.	If derived class do not redefine virtual function of base class, then no compilation error but derived class also becomes abstract just like the base class.
All derived class may or may not redefine virtual function of base class.	All derived class must redefine pure virtual function of base class otherwise derived class also becomes abstract just like base class.

https://www.geeksforgeeks.org/difference-between-virtual-function-and-pure-virtual-function-in-c/

Abstract Classes

- Abstract class
  - Is too generic (abstract) to define real objects
    → Cannot be instantiated as an object
  - Specifies how the subclass must be implemented
- Concrete class
  - Can be used to instantiate objects
  - Define or inherit implements for every member function they declare

Wrap Up

Review - Composition Example

- The 'has-a' relationship
  - A `MotherInfo` object 'has a' string object and 'has a' vector of `ChildInfo` objects

```
class ChildInfo {
   string firstName;
   string birthDate;
   string schoolName;

   ...
};

class MotherInfo {
   string firstname;
   string birthDate;
   string spouseName;
   vector<ChildInfo> childrenData;

   ...
};
```

Inheritance

- The 'is-a' relationship.
  - A `MotherInfo` object 'is a' kind of `PersonInfo`.
  - The `MotherInfo` class thus inherits from the `PersonInfo` class.
  - Likewise for the `ChildInfo` class.

```
class PersonInfo {
   string firstName;
   string birthDate;

   ...
};

class ChildInfo : public PersonInfo {
   string schoolName;

   ...
};

class MotherInfo : public PersonInfo {
   string spouseName;
   vector<ChildInfo> childrenData;

   ...
};
```

More Information

- https://www.geeksforgeeks.org/inheritance-in-c/
- https://isocpp.org/wiki/faq/strange-inheritance
- https://www.geeksforgeeks.org/multiple-inheritance-in-c/