# More on Class Design

# Today's Objectives

- Continue Case Study: The **string** class
- Understand **object composition** (10.8)
- Passing objects to functions by **value** and by **reference** (10.3)
- **Constant** functions (10.6)
- **Arrays of objects** (10.4)

# The string class is a great example of Object Oriented Design

## STRING

**String member variables are private.**

**SUBSET OF THE STRING CLASS METHODS…**

Member Functions

- size or length - Returns the number of characters in string.
  - at - Accesses specified character with bounds checking.
- empty - Tests if a string is empty
- clear - Clears the contents
- insert - Inserts characters or string n times
- erase - Erases characters
- find - Search within a string
- append - Appends characters to the end
- replace - Replace characters of a string with another string
- resize - Changes the number of stored characters
- push_back - appends a character
- swap - Swaps the contents with another string

Member Operators

- operator[] - Accesses specified character
- operator+= Append to a a string
- operator== Check equality of strings, also !=, <, >, <=, >=, etc.

The properties (internal representation) are mostly "hidden" from the user.

The string class has a nice "interface" that we can use to manipulate string objects

# Other useful string class methods

substr(index, length): return the substring of length from specified index

at(index): retrieve a **character** at a specified index

erase(index, length): delete part of the string of length from specified index

insert(index, string): insert into a string

replace(index, n, string): replace part of string

clear(): clear the string

empty(): test if a string is empty.

# string::find() methods

find(string): finds the **string** argument in the string and return its starting index position

If not found, returns **string::npos** a **static variable**!

find(string, pos): finds the **string** argument starting at position **pos**

find(char, pos): finds the character **char** starting at position **pos**

# string::compare() method

```
string s3("welcome");
string s4("welcomg");
cout << s3.compare(s4) << endl; //-1
cout << s4.compare(s3) << endl;  //1
cout << s3.compare("welcome") << endl;  //0
```

**Lexicographical** comparison

The compare function returns a **value greater than 0**, **equal to 0**, or **less than 0**.

# Exercise - What other member functions does string have?

- Check **online documentation** to see the member functions that string has. http://www.cplusplus.com/reference/string/string/
- What do the following string member functions do?
  - `length()`
  - `push_back()`
  - `copy()`

# String has also redefined operators in its class definition

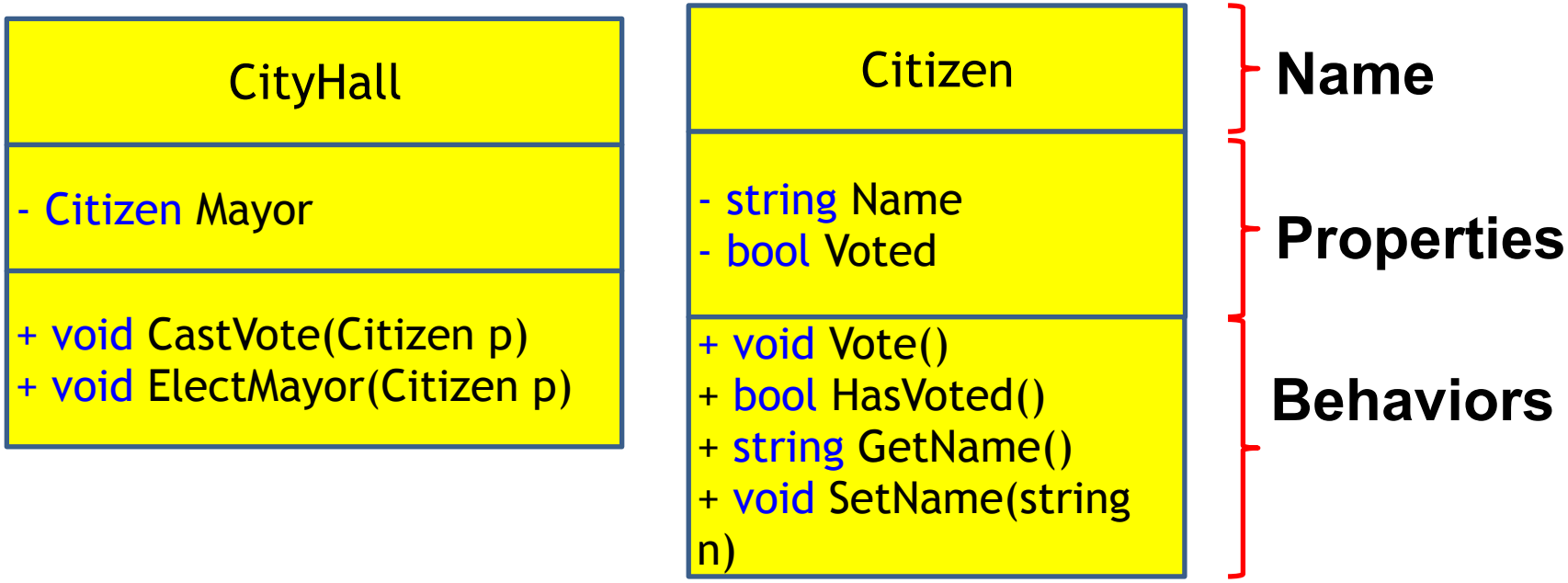
| OPERATOR | DESCRIPTION |
|---|---|
| [] | Accesses characters using the array subscript operators. |
| = | Copies the contents of one string to the other. |
| += | Appends a string. |
| + | Concatenates two strings into a new string. |
| ==, !=, <, <=, >, >= | Six relational operators for comparing strings. |
| << | Sends string contents to output stream (cout) |
| >> | Sends contents of stream to string (cin) |

Next topic:

# OBJECT COMPOSITION

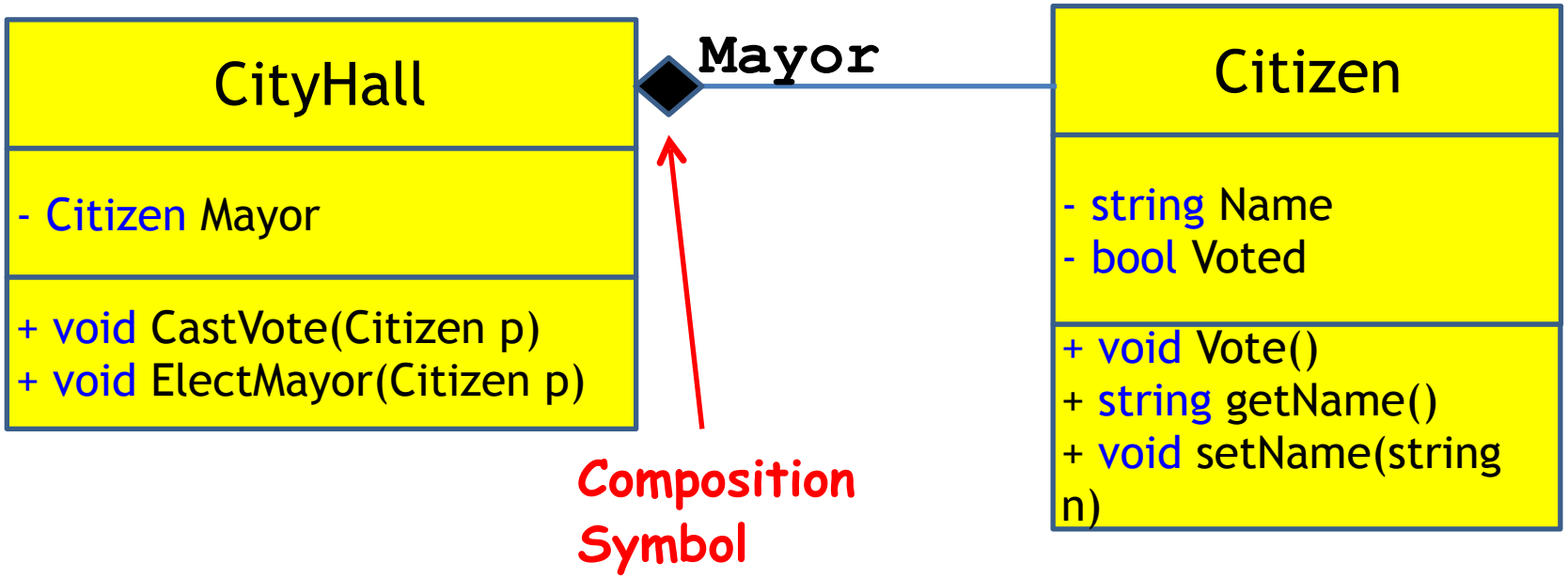
# Section 1&2 is here

# UML Class Diagrams




What is the **relationship** between CityHall and Citizen?

# Object Aggregation and Composition Relationships in UML

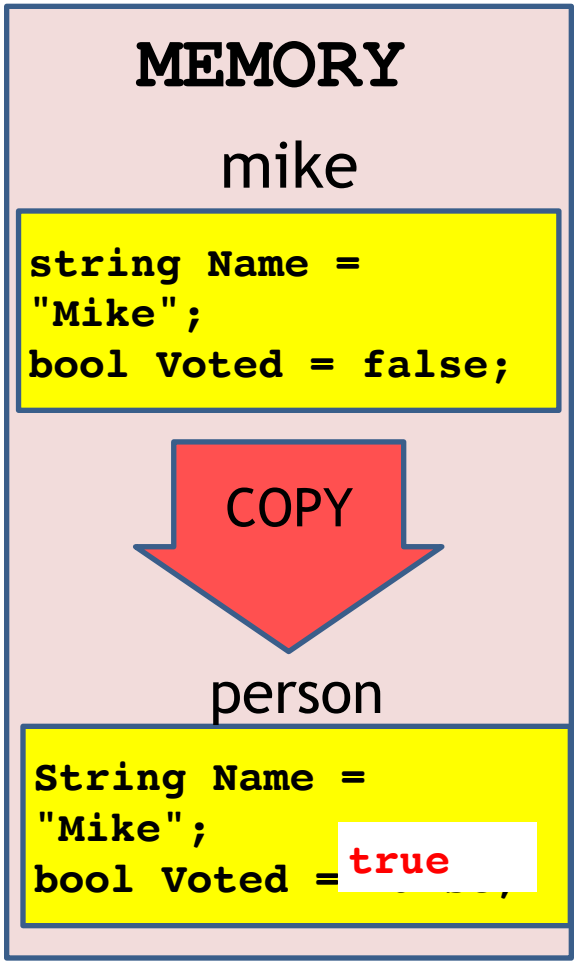



**Composition describes a "HAS-A" relationship**

Is there a HAS-A relationship in faculty and student? Update that UML

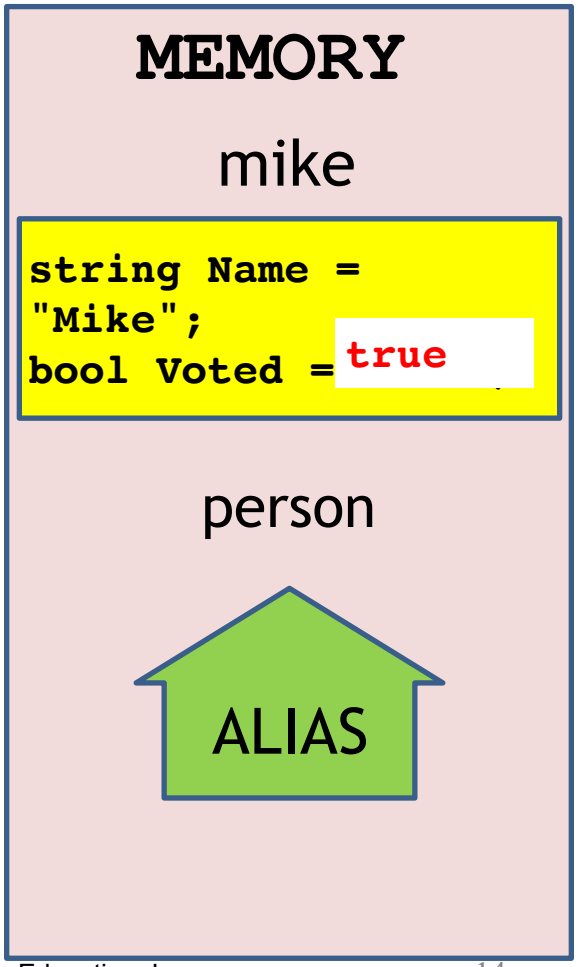
# Passing an **object** by value makes a copy of the object to the function parameter

```
class CityHall {

   void CastVote(Citizen person)
   {
      person.Vote();
   }
};
void main()
{    CityHall Spokane;
     Citizen mike("Mike");
     Spokane.CastVote(mike);
}
```



# Passing an **object** by reference allows the function parameter to act like an "alias" for the argument

```
class CityHall {

   void CastVote(Citizen &person)
    {
      person.Vote();
    }
};
void main()
{    CityHall Spokane;
     Citizen mike("Mike");
     Spokane.CastVote(mike);
}
```



# Reference Parameters Take-Away

- Are simply new names for the argument variables (i.e. alias )
- **MAY BE USED TO MODIFY THE ARGUMENTS VALUE IN THE FUNCTION**

# Value Parameters Take-Away

- Are copies of the argument variables
- **MAY BE USED TO PREVENT THE ARGUMENTS FROM BEING MODIFIED IN THE FUNCTION**
- **Drawback: copying into a new variable causes some overhead**

# BUT, WHAT IF I WANTED TO AVOID THAT OVERHEAD?

# Defining an object as a const reference prevents the method from changing the parameter

const protects the reference parameter s from modification in the function.

```
class CityHall {

  void CastVote(const Citizen &person)
  {
     person.Vote();
  }
};
void main()
{    CityHall Spokane;
     Citizen mike("Mike");
     Spokane.CastVote(mike);
}
```

# You can also make a member function const so we can't accidently change a member variable!

```
class Citizen
{
private:
    string Name;
    bool Voted;
public:
     …
    string GetName() const
    {
         Name = "Donald"; // ERROR! Const function
                          // can't change an member variable!
        return Name;      //OK
    }
};
```

const **prevents** getName() from changing any instance variables.

# In-class Exercise

- Return to your Faculty and Student running example

- Are there places for passing by reference, rather than value? Consider additional functions that might be useful for your classes
- Are there places for passing by const reference?
- Are there functions that could be marked const?

- In all cases, be able to explain why the change makes sense

Next topic:

# ARRAYS OF OBJECTS!

# Array Analogy

```
int x;
```

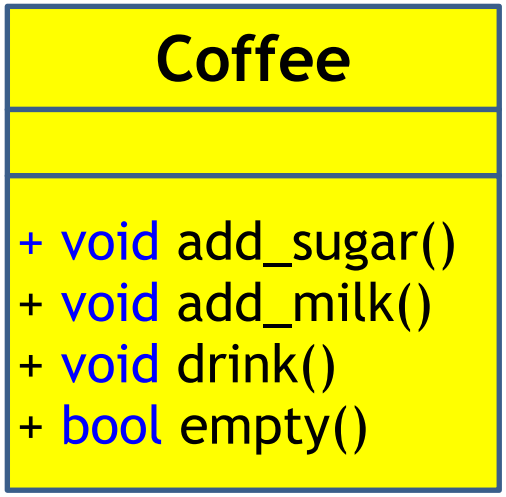
```
int x[6];
```


```
x[0]=0;
```

index starts at 0

```
int y=x[5];
```

A class is just a data type, so we can declare **an array of objects** of that class type.




```
class Coffee
{
public:
      void add_sugar();
      void add_milk();
      void drink();
      bool empty();
};
```

```
Coffee cups[6]; // 6 cups of coffee
cups[0].add_sugar();
cups[1].add_milk();
cups[1].drink();
```

We can initialize **an array of citizens** to represent a population.

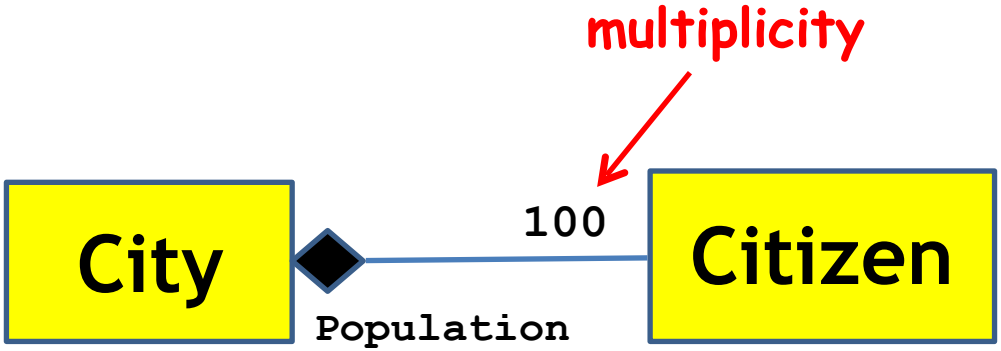
```
Citizen population[3] = {
      Citizen("Mike"),
      Citizen("Jim"),
      Citizen("Sally");
};
```

Calling constructor with 1 parameter!

We can create **an array of citizens** to represent a population.

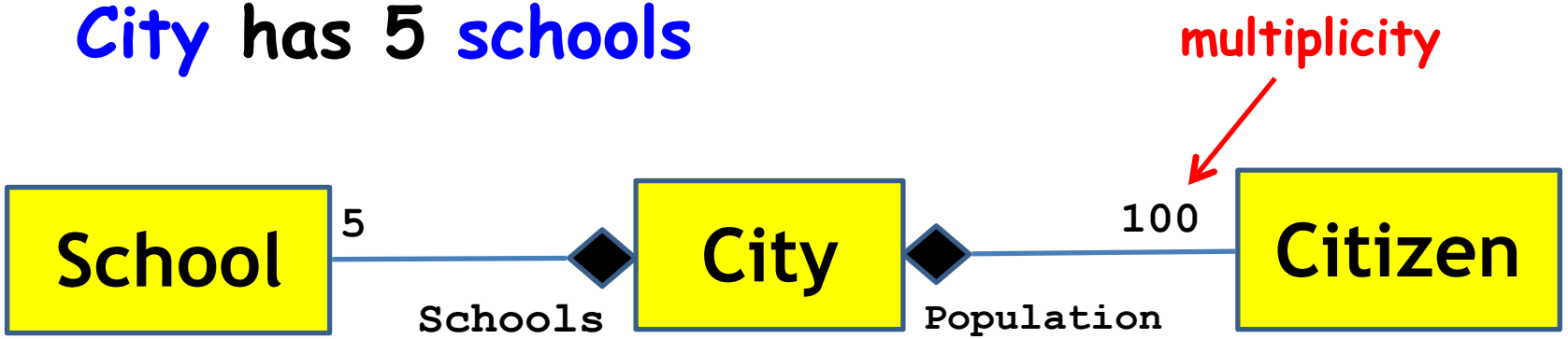
```
const int CITYSIZE=100;
Citizen population[CITYSIZE];

for (int i=0; i<CITYSIZE; i++ )
{
  cout << "Enter name of citizen "
       << i+1 << ":";
  string name;
  getline( cin, name );
  population[i].SetName(name);
}
```

# City has a population of 100 citizens



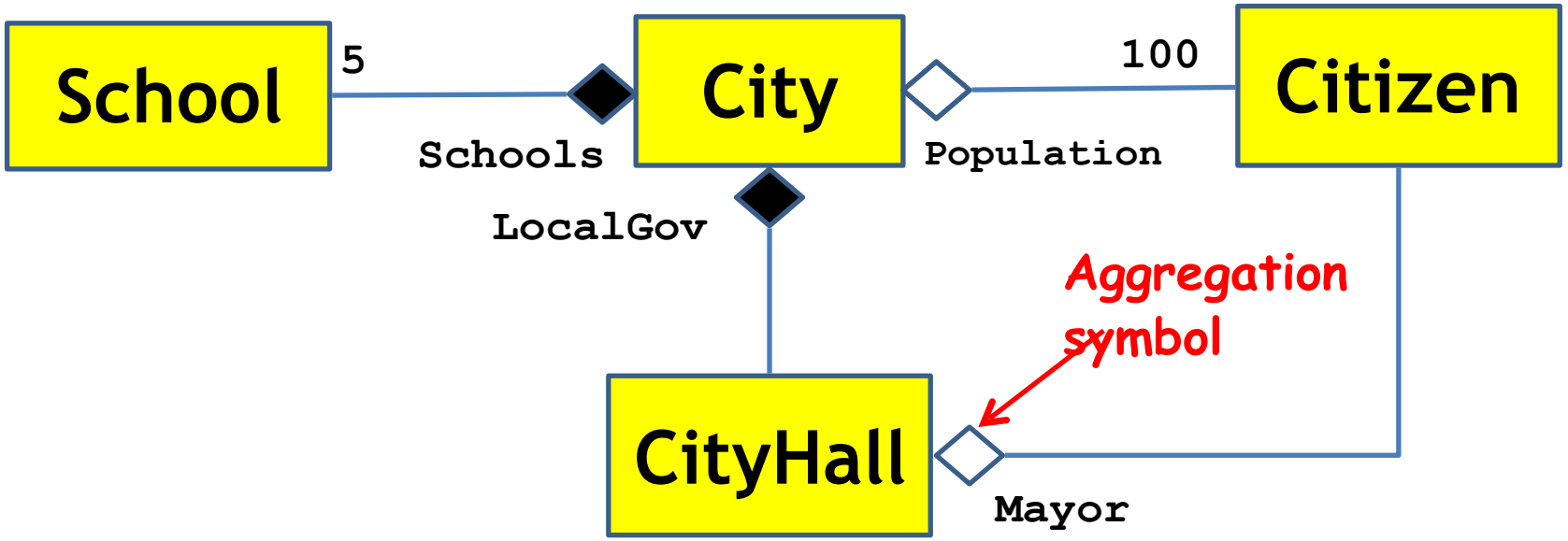
# City has a population of 100 citizens

## City has 5 schools



# City also has a City Hall

- Composition: Defines an owner of the object. The object goes away when the owner does.
- Aggregation: The object uses another object but does own it.

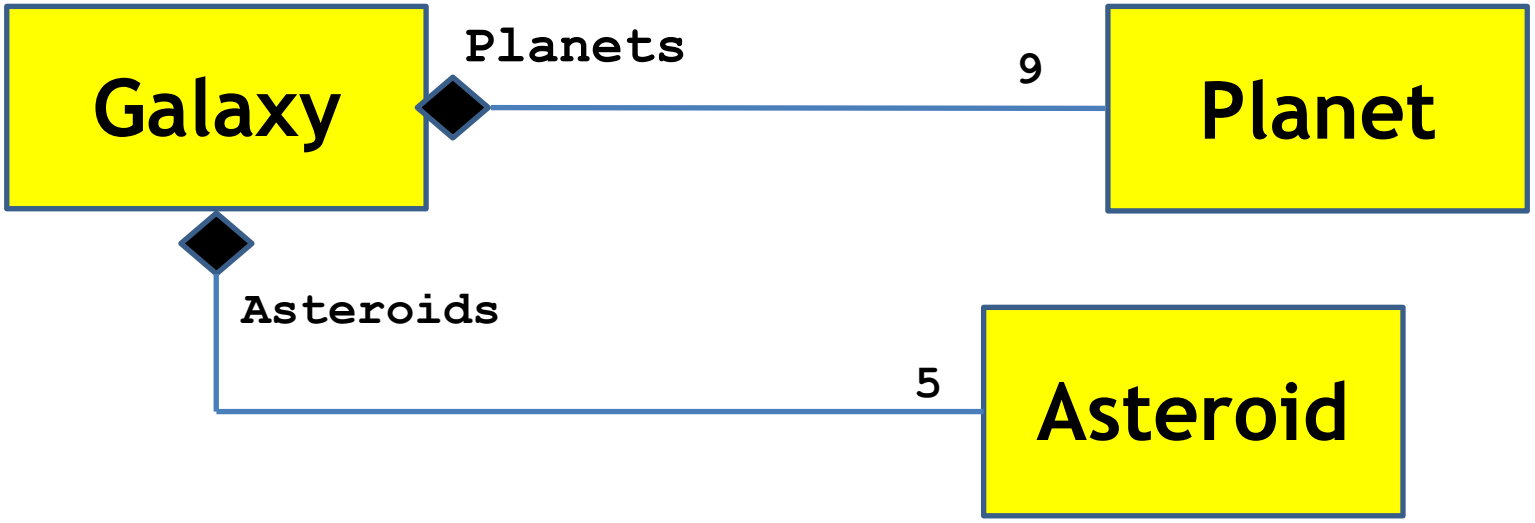



**Aggregation describes a non-exclusive "HAS-A" relationship**

# We can now translate our UML diagram into a class definition …

```
class City
{
    Population Citizen[100];
    School School[5];
    CityHall LocalGov;
};
```

# Exercise: "Galaxy has 9 planets and 5 asteroids"




1. Define **3 classes**: **Planet**, **Asteroid**, and **Galaxy**

2. Define the relationship shown in the UML diagram above in your **Galaxy** class