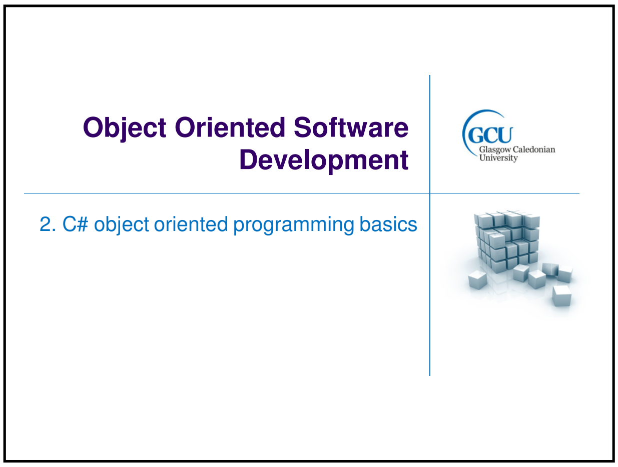
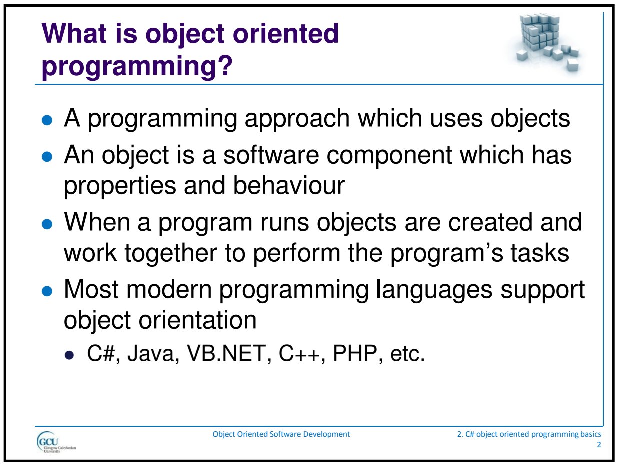


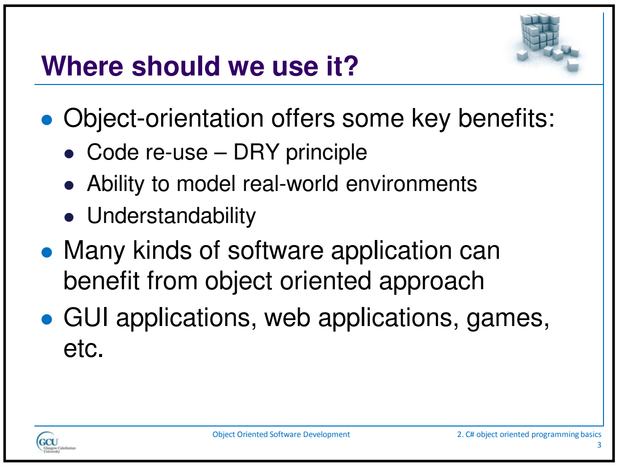
Object Oriented Software Development

2. C# object oriented programming basics



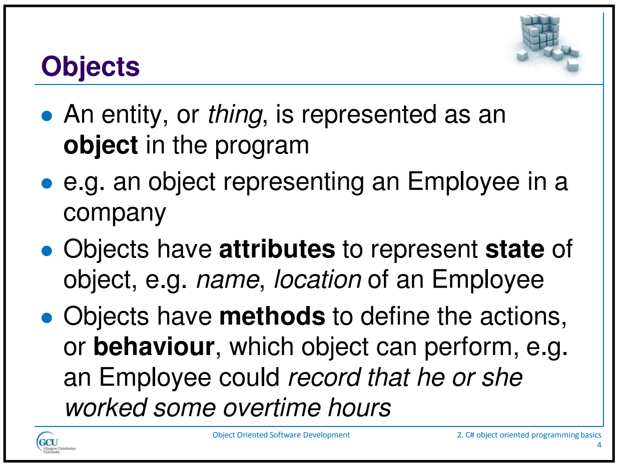
What is object oriented programming?

- A programming approach which uses objects
- An object is a software component which has properties and behaviour
- When a program runs objects are created and work together to perform the program's tasks
- Most modern programming languages support object orientation
  - C#, Java, VB.NET, C++, PHP, etc.



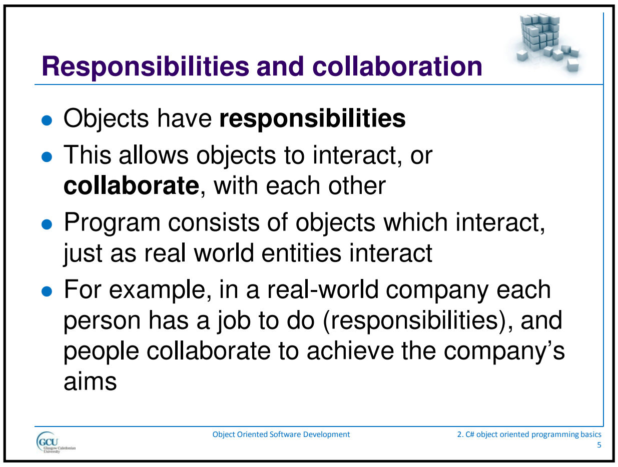
Where should we use it?

- Object-orientation offers some key benefits:
  - Code re-use – DRY principle
  - Ability to model real-world environments
  - Understandability
- Many kinds of software application can benefit from object oriented approach
- GUI applications, web applications, games, etc.



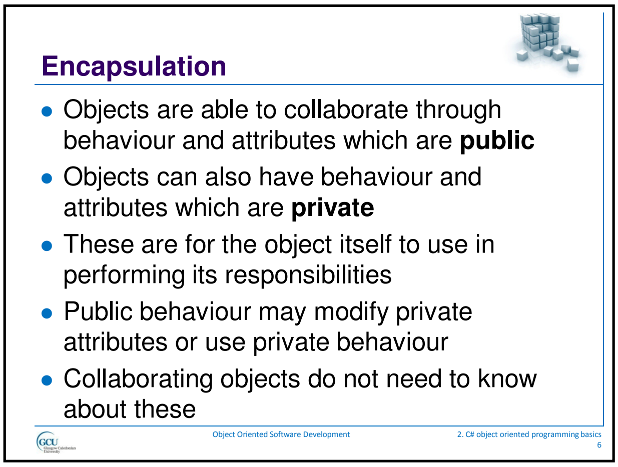
Objects

- An entity, or *thing*, is represented as an **object** in the program
- e.g. an object representing an Employee in a company
- Objects have **attributes** to represent **state** of object, e.g. *name*, *location* of an Employee
- Objects have **methods** to define the actions, or **behaviour**, which object can perform, e.g. an Employee could *record that he or she worked some overtime hours*



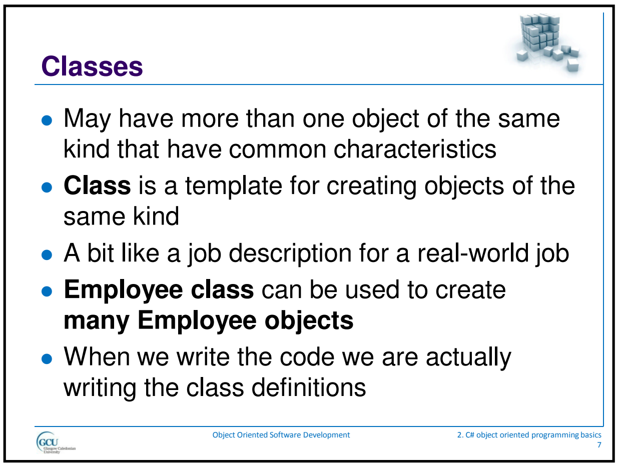
Responsibilities and collaboration

- Objects have **responsibilities**
- This allows objects to interact, or **collaborate**, with each other
- Program consists of objects which interact, just as real world entities interact
- For example, in a real-world company each person has a job to do (responsibilities), and people collaborate to achieve the company's aims



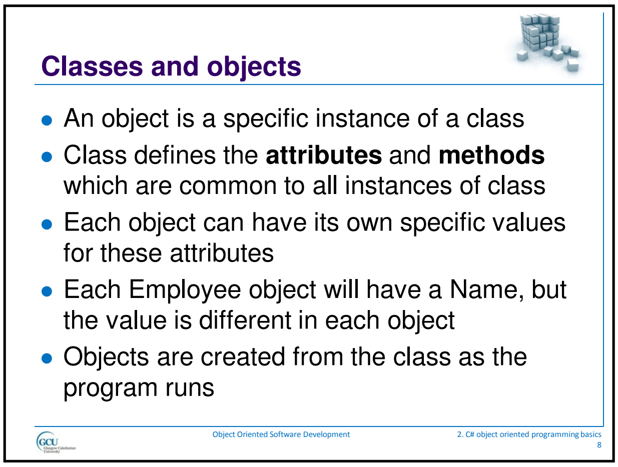
Encapsulation

- Objects are able to collaborate through behaviour and attributes which are **public**
- Objects can also have behaviour and attributes which are **private**
- These are for the object itself to use in performing its responsibilities
- Public behaviour may modify private attributes or use private behaviour
- Collaborating objects do not need to know about these



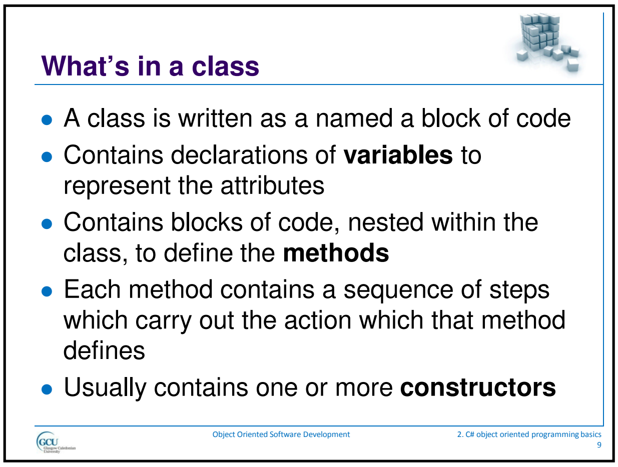
Classes

- May have more than one object of the same kind that have common characteristics
- **Class** is a template for creating objects of the same kind
- A bit like a job description for a real-world job
- **Employee class** can be used to create **many Employee objects**
- When we write the code we are actually writing the class definitions



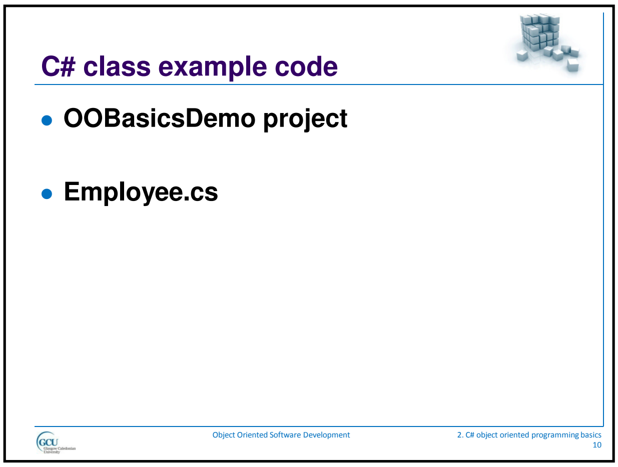
Classes and objects

- An object is a specific instance of a class
- Class defines the **attributes** and **methods** which are common to all instances of class
- Each object can have its own specific values for these attributes
- Each Employee object will have a Name, but the value is different in each object
- Objects are created from the class as the program runs



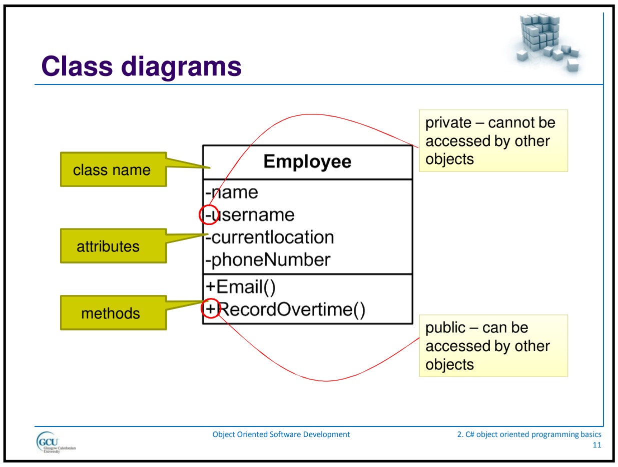
What’s in a class

- A class is written as a named a block of code
- Contains declarations of **variables** to represent the attributes
- Contains blocks of code, nested within the class, to define the **methods**
- Each method contains a sequence of steps which carry out the action which that method defines
- Usually contains one or more **constructors**



C# class example code

- OOBasicsDemo project
- Employee.cs



Class diagrams

class name

attributes

methods

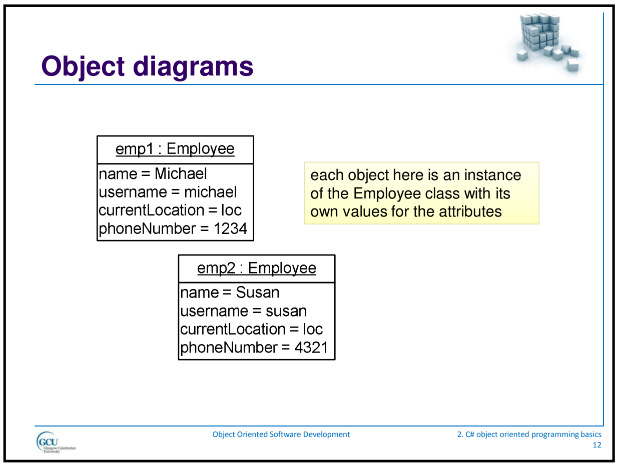
Employee

-name
-username
-currentlocation
-phoneNumber

+Email()
+RecordOvertime()

private – cannot be accessed by other objects

public – can be accessed by other objects



Object diagrams

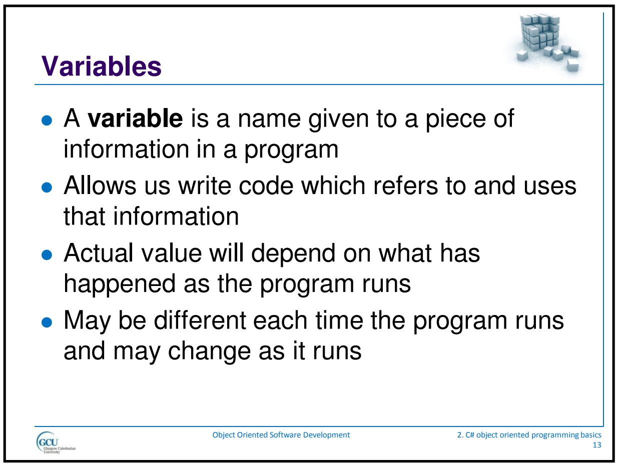
emp1 : Employee

name = Michael
username = michael
currentLocation = loc
phoneNumber = 1234

each object here is an instance of the Employee class with its own values for the attributes

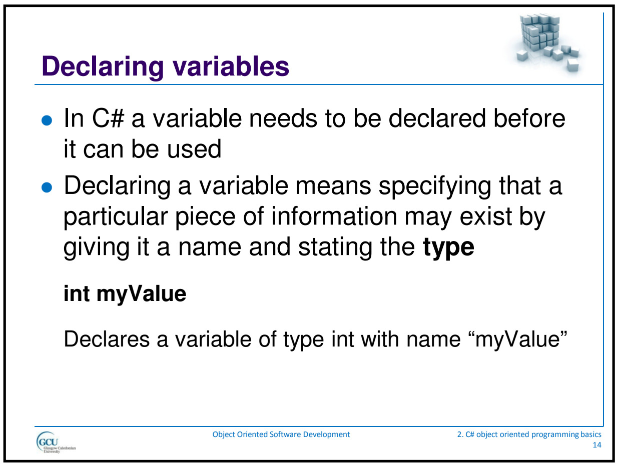
emp2 : Employee

name = Susan
username = susan
currentLocation = loc
phoneNumber = 4321



Variables

- A **variable** is a name given to a piece of information in a program
- Allows us write code which refers to and uses that information
- Actual value will depend on what has happened as the program runs
- May be different each time the program runs and may change as it runs

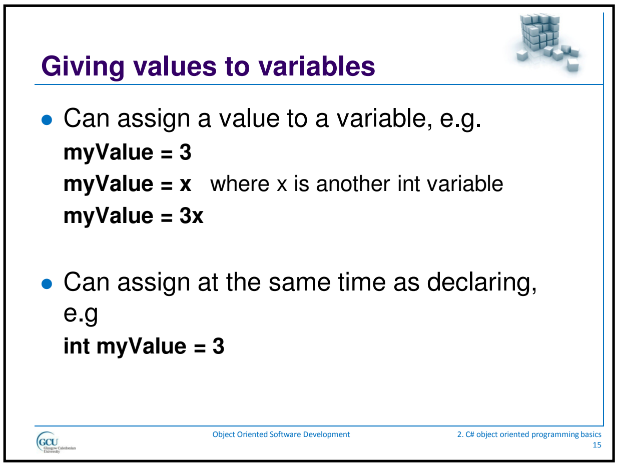


Declaring variables

- In C# a variable needs to be declared before it can be used
- Declaring a variable means specifying that a particular piece of information may exist by giving it a name and stating the **type**

int myValue

Declares a variable of type int with name "myValue"



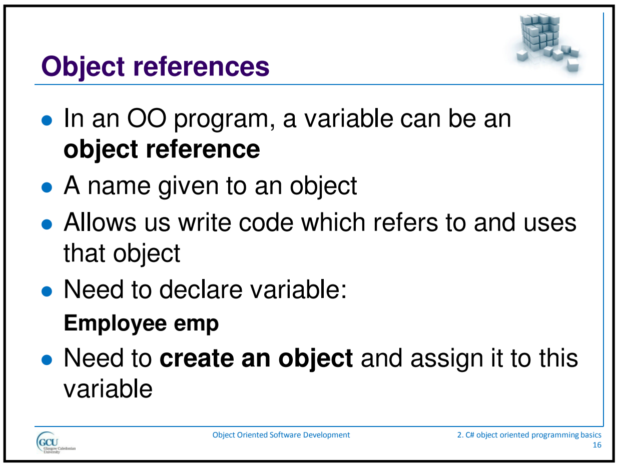
Giving values to variables

- Can assign a value to a variable, e.g.

myValue = 3
myValue = x where x is another int variable
myValue = 3x

- Can assign at the same time as declaring, e.g

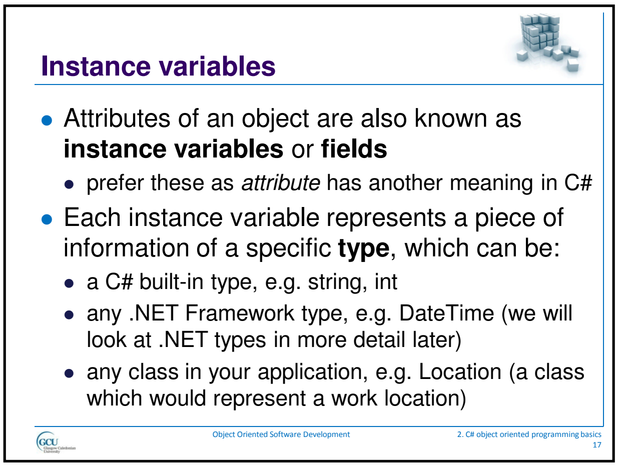
int myValue = 3



Object references

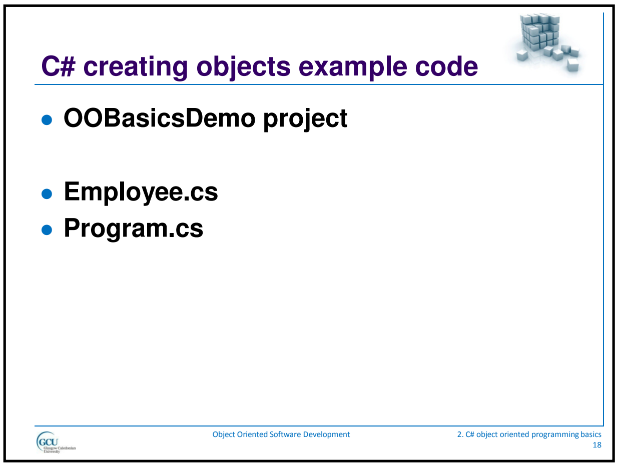
- In an OO program, a variable can be an **object reference**
- A name given to an object
- Allows us write code which refers to and uses that object
- Need to declare variable:

  Employee emp
- Need to **create an object** and assign it to this variable



Instance variables

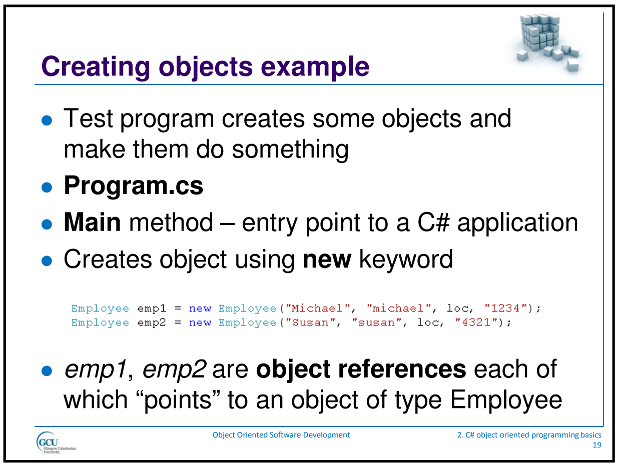
- Attributes of an object are also known as **instance variables** or **fields**
  - prefer these as *attribute* has another meaning in C#
- Each instance variable represents a piece of information of a specific **type**, which can be:
  - a C# built-in type, e.g. string, int
  - any .NET Framework type, e.g. DateTime (we will look at .NET types in more detail later)
  - any class in your application, e.g. Location (a class which would represent a work location)



C# creating objects example code

- **OOBasicsDemo project**

- **Employee.cs**
- **Program.cs**

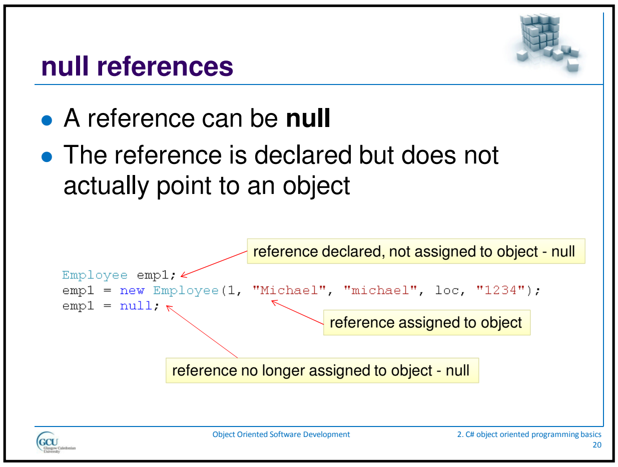


Creating objects example

- Test program creates some objects and make them do something
- **Program.cs**
- **Main** method – entry point to a C# application
- Creates object using **new** keyword

```
Employee emp1 = new Employee("Michael", "michael", loc, "1234");
Employee emp2 = new Employee("Susan", "susan", loc, "4321");
```

- *emp1*, *emp2* are **object references** each of which “points” to an object of type Employee



null references

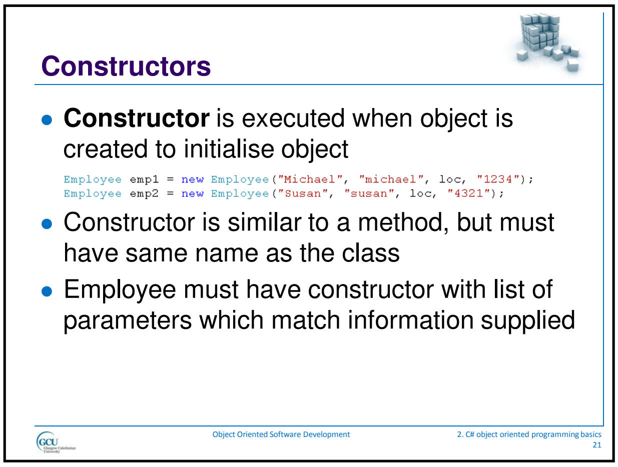
- A reference can be **null**
- The reference is declared but does not actually point to an object

```
Employee emp1;
emp1 = new Employee(1, "Michael", "michael", loc, "1234");
emp1 = null;
```

reference declared, not assigned to object - null

reference assigned to object

reference no longer assigned to object - null

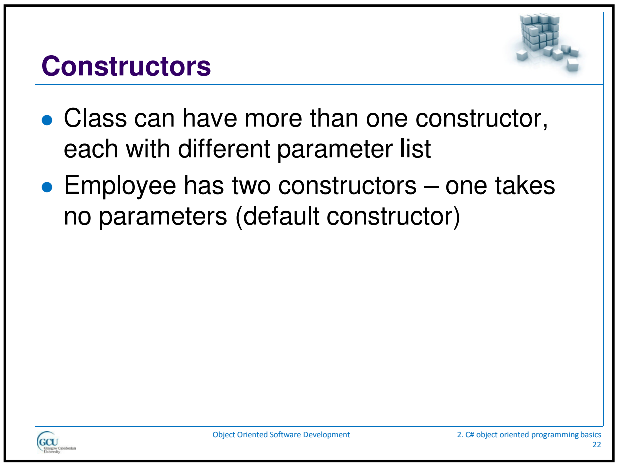


Constructors

- **Constructor** is executed when object is created to initialise object

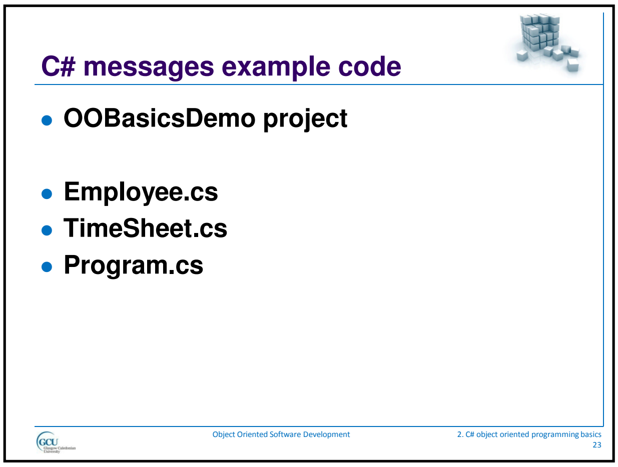
```
Employee emp1 = new Employee("Michael", "michael", loc, "1234");
Employee emp2 = new Employee("Susan", "susan", loc, "4321");
```

- Constructor is similar to a method, but must have same name as the class
- Employee must have constructor with list of parameters which match information supplied



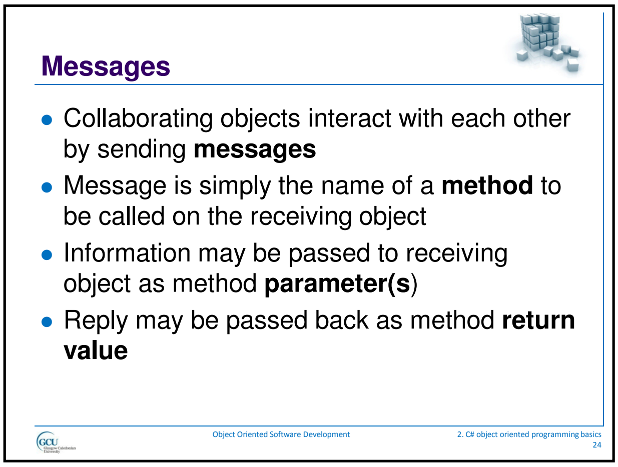
Constructors

- Class can have more than one constructor, each with different parameter list
- Employee has two constructors – one takes no parameters (default constructor)



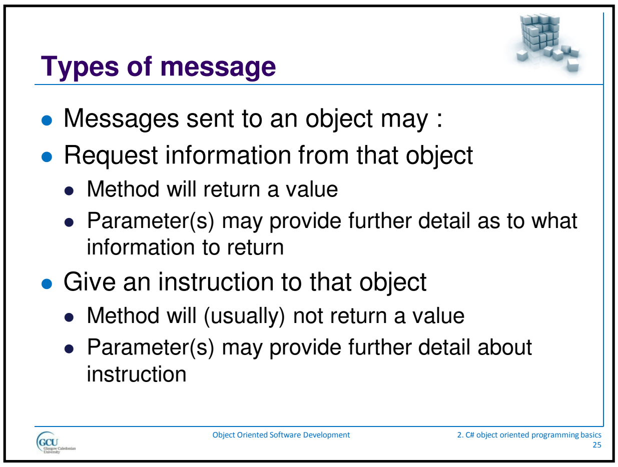
C# messages example code

- **OOBasicsDemo project**
- **Employee.cs**
- **TimeSheet.cs**
- **Program.cs**



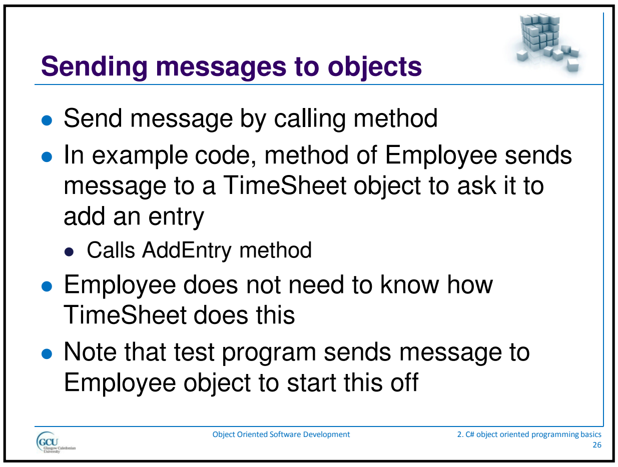
Messages

- Collaborating objects interact with each other by sending **messages**
- Message is simply the name of a **method** to be called on the receiving object
- Information may be passed to receiving object as method **parameter(s)**
- Reply may be passed back as method **return value**



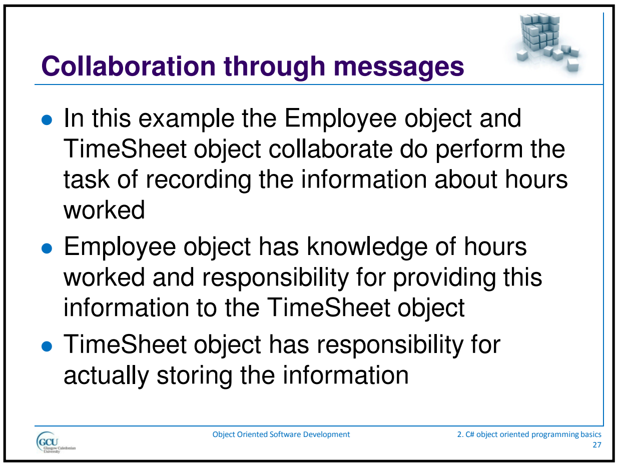
Types of message

- Messages sent to an object may :
- Request information from that object
  - Method will return a value
  - Parameter(s) may provide further detail as to what information to return
- Give an instruction to that object
  - Method will (usually) not return a value
  - Parameter(s) may provide further detail about instruction



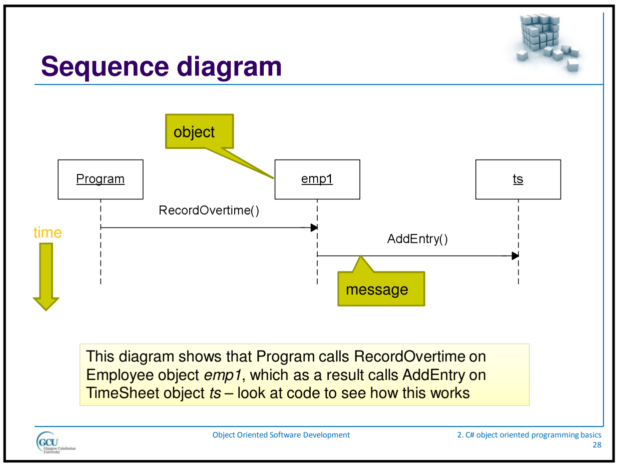
Sending messages to objects

- Send message by calling method
- In example code, method of Employee sends message to a TimeSheet object to ask it to add an entry
  - Calls AddEntry method
- Employee does not need to know how TimeSheet does this
- Note that test program sends message to Employee object to start this off



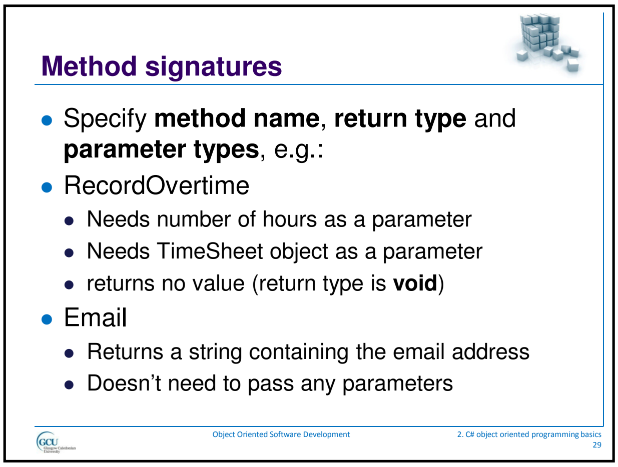
Collaboration through messages

- In this example the Employee object and TimeSheet object collaborate do perform the task of recording the information about hours worked
- Employee object has knowledge of hours worked and responsibility for providing this information to the TimeSheet object
- TimeSheet object has responsibility for actually storing the information



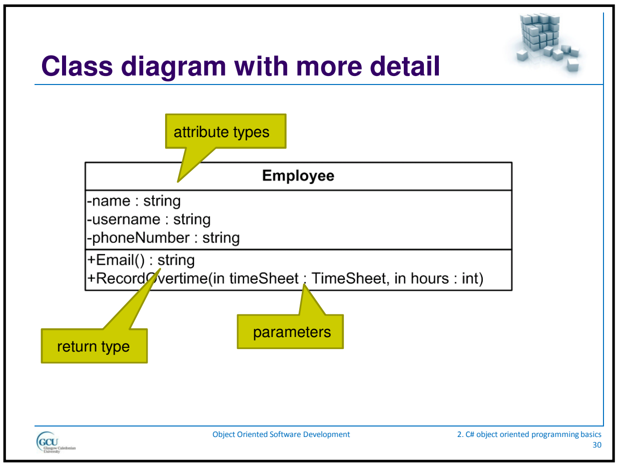
Sequence diagram

This diagram shows that Program calls RecordOvertime on Employee object *emp1*, which as a result calls AddEntry on TimeSheet object *ts* – look at code to see how this works



Method signatures

- Specify **method name**, **return type** and **parameter types**, e.g.:
- RecordOvertime
  - Needs number of hours as a parameter
  - Needs TimeSheet object as a parameter
  - returns no value (return type is **void**)
- Email
  - Returns a string containing the email address
  - Doesn't need to pass any parameters



Class diagram with more detail

Employee

-name : string
-username : string
-phoneNumber : string

+Email() : string
+RecordOvertime(in timeSheet : TimeSheet, in hours : int)

Methods and algorithms

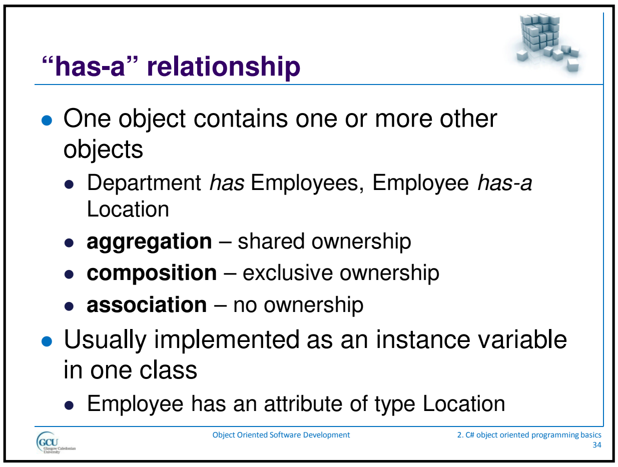
- A method contains code which defines the steps required to perform an action
- Sometimes this can be very simple:
  - AddEntry method simply writes to console
  - RecordOvertime method just sends a message to another object
- Sometimes the action is more complicated, and requires many steps to define an **algorithm**

Methods and algorithms

- Algorithm is a step-by-step procedure to solve a problem
- Steps are defined as program statements
- May involve combination of any of the following:
  - Calculations
  - Decisions: if-else statements
  - Repeated actions: for, while loops
  - Calls to other methods

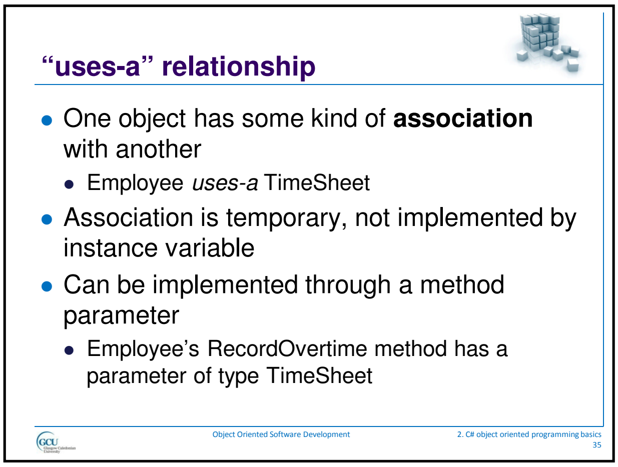
Relationships

- Need to define relationships to allow objects to interact
- Can show relationships in class and object diagrams
- Need to implement these in code
- Learn code patterns for implementing various relationship types



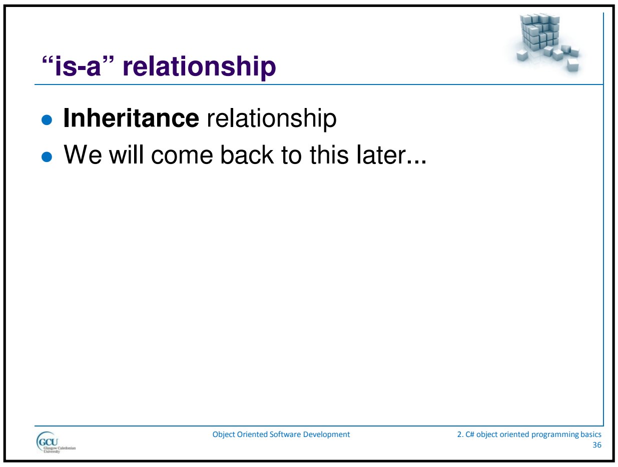
“has-a” relationship

- One object contains one or more other objects
  - Department *has* Employees, Employee *has-a* Location
  - **aggregation** – shared ownership
  - **composition** – exclusive ownership
  - **association** – no ownership
- Usually implemented as an instance variable in one class
  - Employee has an attribute of type Location



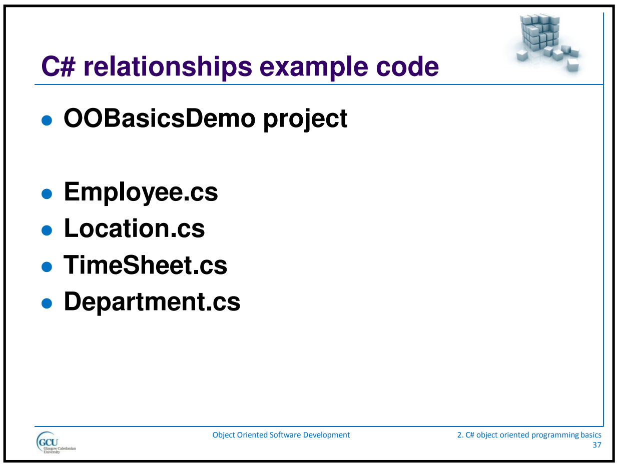
“uses-a” relationship

- One object has some kind of **association** with another
  - Employee *uses-a* TimeSheet
- Association is temporary, not implemented by instance variable
- Can be implemented through a method parameter
  - Employee’s RecordOvertime method has a parameter of type TimeSheet



“is-a” relationship

- **Inheritance** relationship
- We will come back to this later...



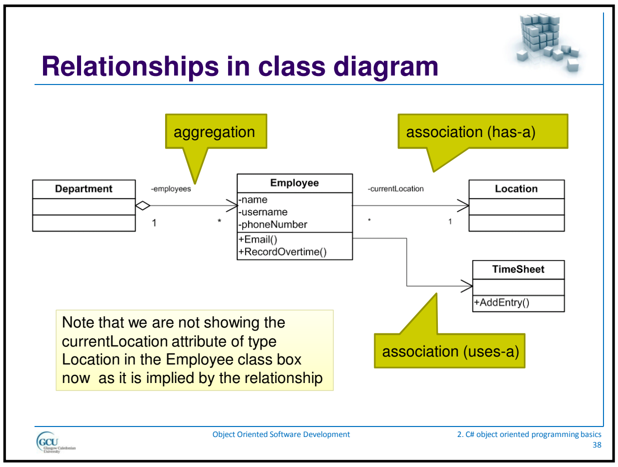
C# relationships example code

- OOBasicsDemo project

- Employee.cs
- Location.cs
- TimeSheet.cs
- Department.cs

Object Oriented Software Development

2. C# object oriented programming basics 37



Relationships in class diagram

aggregation

association (has-a)

Department

-employees

1

*

Employee

-name
-username
-phoneNumber

+Email()
+RecordOvertime()

-currentLocation

*

1

Location

TimeSheet

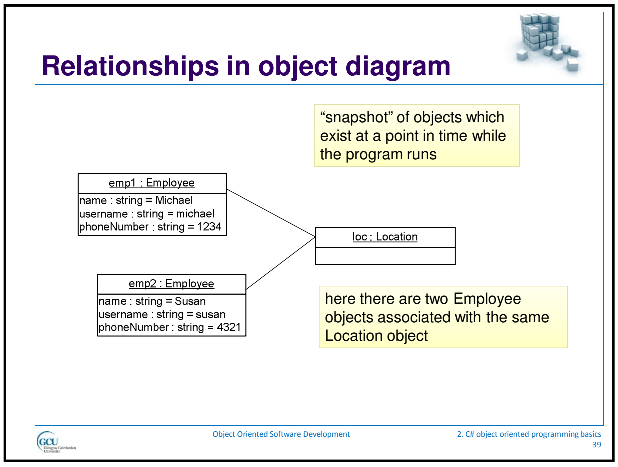
+AddEntry()

Note that we are not showing the currentLocation attribute of type Location in the Employee class box now as it is implied by the relationship

association (uses-a)

Object Oriented Software Development

2. C# object oriented programming basics 38



Relationships in object diagram

"snapshot" of objects which exist at a point in time while the program runs

emp1 : Employee

name : string = Michael
username : string = michael
phoneNumber : string = 1234

loc : Location

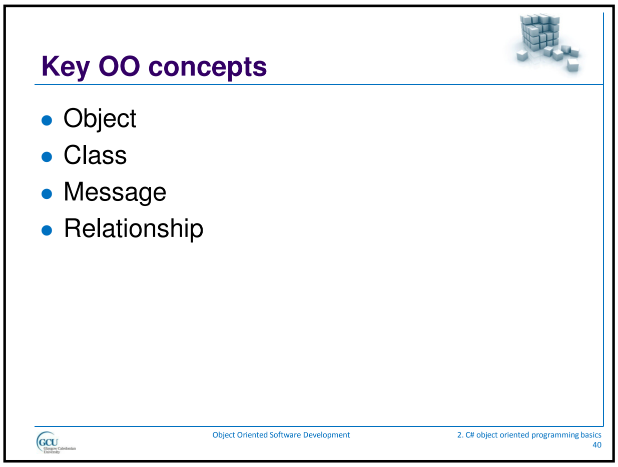
emp2 : Employee

name : string = Susan
username : string = susan
phoneNumber : string = 4321

here there are two Employee objects associated with the same Location object

Object Oriented Software Development

2. C# object oriented programming basics 39

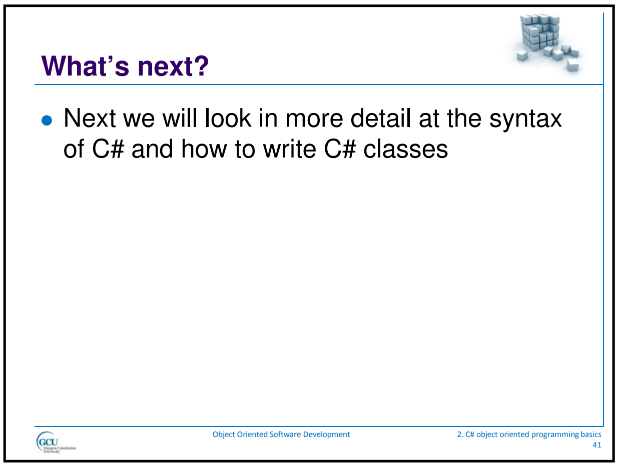


Key OO concepts

- Object
- Class
- Message
- Relationship

Object Oriented Software Development

2. C# object oriented programming basics 40

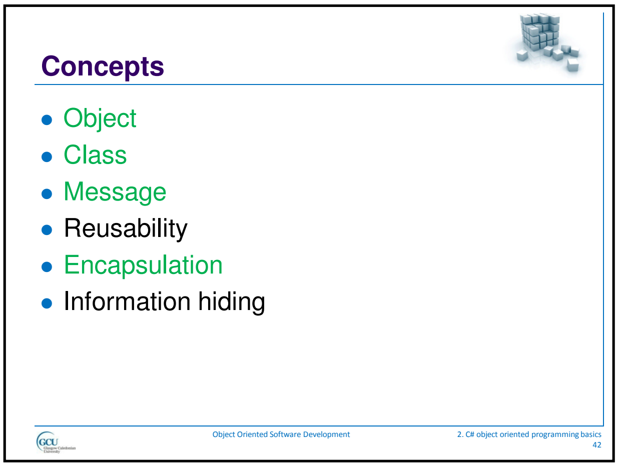


What’s next?

- Next we will look in more detail at the syntax of C# and how to write C# classes

Object Oriented Software Development

2. C# object oriented programming basics 41



Concepts

- Object
- Class
- Message
- Reusability
- Encapsulation
- Information hiding

Object Oriented Software Development

2. C# object oriented programming basics 42