

# G52OBJ answers

## Question 2

a)

- Objects are represented as blocks of memory,
- Pointer to the base of each object is stored as the object reference when object created
- Each variable is at a fixed position or offset within the object's memory
- This pointer is passed to the method when it is called
- Method can then access the variable by adding the offset to the pointer to find the location of the member variable

b)

- The memory layout of class Foo is arranged such that the first part of the object has an identical structure to the layout of an object of class Bar.
- This means that any member variable of Bar has the same offset from the base in an object of class Foo as it does in an object of class Bar.
- Therefore when one of Bar's methods is called and passed the pointer to an object of type Foo, when it adds the offset to access a member variable it ends up at the correct location.
- So Bar's methods work correctly without needing any knowledge of the fact they are actually manipulating an object of class Foo.
- Any variables defined in Foo (such as `_fooInt`) are then laid out in memory in the second part of the object, so as not to clash with the member variables of type `_Bar`
- Methods defined in Foo can access member variables in both the Foo and Bar parts of the object since the offsets from the base of the object will be known at compile time.

c)

- Methods are called by the CPU jumping to the relevant piece of memory containing the code for that method
- This address of a method can not be known at compile time, since inheritance allows methods to be replaced.
- If the address was bound in at compile time, then the code would always call the implementation defined against the type of the pointer.
- OO systems get around this by finding the address to call at run-time.
- The addresses of all methods are stored in an array called a vtable.
- Each method's address is stored at a fixed position in the vtable, first the base-class methods, followed by the sub-classes
- A pointer to each vtable is stored in **every** object
- Only one vtable is needed per class, it can be shared between objects (since they all support the same methods)
- When a method is called, it looks up the address of the method in the vtable pointed to by the object and calls this address.
- If the sub-class redefines a method, then it puts the address of its implementation into the array at the same position as the original implementation would use. Therefore when the address is looked up, it finds this implementation rather than the base-class

## Question 3

*Note: marks will be awarded based on the suitability of the design given, an example design is given below. This is not to be taken as a canonical solution.*

a)

**Game** — represents the whole game, holds references to the players and the dealer

**Player** — Represents a player in the game, keeps track of the players state and has a reference to the **hand** of cards the player holds

**Dealer** — Sub-class of player that extends it to add the functionality needed by the dealer (e.g. dealing cards), has a **Deck** of cards

**Deck** — Queue of playing cards, with the ability to shuffle them

**Hand** — Collection of cards held by the player, can also work out the players 'score'

**Card** — represents a playing card

b)

See end of document

#### Question 4

a)

- The Abstract Factory Pattern is a method for allowing programs to create objects without knowing the exact concrete type of the object that is being created.
- Factory interface defined which has methods to create objects of different types.
- Client code calls methods on the factory interface which return the created objects as interfaces rather than pointers to an object of a concrete class.
- Different implementations of this 'factory' interface can then be provided which create different concrete implementations.

b)

- The Factory Method pattern also allows objects to be created without defining exactly which type of object is being created.
- A class is defined that uses an object that supports a particular interface
- Inside this class there is an abstract method which is used to create the object e.g.  
public abstract ObjInterface createObj();
- Methods inside the class call this abstract method to create instances
- Sub-classes of the class provide a concrete implementation of the abstract method to create a specific object

c)

```
public abstract class CarProductionLine
{
    private abstract ICar makeCar(String type);

    public void produceCar(String type)
    {
        ICar car = makeCar(type);

        car.allocateMaterials();
        car.buildChasis();
        car.buildBody();
        car.combineChasisAndBody();
        car.ship();
    }
}
```

```
}  
}
```

```
public class FrioProductionLine extends CarProductionLine  
{  
    private abstract ICar makeCar(String type)  
    {  
        ICar car;  
  
        if(type.equals("saloon"))  
        {  
            car = new FrioSaloonCar();  
        }  
        else if(type.equals("hatchback"))  
        {  
            car = new FrioHatchbackCar();  
        }  
        else if(type.equals("estate"))  
        {  
            car = new FrioEstateCar();  
        }  
    }  
}
```

```
public class McCoyProductionLine extends CarProductionLine  
{  
    private abstract ICar makeCar(String type)  
    {  
        ICar car;  
  
        if(type.equals("saloon"))  
        {  
            car = new McCoySaloonCar();  
        }  
        else if(type.equals("hatchback"))  
        {  
            car = new McCoyHatchbackCar();  
        }  
        else if(type.equals("estate"))  
        {  
            car = new McCoyEstateCar();  
        }  
    }  
}
```

```
public class CVEProductionLine extends CarProductionLine  
{  
    private abstract ICar makeCar(String type)  
    {  
        ICar car;
```

```
        if(type.equals("saloon"))
        {
            car = new CVESaloonCar();
        }
        else if(type.equals("hatchback"))
        {
            car = new CVEHatchbackCar();
        }
        else if(type.equals("estate"))
        {
            car = new CVEEstateCar();
        }
    }
}
```

Or code to this effect, marks awarded based on how well this is implemented. An alternative approach would use three factory methods, one for each type of car, but this would gain slightly less marks as it ties the production line to producing those three types of cars.

