# 8. CASE STUDY: CREATING AN OBJECT ORIENTED PROGRAM



## Creating a program

As you have learned, an object oriented program works by creating objects that model the important entities ("things" or concepts) that are relevant to the purpose of the program. Much of the work of the program is done by the objects as they collaborate with each other by sending messages that result in other objects performing the actions defined in their methods. You have learned how to create Java classes that define templates to allow objects to be created, and how to write code to create objects and call methods. You have also learned that in order to be executable, the program needs an entry point, the main method, which runs first when the user commands the program to execute. The main method creates one or more objects that are needed to get started. The program may also need a user interface that allows the user to interact with it while it is executing.

To create a program you need to:

- Find out exactly what the program has to do – the **requirements**
- Decide what types of objects are needed to meet those requirements
- Design and write code for the classes to allow those objects to be created

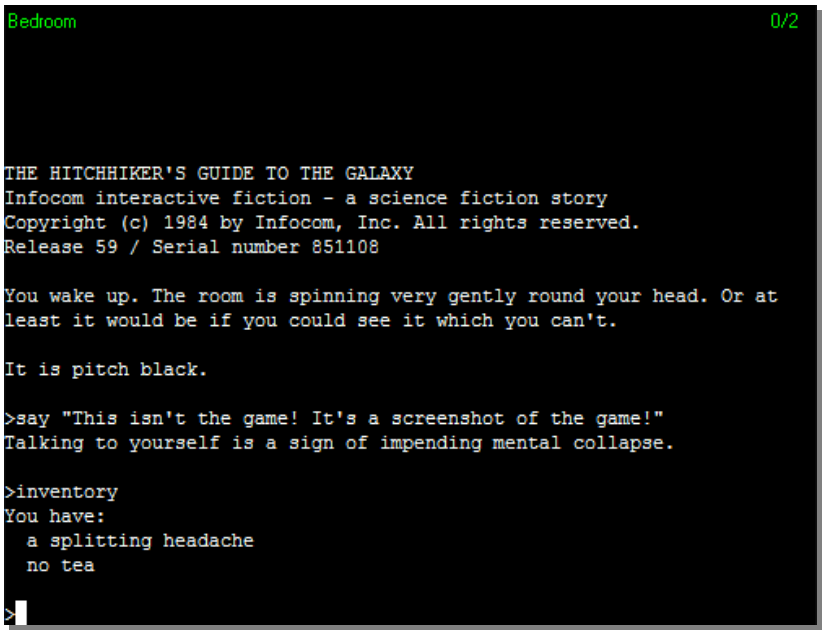
Sounds simple! Actually it isn't. It takes experience to do it well, and you will learn more about program design in this and other modules. In the next few lectures you will see an example of a program being developed.

This program will require more different kinds of objects than the simple examples you have seen so far. You will also see some new Java programming features and techniques that are needed to create the classes for the program. The example program is a simple game, but the way it is created would be similar for any kind of program.

## The GCU adventure game - requirements

The GCU adventure game is a text adventure game (sometimes known as interactive fiction). Text adventure games are a legacy from a time when computing power was small, when terminal access was commonplace, and when monochrome graphics was "state of the art". Players used imagination, fired by descriptions of old

abandoned caves, dungeons, and even futuristic spaceships populated by Vogons. For example, the screenshot shows the opening screen of the classic Hitchhiker's Guide to the Galaxy text adventure.




OK, our own game is really not very adventurous. In fact it's based loosely on the "World of Zuul" game in Barnes and Kölling's book *Objects First* (highly recommended as additional reading), which the authors describe as the "world's most boring adventure game". Ours is not any more exciting, but it will do fine for our purposes, though. It has a text-based user interface, which means we can create a fully-working interactive game without having to learn just yet about how to create graphical user interfaces.

The **game** centres around a character travelling through a world which consists of **rooms** within a university campus. Each room has a **description**. Rooms have **exits** which lead to other rooms. Each room can contain several **items** which may be used by a character when in that room. An item will also have a description. The **player** navigates through a world by typing **commands** (*go north*, *go south*, *quit*, and so on).

The main program requirements are as follows:

- The game world consisting of connected rooms should be created when the program starts
- The game should allow the player to take one turn after another
- The sequence of turns should repeat until a command is given to quit
- At each turn, the player can type a command to specify the action which he or she wants to take during that turn
- The player should be able to navigate from one room to another
- The player should be able to use any items which are located in a room
- The player should be able to ask for help during any turn

## Objects in the adventure game

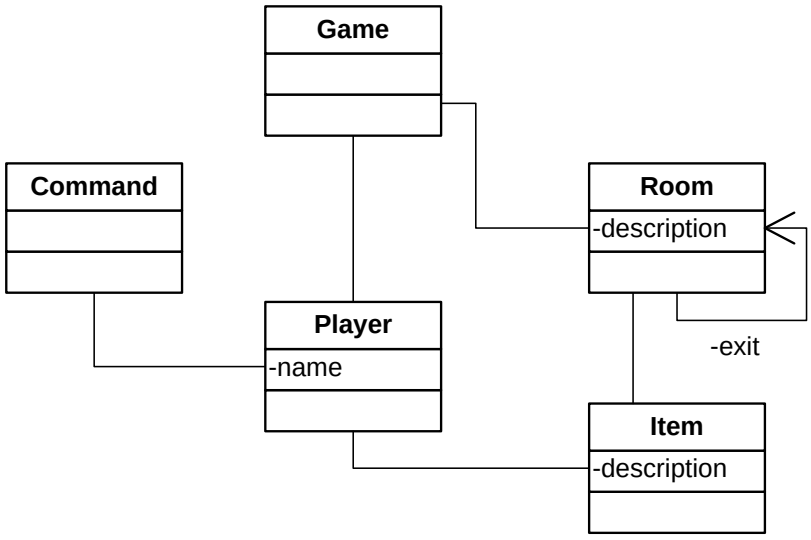
We can use the description and requirements above to make a first attempt at designing an object-oriented program that will implement the game. What kind of

objects will we need? The words highlighted in bold may help. These are all nouns, and name some 'things' that may be objects in the game. Let's look at these:

- **game** – this will be an object which sets up the game world and controls the game play. There will be **only one game** object.
- **room** – this will be an object which represents a room in the game world. There may be **many rooms**.
- **item** – this will be an object which represents an item in a room. There may be **several items** in each room.
- ***description*** – this simply describes a room or item, and should probably be a **property** rather than an object in its own right
- ***exit*** – if you think about it, an exit from a room is really just a way into another room, so an exit is actually a way of **referring to a room object**, not another type of object
- **player** – this will be an object which represents a player. A player will **be in one room** at any point in the game. There will be **only one player** in the game. The player should have a **name**, which will be a property of the object.
- **command** – it may not be immediately obvious that a command will be an object, but the job of representing the player's input may be quite complicated. For example, we need to check whether the words the player types are in fact a valid command. Command objects will be useful, and there will be **one object for each input** entered by the player.

## Modelling the adventure game

So let's look at a first attempt at a **model** for the adventure game. A class diagram is one kind of diagram that you can use to illustrate a model. Here is a class diagram that shows the classes which will be needed to create these objects.



The diagram shows Game, Room, Item, Player and Command classes, with properties as described above. At this stage we have not decided on the methods that each class will need, and there may be additional properties needed also.

Unlike the example you saw earlier, this class diagram has many classes. The **links**, or **associations**, between the classes in the class diagram show that **objects will interact** in some way. For example, a game object will interact with a player object. We'll look at these interactions in more detail as we go along.

Note that the Room class is linked to itself – this is because we noted above that a room's exit is actually a reference to another room object, so rooms can be linked with each other.

While it is not the only part of a model of a system, the class diagram is very important when you start to build the system as a Java program. The classes become the Java classes that you need to write.

## Incremental development

We now know what kind of objects are required, and so what classes need to be created. It would not be wise, however, to go ahead and write all the classes at once. Instead, we will build up the program in smaller stages. Each stage, or **increment**, will create a version of the program that runs, but implements only part of the list of requirements for the program.

The class or classes created in each increment should be tested to make sure they work as they should. Then, in the following increments we will implement more requirements by adding further classes and/or adding new capabilities to the classes that have already been created. At each stage we will take the class in the original class diagram above and think carefully about what properties and methods need to be included for that stage. We will also think about any associations between classes and how they need to be implemented.

## Increment 1: interaction between game and players

In the first increment we will meet only the following requirement:

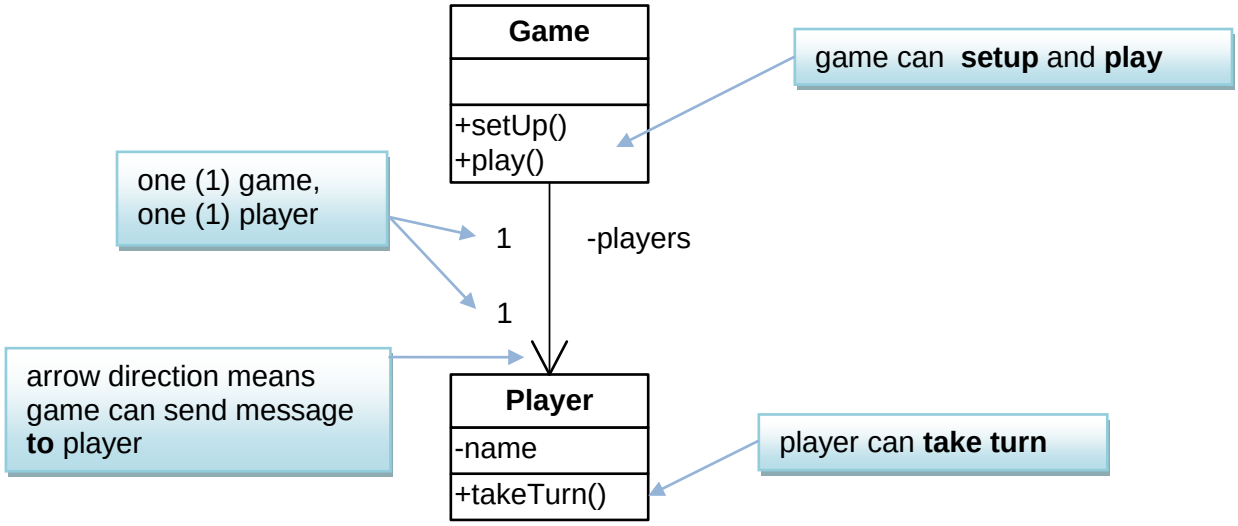
- *The game should allow the player to take a turn…*

What classes do we need to create for this? It sounds like we just need `Game` and `Player`, so we will start to create Java code to implement the `Game` and `Player` classes. It is important that we write the code so that game and player objects can interact the way we want them to.

We said that there will be only **one game** object, and that there will also be **one player**. The player will be part of the game. In fact, the game will create its player object as part of the set up. The game will then need to communicate with the player during the game play – it will **send a message** to to tell the player to take a turn.

Since the player is part of the game, we say there is a **"has-a"** relationship between `Game` and `Player`. We can also describe this as a **"whole-part"** relationship.

We can now add some more detail to the model for these classes. The figure shows the class diagram for these classes with some additional features based on the description above.




## The Player class in Java

The code for the player class is shown below. Note that there is one field, `name`, which is accessed publicly through getter and setter methods.

```
/**
 * Class Player
 *
 * Represents a player in the game
 *
 * @version 1.0
 */
public class Player
{

    // the player's name
    private String name;

    /**
     * Constructor for objects of class Player
     */
    public Player(String name)
    {
        this.name = name;
    }
```

```
    /**
     * the player takes a turn in the game
     */
    public void takeTurn()
    {
        System.out.println("Player " + name +
             " taking turn...");
    }
    /**
     * returns the name of this player
     */
    public String getName()
    {
        return name;
    }

    /**
     * set a new name for this player
     */
    public void setName(String name)
    {
        this.name = name;
    }
}
```

Note that this will **not** be the final version of the Player class. For now, it has just enough functionality to meet the requirement for this increment.

### Testing the Player class

It is important to test your code during each increment. You can test the (partially) completed program. You should also test each class individually to ensure that it can perform its actions correctly – BlueJ makes it easy to test each class.

You can try creating the `Player` class for yourself. Open the *adventure* project which you can download from GCULearn. Add a new class called `Player`. Open the `Player` class for editing and replace the existing code with the code above, and compile the class.

Create a new `Player` object in the BlueJ object bench. Choose any name you want. Test the class as follows:

1. Select Inspect from the object's right-click menu and check that the *name* field has the value you chose
2. Run the `getName` method and check that it returns the value of the *name* field
3. Run the `setName` method and enter a new value for the *name* field, then Inspect the object again to check that the value has been changed
4. Run the `takeTurn` method and observe the result

The `takeTurn` method does not do anything useful yet. All it does is print out a message which shows that the method has been called. We will complete this method later.

# Implementing the interaction

If you look through the code for the `Player` class, there is no mention of the `Game` class. That is because a `Player` object does not need to communicate with the `Game` – it simply takes a turn when it is told.

However, the `Game` does need to communicate with its players, to tell them to take their turns. The `Game` class will therefore need a reference to `Player` to allow it to send a message.

> *KEY POINT*
>
> *Sending a message to an object is done by calling a method of that object. The Game tells a Player to take its turn by calling the takeTurn method of the Player object*

## Using a code pattern

We now have to solve the problem of how we allow `Game` to communicate with `Player`?

It is very common for classes to be related with a "has-a" relationship like this. Common problems often have **common solutions** that are based on the experience of people who have solved the problems successfully in the past.

In programming, these solutions are known as **patterns**. The pattern we will use here can be described like this:

**CODE PATTERN:** ***"HAS-A"***

**Problem:** how do you implement a "has-a" relationship, where a "whole" object needs to send messages to its "part" objects?

**Solution:** the class that needs to send the message has a field whose type is the name of the other class.

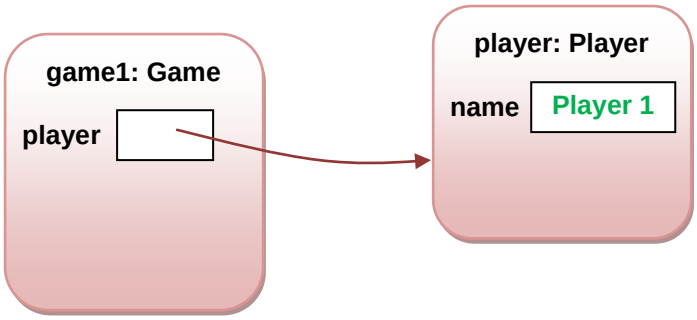
So the `Game` class should have a **field** declared like this:

```
public class Game {

    // the player
    private Player player;
    ...
```

The field, or instance variable, `player` is a **reference to an object of type** `Player`**.**

We can represent this in an **object diagram**:




The setup code in `Game` should then be like this.

```
private void setUp()
{
    player = new Player();
}
```

Within the play method of `Game`, a message should then be sent by calling the `takeTurn` method of the `player` field. Note the method call syntax, which includes the brackets (empty as there are no parameters for this method).

```
public void play()
{
    ...
    player.takeTurn();
    ...
}
```

The `Player` object will then respond by running its `takeTurn` method.

## The game loop

Game play usually involves a **game loop**, which repeats until a signal is given to end the game. In our `Game` class, the game loop will be part of the `play` method. We can't do very much in the game yet, so we will implement a minimal game loop which simply tells the player to take a turn and then exits.

```
public void play()
{
    // Enter the main command loop.
    // Here we will repeatedly read commands and execute
    // them until the game is over.

    boolean finished = true;   // put this in temporarily
                               // so game finishes
                               // after one turn
    do
    {
         System.out.println("Player:" + player.getName());
         player.takeTurn();
    } while (! finished);
```

```
        System.out.println(
              "Thank you for playing.  Good bye.");
    }
```

This is a do-while loop, so the code inside it will run at least once. The boolean variable *finished* is a **flag** – setting the value of the flag to **true** will cause the game to finish. In the final, playable version of the game, the flag will be set when a player enters a command to quit. However, we are not ready to handle commands yet, so we will set this to true immediately, so that the game loop will terminate after one complete turn.

## The Game class in Java

The complete code for the initial version of the `Game` class is shown below. There are a few things to note here:

- The `setup` method is called in the `Game` constructor, which makes sense as instantiating a new game object should include setting up the game world.
- The `setup` method is declared **private** as it will not be called by any other object.
- The code in the setup method creates the `Player` object after reading in the player's name from the terminal. This is done with the help of a Java library class called `Scanner` – we will look at this again in a later lecture
- There is another private method, `printWelcome`, which is used by the `play` method.
- The only **public** method, then, is `play`. That means that the only thing that can be done with a `Game` object is to tell it to play.

```
import java.util.Scanner;

/**
 * Class Game
 *
 * Sets up and controls the game

 * @version 1.0
 */

public class Game
{
    // the players in the game
    private Player player;

    /**
     * Create the game and initialise it
     */
    public Game()
    {
        setUp();
    }
```

```
    /**
     * Initialize the game world
     */
    private void setUp()
    {
        printWelcome();
        System.out.print("Please enter your name\n>");
        Scanner in = new Scanner(System.in);
        String playerName = in.nextLine();
        player = new Player(playerName);
    }

    /**
     *  Main play routine.  Loops until end of play.
     */
    public void play()
    {
        // Enter the main command loop.
        // Here we will repeatedly read commands and execute
        // them until the game is over.

        boolean finished = true;   // put this in temporarily
                                   // so game finishes
                                   // after one turn
        do
        {
             System.out.println("Player:" + player.getName());
             player.takeTurn();
        } while (! finished);

        System.out.println(
              "Thank you for playing.  Good bye.");
    }

    /**
     * Print out the opening message
     */
    private void printWelcome()
    {
        String welcome = "\n";
        welcome += "Welcome to the World of GCU!" + "\n";
        welcome += "World of GCU is a new, incredibly boring
            adventure game." + "\n";
        welcome += "\n";
        System.out.print(welcome);
    }

   /**
    * Create and play a game.
    */
    public static void main(String[] args)
    {
        Game game = new Game();
        game.play();
    }
}
```

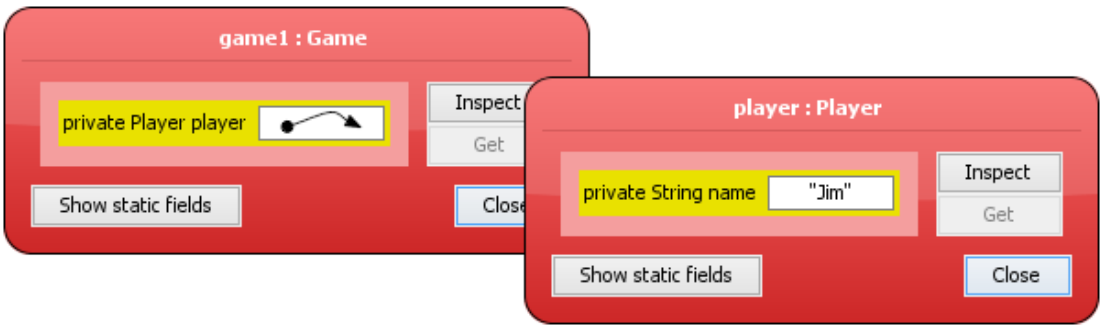
## Testing the Game class

You can try creating this class for yourself using the *adventure* project. Open the project, which should contain `Player` and `Game` classes, and compile the classes. Test the `Game` class as follows:

1. Create a new `Game` object in the BlueJ object bench, accepting the default name for the instance. As the setup method requires the input of a player name the terminal window should open and you should be prompted to enter a name. Once you have done so a `Game` object should appear in the object bench.

2. Select Inspect from the object's right-click menu and check that there is a field that is a reference of type `Player`.


3. Click the Inspect button beside the `player` field and check that the field refers to a `Player` object with a `name` field containing the value you entered when creating the `Game` object. This shows that the `Game` object has correctly created a `Player` object.




4. Call the `play` method of the Game object. Check that the welcome message is printed, and that the `takeTurn` method is called for the `Player` object. Look at the lines which appear in the terminal window – which method, in which class, do you think caused each one of these lines to be printed?

These tests show that a `Game` object does what we want it to. Finally, we can actually run the program using the `main` method.

5. Right-click the `Game` **class in the BlueJ class diagram**, and select `main`. Remember that `main` is a static method, so you call it using the class, not using an object. Click OK in the Method Call box. Remember that the main method simply creates a `Game` object and calls its `play` method. Check that you see the output from the `play` method.

# Wrap up

You've been introduced in this lecture to the following concepts:

***Modelling, Incremental development, "has-a" relationship, Code patterns***

In the next lecture you will see the next increment of the game. We will add some rooms to the game and put some items in the rooms for the players to use.