2. Creating classes: Game and Player

Programming techniques in this chapter:
Arrays, static variables and methods, constants, do-while loop, main method

Class associations in this chapter:
"has-a" relationships – one-to-one and one-to-many

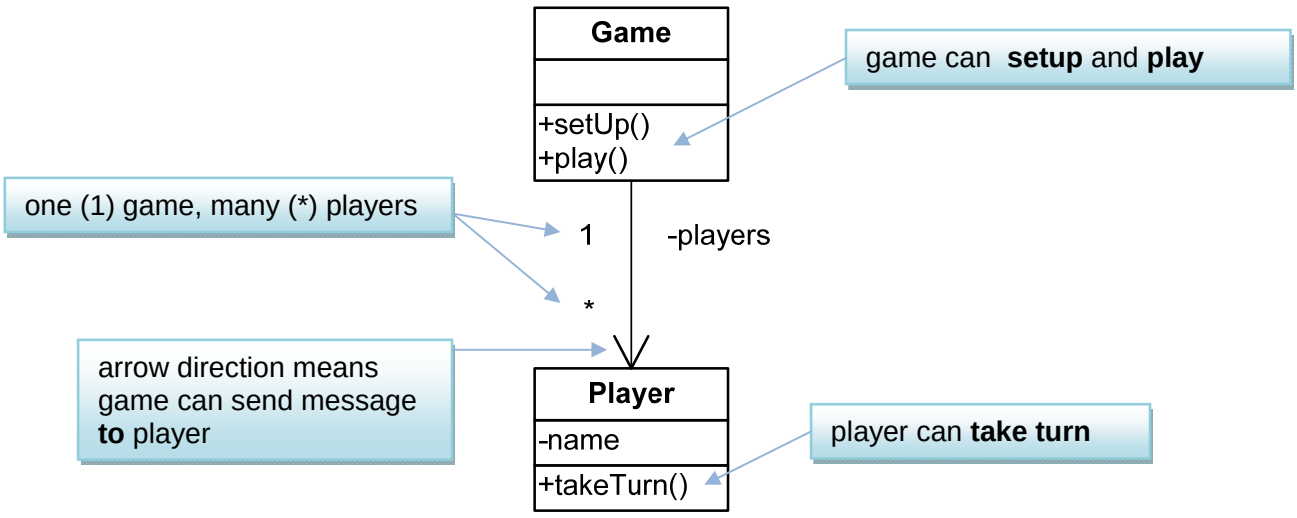
Interaction between game and players

In this chapter 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. As we do so, we will need some new programming techniques to help us.

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

Since the players are 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;
    }
}
```

Testing the Player class

You can try creating this class for yourself. Create a new BlueJ project called *adventure*. 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 a problem - how do 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** which 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"***

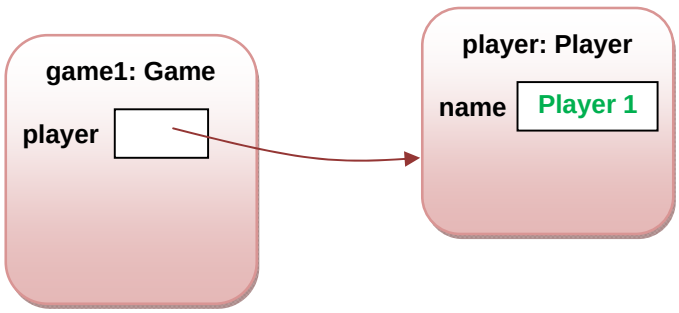
Problem: how do you implement a "has-a" relationship, where a "whole" object needs to send messages to its "part" objects?

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

So the `Game` class could 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` would then be like this.

```
    private void setUp()
    {
        player = new Player("Player 1");
    }
```

Within the play method of `Game`, a message would 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 object which is referred to will then respond by running its `takeTurn` method.

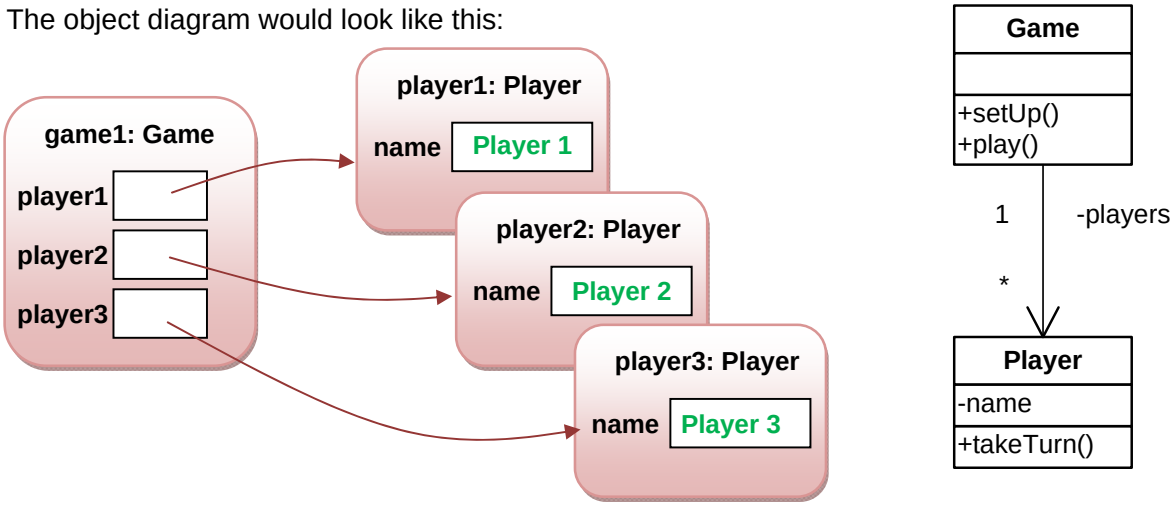
Adding more players

The `Game` class now has a reference to one `Player`. However, we actually need to be able to have more than one player. How do we deal with this?

One way would be to have more than one field for players, for example:

```
public class Game {

    // the player
    private Player player1;
    private Player player2;
    private Player player3;
    ...


    private void setUp()
    {
        player1 = new Player("Player 1");
        player2 = new Player("Player 2");
        player3 = new Player("Player 3");
    }
```

The object diagram would look like this:



Note that the class diagram, shown again here for comparison, illustrates the **types of object** which can be created. The object diagram shows **actual objects** which are in existence at a point in time while the program is running.

The `play` method would also need to be modified:

```
    public void play()
    {
        ...
        player1.takeTurn();
        player2.takeTurn();
        player3.takeTurn();
        ...
    }
```

We can now have three players in the game. What if we need to add more? We would have to add more fields, and change the setup code and the play code. There has to be a better way. There is. We can use an **array**.

Arrays

In Java, as in most programming languages, an array is a structure that holds **multiple values of the same type**. A Java array is also an object.

An array can contain primitive data type values. As it is an object, an array must be declared and instantiated. The size of the array is specified when it is instantiated. For example

```
int[] anArray;
anArray = new int[10];
```

An array can also be created using a shortcut. For example:

```
int[] anArray = {1,2,3,4,5,6,7,8,9,10}
```

An array element can be accessed using an index value. For example:

```
int i = anArray[5]
```

Note that **array indexes start from 0**. The value of anArray[0] is 1, while anArray[5] is actually the 6th element, with value 6 in this example.

The size of an array can be found using the length attribute. For example:

```
int len = anArray.length
```

Arrays of objects

Arrays can also hold objects of any type. Actually, the array doesn't hold the objects themselves – it holds **references** to objects. It is important to realise that when creating an array of objects, the array itself must be declared and instantiated, and that each individual element in the array must also be instantiated. This code creates an array of *Integer* objects.

```
Integer[] anArray = new Integer[5];      // create the array (does not create
                                          // the objects in the array)

for (int i=0;i<anArray.length;i++){
       anArray[i] = new Integer(i);       // create the objects in the array
       System.out.println(anArray[i].toString());
}
```

Note that you must create the array first, and then create the objects in the array. If you miss out the second of these steps, you will probably get errors when you run the code as the array will simply contain **null** references – the elements in the array do not have any objects to point to as no objects have been created.

> **NOTE**
>
> Java has a set of primitive data types, such as `int`. These have object equivalents, such as `Integer`. An `Integer` object holds an `int` value but provides additional functionality, for example for converting the value to other data types.

Implementing the Game class with an array

Using an array is a common solution when a "has-a" relationship is also a **one-to-many** relationship. The pattern we are using now is slightly different to before:

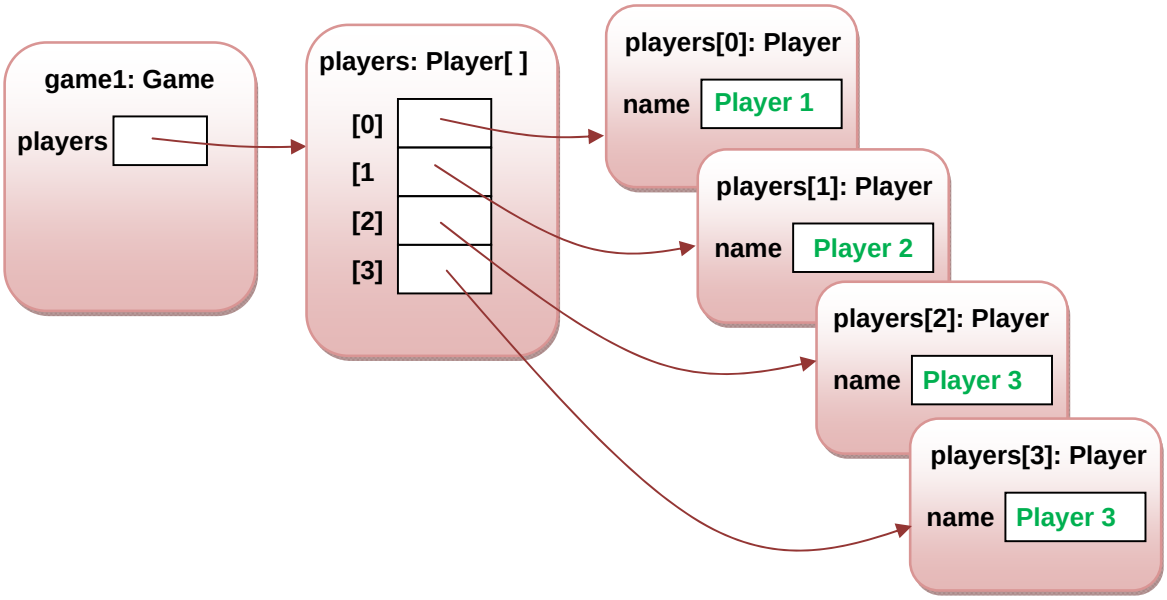
CODE PATTERN: ***"HAS-A(ARRAY)"***

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

Solution: the class which needs to send the message has a field which is an array of objects whose type is the name of the other class.

```
public class Game {

    // the players in the game
    private Player[] players;
    // the number of players in the game
    private static final int NUM_PLAYERS = 4;

    ...

    private void setUp()
    {
        players = new Player[NUM_PLAYERS];
        for(int i=0; i<NUM_PLAYERS; i++)
        {
            players[i] = new Player(String.format("Player %d", i+1)
        }
    }

    public void play()
    {
        ...
        for(int i=0;i<NUM_PLAYERS;i++)
        {
            players[i].takeTurn();
        }
        ...
    }
```

The object diagram now looks like this:



Note that there are four players here. What would we have to do to add, say, two more players to the game? We would simply have to change the value of `NUM_PLAYERS` to 6. We would not have to add more fields or change the play method at all.

Static, final and constants

Look at this line of code from the `Game` class which defines the value of `NUM_PLAYERS`. `NUM_PLAYERS` is a **constant**, and this is how constants are typically defined:

```
    // the number of players in the game
    private static final int NUM_PLAYERS = 4;
```

There are two key words here which may be unfamiliar to you – **static** and **final**. What do these mean, and why are they used here?

Class and instance members

When you declare a field in a class, like this example,

```
public class MyClass {
    public int instanceVar;
```

you declare an **instance variable**. Every time you create an instance of a class, the runtime system creates one copy of each the class's instance variables for the instance. To use the value of the variable you need to have an instance of the class, i.e. an object.

```
MyClass anObject = new MyClass();
int i = anObject.instanceVar;
```

If a variable is declared with the **static** keyword, it is a **class variable**. There is one copy of each class variable **shared between all instances** of the class.

```
    public static int classVar;
```

To use the value of the class variable, you do not need an instance – you simply use the class name:

```
int i = MyClass.classVar;
```

Methods are similar. Your classes can have **instance methods** and **class methods**. Instance methods operate on the current object's instance variables but also have access to the class variables.

```
    public void instanceMethod()
    {
        instanceVar = instanceVar * 2;    // do some action using instance var
    }
```

Class methods, on the other hand, cannot access the instance variables declared within the class (unless they create a new object and access them through the object). Class methods can access class variables.

```
    public static void classMethod()
    {
        classVar = classVar * 2;          // do some action using class var
    }
```

To use a class method, you do not need an instance – you simply use the class name:

```
MyClass.classMethod();
```

Class methods provide a way of providing specific functionality **without the need to create an object**.

Note that you can, although there is usually no point in doing so, access class variables and methods through an instance:

Constants

A **constant** is a value which will **never change** while the program is running. To create a named constant in Java you use the **final** type modifier. A field declared final cannot be changed in the program.

```
public class CircleStuff {
    public final float PI = 3.1416;
    ...
}
```

You can group your constants in their own class, like this:

```
public class Suit {
    public static final int CLUBS = 0;
    public static final int DIAMONDS = 1;
    public static final int HEARTS = 2;
    public static final int SPADES = 3;
}
```

You access these constants with the dot notation.

```
float pi = CircleStuff.PI
int suit = Suit.HEARTS
```

It is customary to make constant names all capitals.

The **static** modifier makes these constants **class constants**. They belong to their classes, not to the objects derived from these classes. This means that there is only one copy of the constant in memory. Without the static modifier, each object derived from the class would have its own copy of the constant.

NOTE

The Math library class contains some useful mathematical constants, including PI, so if you need to use a value for π you can simply use `Math.PI`

The Math class also has many useful class (static) methods for mathematical expressions, for example the `sin` method which calculates the sine of an angle. To call this method you use the class name, like this:

```
double result = Math.sin(x)
```

Static and final – a summary

Don't confuse the effect of the key words static and final!

- **static**
  - Class variable (or method)
  - Shared by all instances of a class
- **final**
  - Constant value
  - Can't be changed after it is assigned

We often use the combination **static final** to define a **class constant.**

Loops: do-while

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 each player to take a turn and then exits.

```
    public void play()
    {
        printWelcome();

        // 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
        {
            for(int i=0;i<NUM_PLAYERS;i++)
            {
                System.out.println("Player: " + players[i].getName());
                players[i].takeTurn();
            }
        } while (! finished);

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

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.

You have seen **while** loops before. Note that the loop in this method is a variation on this, a **do-while** loop.

```
do {
     statement(s)
} while (expression);
```

The difference between **do-while** and **while** is that **do-while** evaluates its expression at the bottom of the loop instead of the top. Therefore, the statements within the **do** block are always executed at least once. In this case, although the flag was set before the loop, the loop still runs once. What do you think would happen if we used a while loop here instead?

The main method

A Java program works by creating objects and making them interact. However, in order to get started there must be an **entry point** to the program, which starts the program running. This is called the **main method**.

The main method is a static method which must have a **signature** like this (the signature of a method consists of its name, its return type and its list of parameters):

```
public static void main(String [] args)
{
    ...
}
```

The code in the main method usually creates one or more objects and calls methods of the objects to get things going. In the game program, the main method will create a `Game` object, and call its `play` method. That `Game` object will create `Player` objects and interact with them to perform the game play.

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

Where does the main method go?

A program should only have one main method, and the code for `main` **can be placed in any class**, although it does not actually represent any behaviour of a particular object. We will put it in the `Game` class for convenience. Alternatively, we could have created a separate class, called something like `GameRunner`, just to hold the `main` method program.

> **NOTE**
>
> Standalone Java applications require a main method as an entry point. There are other types of Java programs, such as **servlets** (which run on web servers) and **applets** (which run inside web browsers) which have different entry points. Some Java projects are **class libraries**, designed to be used in other applications, which have no entry point at all.

As a general rule, the main method should not have much code in it. It is really there to create the objects required to get the program going. In this example, it simply creates a `Game` object and passes responsibility for playing the game to that object. In a program with a graphical user interface, the main method would simply create and display the frame which represents the main window of the application.

Some textbooks show code examples which consist of a class which only has a main method, and all the code is in the main method. This is not good practice for real programs!

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.
- 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.

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

 * @version 1.0
 */

public class Game
{
    // the players in the game
    private Player[] players;
```

```
    // the maximum number of players allowed
    private static final int NUM_PLAYERS = 4;

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

    /**
     * Initialize the game world
     */
    private void setUp()
    {
        players = new Player[NUM_PLAYERS];
        for(int i=0;i<NUM_PLAYERS;i++)
        {
            players[i] = new Player("Player " + Integer.toString(i+1));
        }
    }
```

```
    /**
     *  Main play routine.  Loops until end of play.
     */
    public void play()
    {
        printWelcome();

        // Enter the main command loop.  Here we 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
        {
            for(int i=0;i<NUM_PLAYERS;i++)
            {
                System.out.println("Player: " + players[i].getName());
                players[i].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();
    }
}
```

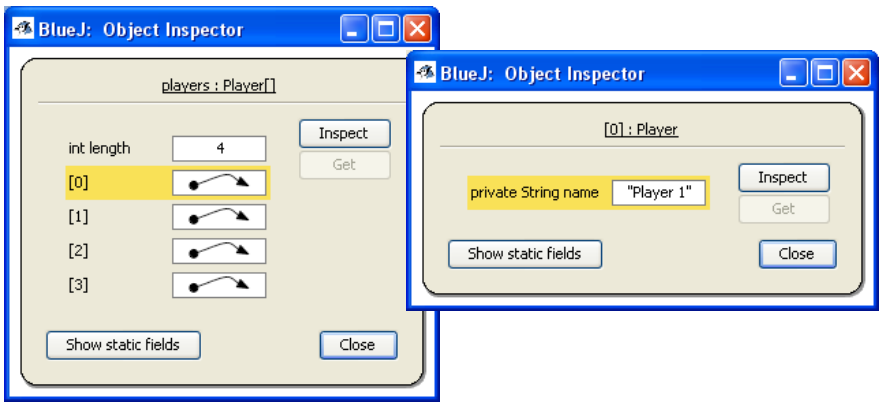
Testing the Game class

You can try creating this class for yourself. Open your *adventure* project. Add a new class called `Game`. Open the `Game` class for editing and replace the existing code with the code above, and compile the class. You should now have `Player` and `Game` classes both of which should be compiled.

Create a new `Game` 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 there is an array of `Players`.

2. Click the Inspect button beside players and check that there are four `Player` objects in the array. You can inspect each `Player` object in turn.



3. Run the `play` method. Check that the welcome message is printed, and that the `takeTurn` method is called for each `Player` object.

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

4. 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.

What's next?

We now can create game and player objects and make them interact, and we have a basic program which uses these objects. In the next chapter we will add some rooms to the game and put some items in the rooms for the players to use.