5. Completing the program

Programming techniques in this chapter:
Inheritance, polymorphism, casting, switch statement, command prompt tools

Class associations in this chapter:
"is-a" relationships, "uses-a" relationships, "creates-a" relationships

Different kinds of items

At the moment there is only one kind of item in the game. It would be nice to have the possibility of different kinds of items which would behave differently when used. All kinds items might actually have some common behaviour, but some kinds may do some things differently, or have their own special extra behaviour.

We will add a new kind of item to the game. This will be called a `BonusItem`, and its extra feature is that it can reveal a secret bonus keyword.

This situation is an example of a new code pattern, the **"is-a" pattern.**

CODE PATTERN: ***"IS-A"***

Problem: how do you implement a relationship where one class is a specialized version of another more general class and shares some of its behaviour

Solution: the specialised class extends the more general class and adds new methods or overrides methods of the general class

This pattern is usually called **inheritance**.

Inheritance

Defining a new class to create a new type can involve a lot of effort. Sometimes a class already exists that is close to what you need. You can **extend** that class to produce a new class that is exactly what you need. In many cases, this will require much less effort than that required to start from scratch and define a new class.

You can extend your own classes, or you can extend classes which have been written by others and which you have access to (for example the Java API classes).

When you extend a class, the new class is called the **subclass** and the class that was extended is called the **superclass**.

To extend another class you use the `extends` keyword in your new class declaration:

```
public class MyNewClass extends MyOtherClass {
```

What is inherited?

The subclass inherits all of the variables and all of the methods defined the superclass, as if you had completely defined the new class from scratch, and had reproduced all of the code already defined in the existing superclass.

Therefore, inheritance often makes it possible to define a new class with a minimum requirement to write new code by reusing the code that was previously written in superclasses.

The behaviour of the methods defined in a superclass and inherited into your new class may or may not be appropriate for an object instantiated from your new class. If those methods are appropriate, you can simply leave them alone.

Overriding

If the behaviour of one or more methods defined in a superclass and inherited into your new class is **not** appropriate for an object of your new class, you can change that behaviour by **overriding** the method in your new class.

To override a method in your new class, define a method in your new class with the same name and signature (i.e. parameter list, and return type) as the original. Then provide a body for the new method. Write code in that body to cause the behaviour of the overridden method to be appropriate for an object of your new class.

Any method that is not declared **final** can be overridden in a subclass.

Don't confuse method **overriding** with method **overloading.** Overloading means having methods (or constructors) within the same class with the same name, but different argument lists.

Additional Methods

If your new class needs to implement additional behaviour, you can simply add new methods to the subclass.

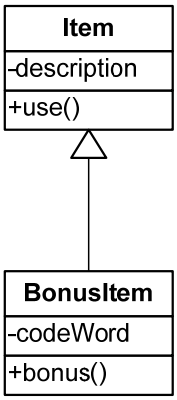
Inheriting from Object

Every class in Java extends some other class. If you don't explicitly specify the class that your new class extends, it will automatically extend the class named ***Object***. All classes in Java are in a class hierarchy where the class named *Object* is the root of the hierarchy.

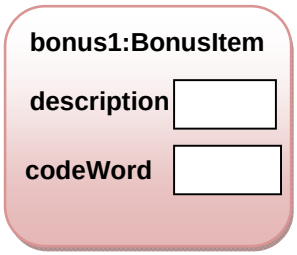
Some classes extend *Object* directly, while other classes are subclasses of *Object* further down the hierarchy.

The BonusItem class

The `BonusItem` class extends the `Item` class, and inherits its `use` method. It adds a new method, `bonus`, which prints out the value of a new field, `codeWord`. The class diagram for `Item` and `BonusItem` looks like this:



The object diagram for a situation where a `BonusItem` has been created looks like this:



Note that there is **only one object** here. In the other relationships we have seen, the classes are used to create two or more collaborating objects. Here, a **single object** is created by **combining template information from two classes**.

The code for the BonusItem class is as follows:

```
public class BonusItem extends Item
{
    private String codeWord;

    public BonusItem(String description, String codeWord)
    {
        super(description);
        this.codeWord = codeWord;
    }

    public void bonus()
    {
        System.out.format("This item's secret code word is %s\n", codeWord);
    }
}
```

new field – **description** field is inherited from Item

calls the constructor of Item to set description

new method – **use** method is inherited from Item

Polymorphism

The word “polymorphism” literally means “one name, many forms”. Polymorphism is an important idea in object-oriented programming. One form of polymorphism makes use of inheritance.

Here’s how it works. We can declare a variable of type `Item`, like this:

```
Item myItem;
```

This declaration says that there will be a variable called `myItem` which can refer to an object of type `Item`.

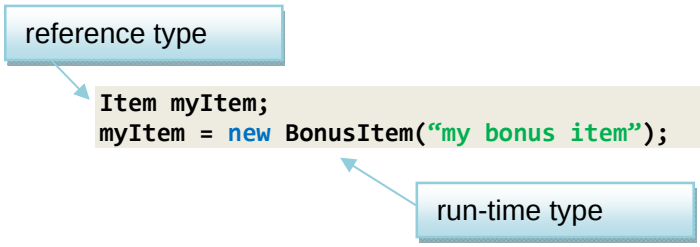
The object doesn’t exist yet. We need to create, or instantiate it, using the `new` keyword.

```
Item myItem;
myItem = new Item(“my item”);
```

Polymorphism allows us to do a trick here. A variable of type `Item` can refer to either:

- an `Item` object, OR
- an object whose type is a subclass of `Item`, for example `BonusItem`

This means we can do this:



It is possible to have a situation where a variable is declared with a specific type, known as the **reference type**, but the actual type of the object it refers to is not defined until the program is actually running. The actual object type is the **run-time type**. This is runtime polymorphism, sometimes also referred to as **late-binding**.

Note that polymorphism **doesn't work the other way**:

```
BonusItem myItem;
myItem = new Item(“my bonus item”);
```

Polymorphism in collections

Polymorphism is particularly useful when dealing with **collections of objects**. Think about the `Room` class. It has an `ArrayList` which can hold `Item` objects. Through polymorphism, a reference to an `Item` can also refer to any subclass of `Item`.

The result is that the `ArrayList`, `items`, in the `Room` class can hold `Item` objects, or `BonusItem` objects, or any combination of these. When we add an item to the room, we can add either one of `Item` or `BonusItem`.

We can use this when we set up the game. Any combination of `Item` and `BonusItem` objects can be added to any room. For example:

```
Room theatre = new Room("in a lecture theatre");
theatre.addItem(new Item("projector"));
theatre.addItem(new BonusItem("screen","BLUEJ"))

Room lab = new Room("in a computing lab");
lab.addItem(new BonusItem("computer","JAVA"));
lab.addItem(new Item("printer"));
```

Casting

We have to be careful when using polymorphism. Look at this code:

```
Item myItem = new BonusItem(“my bonus item”);
myItem.bonus();
```

At first sight, this looks OK. However, the second line will cause a compiler error.

Although the object `myItem` has **run-time type** `BonusItem`, the **reference type** is still `Item`. You cannot call a method which is not defined in the object's reference type. The method `bonus` is **only defined in the subclass** `BonusItem`.

The solution is to convert, or **cast,** the object to its run-time type, like this:

```
BonusItem myBonus = (BonusItem) myItem;
myBonus.bonus();
```

This is called **downcasting**. We have cast the *Item* reference *myItem* to type *BonusItem* and assigned it to a reference of type *BonusItem*. We can call the *bonus* method using this *BonusItem* reference.

Introducing interactivity - handling commands

We are now ready to move on and turn our game into a complete(ish) program.

Up to this point the adventure game is lacking in interactivity. There is no way for someone who is playing the game to control what happens. In a text-based adventure game, players interact with the game by typing **commands**. There is usually a limited set of commands which the game understands and which may cause some change in the game state. The user can type anything at all, but only valid commands will be understood.

An example of a command might be:

```
go west
```

The result of this command would be that the Player object would *go* to another room, using the exit from the current room marked *west*. The first command word (go) indicates the type of action to take, while the second command word (west) gives additional information about how to perform the action.

Some commands may have only one command word, for example:

```
help
```

This command would simply list the valid (first) command words in the game.

The Command class

A command is fairly simple –just one, or possibly two, strings. It will be useful, though, to have a class which represents a command. A `Player` object will then process a `Command` object within its `takeTurn` method, and perform the requested action. It will be easier to write the new code in `Player` to do this if it can get a command as a single object rather than two separate strings.

We can also put some additional methods into `Command` to make it more convenient to use. A method `hasSecondWord` will provide an easy way to check whether a one-word or two-word command has been entered. Another method `isUnknown` will provide an easy way to check whether an command with an invalid first word has been entered.

Here is the code for the `Command` class:

```
public class Command
{
    private String commandWord;
    private String secondWord;

    public Command(String commandWord, String secondWord)
    {
        this.commandWord = commandWord;
        this.secondWord = secondWord;
    }

    public String getCommandWord()
    {
        return commandWord;
    }

    public String getSecondWord()
    {
        return secondWord;
    }

    public boolean isUnknown()
    {
        return (commandWord.equals("?"));
    }

    public boolean hasSecondWord()
    {
        return (secondWord != null);
    }
}
```

Relationship between Player and Command

There needs to be a relationship between `Player` and `Command` because a `Player` object will need to be able to send messages to a `Command` object to, for example, get the command words.

The `Player` object does not need to own the `Command`, it simply **uses** it in order to get information about what action to perform. This is another example of the “uses-a” pattern.

CODE PATTERN: ***"USES-A"***

Problem: how do you implement a "uses-a" relationship, where an object needs to send a message to another object

Solution: the class which needs to send the message has a method parameter or local variable whose type is the name of the other class.

There is an interesting difference between the `Player`-`Item` relationship which you saw previously and the `Player`-`Command` relationship. An `Item` exists as part of the game world (and belongs to a Room). However, a `Command` object **only needs to exist while it is being processed**. Command objects are **temporary** objects.

The code pattern is similar, though. The revised version of the `takeTurn` method of `Player` now has a local variable of type `Command`.

```
public boolean takeTurn()
{
    Command command = ??
    return processCommand(command);
}
```

`processCommand` will be a new method in `Player` which will contain the code which performs the action indicated by the command. Note that we haven't yet decided how the `Command` will be created, so this method is still not complete.

The `takeTurn` method returns a boolean value, which will be used in the game loop to decide whether to exit the loop after this turn.

Turning user input into a Command

There is something missing here. We need something which will take the players' keyboard input and turn it into commands which can be processed. The player could potentially type anything at all – one word, two words or more; valid or invalid commands; complete or partial commands.

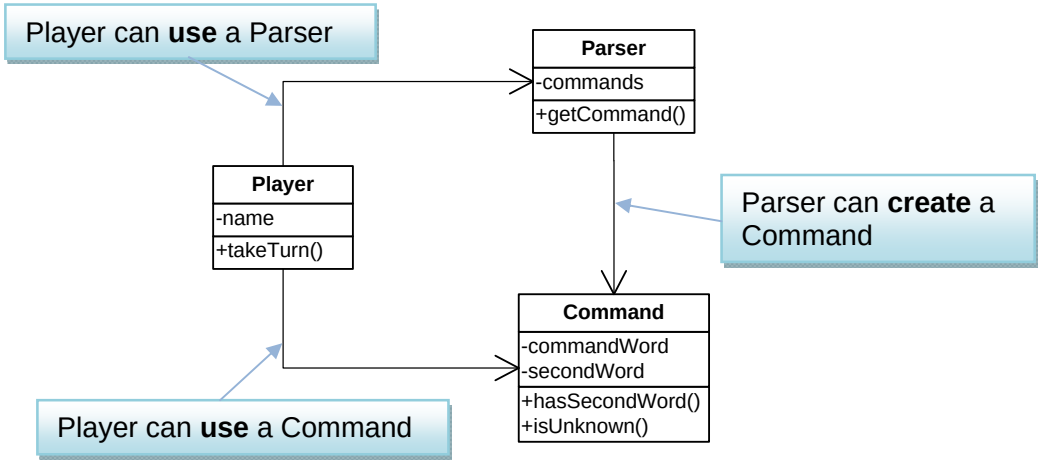
This "something" needs to:

- Read in a line of text typed at the command prompt
- Split the input text into individual words
- Check that the first word is a valid command word
- Construct a `Command` object using the first word and the second word (if there is one), ignoring any additional words

The Parser class

In an object-oriented program, "something" is usually an object. We will need objects which can do this job, and so we will need a class to allow these objects to be created. The class will be called `Parser`. A `Parser` object will **not** represent **information** in the game. Instead, `Parser` is class which **performs a specific role** in the program.

`Parser` is related to both `Player` and `Command`. Here is the class diagram for these classes:



The relationship between `Player` and `Parser` is very similar to that between `Player` and `Command` – a `Player` object uses a `Parser` object. The `Parser` only needs to exist while it is doing its job.

Relationship between Parser and Command

A key part of the job of a `Parser` object is to create a new `Command`. The relationship between `Parser` and `Command` is an example of a new pattern:

CODE PATTERN: ***"CREATES-A"***

Problem: how do you implement a "creates" relationship, where an object creates an instance of another class?

Solution: the class which creates the instance has a method which returns a value whose type is the name of the other class. The instance is newly constructed within this method.

Note that these **three classes work together** as follows:

- A `Player` uses a `Parser` to read input and create a `Command`
- The `Player` then uses that `Command` to decide what action to perform

The following listing shows some key features of the code for the `Parser` class. You can download the full code from the course website if you want to look at the compete class.

```
public class Parser
{
    private String[] commands;   // holds all valid command words
    private Scanner reader;      // source of command input

    public Parser(String[] commands)
    {
        this.commands = commands;
        reader = new Scanner(System.in);
    }

    public Command getCommand()
    {
        String inputLine;   // will hold the full input line
        String word1 = null;
        String word2 = null;

        System.out.print("> ");     // print prompt
        inputLine = reader.nextLine();

        // Find up to two words on the line
        ...

        // replace any invalid command word with ?
        if(!isValidCommand(word1))
        {
            word1 = "?";
        }

        return new Command(word1, word2);  // constructs and returns Command
    }

    private boolean isValidCommand(String commandWord)
    {
       // checks whether commandWord is in array of valid commands
       ...
    }

    public String showCommands()
    {
       // returns a list of valid commands
       ...
    }
}
```

We can now fill in the rest of the `takeTurn` method in `Player`:

```
public boolean takeTurn()
{
    Parser parser = new Parser(commands);
    Command command = parser.getCommand();
    return processCommand(command);
}
```

The variable `commands` is an array of type `String` which contains all the valid command words. The command list is defined as a field in `Player`, which is then passed into the constructor of `Parser`.

```
// valid command words
private String[] commands = {"go","quit","help"};
```

> **NOTE**
>
> The `Parser` and `Command` classes have no knowledge in advance of the actual list of valid commands, and will work with any list supplied by `Player` (or indeed by any other class which may use them). If we decide to add more commands later, then the only class which needs to be changed is the `Player` class.

Processing a Command

The `Player` class has a method `processCommand` which uses the command word of a `Command` to decide what action to take. It can do one of the following:

- print a message if the command word is “?” (the value set if the user input is not recognised)
- print a help message if the command word is “help”
- go to another room if the command word is “go”
- return `true` if the command word is “quit” – this will act as a flag to stop the game loop

In the case of a “go” command, the `Command` object will be passed to another method, `goRoom`, which will use the second word of the command to decide which exit to go through.

The code for `processCommand` is listed here.

```
private boolean processCommand(Command command)
{
    boolean quit = false;


    // get command word and use to select option
    String commandWord = command.getCommandWord();

    if(commandWord.equals("?")) {
        System.out.println("I don't know what you mean...");
    }
    else if (commandWord.equals("help")) {
        printHelp();
    }
    else if (commandWord.equals("go")) {
        goRoom(command);
    }
    else if (commandWord.equals("quit")) {
        System.out.println("the game will
              finish at the end of this round");
        quit = true;
    }
    return quit;
}
```

The switch statement

The sequence of **if** and **else** statements in the above code is a rather clumsy way of selecting from a list of choices based on the value of a variable. The **switch statement** is arguably more elegant and readable. The selection code above can be replaced with:

```
    // get command word and use to select option
    String commandWord = command.getCommandWord();
    char commandChar = commandWord.charAt(0);       // get first character

    switch(commandChar)
    {
        case '?':
            System.out.println("I don't know what you mean...");
            break;
        case 'h':
            printHelp();
            break;
        case 'g':
            goRoom(command);
            break;
        case 'q':
            System.out.println("the game will finish
                   at the end of this round");
            quit = true;
            break;
    }
```

Note that we are using the first character of the command word to select the option. The options in a switch statement can be integers or characters, but can't be strings.

> **NOTE**
>
> Microsoft's C# language, which is similar in many ways to Java, has a very similar switch statement. In C#, however, the options can be strings.

The goRoom method

The `goRoom` method is called if the command word is "go". Here is part of the code for this method, giving an outline of how this works.

```
public void goRoom(Command command)
{
    if(!command.hasSecondWord()) {
            System.out.println("Go where?");
    }
    else
    {
        String direction = command.getSecondWord();

        Room nextRoom = this.getCurrentRoom().getExit(direction);

        if (nextRoom == null) {
            System.out.println("There is no door!");
        }
        else
        {
            this.setCurrentRoom(nextRoom);
            System.out.println(this.getCurrentRoom().getDescription());
            // use the items in the room
            ...
        }
    }
}
```

The first part of the code checks whether the command has a second word – the player can't move unless a direction is specified.

If the command has a second word, then we ask the current room for an exit with a label matching the second word, using the `getExit` method. If there is an exit with this label, we change the current room of the player to the room which that exit refers to.

The complete version of this method also loops through the items in the new room and calls the `use` method of each one. You can download the full code from the course website if you want to look at the compete method.

The modified game loop

We are nearly finished the game. The last thing we will have to do is to modify the game loop, which is in the play method of `Game`. The last time we looked at this it simply gave each player one turn, then stopped. Now, we can make it continue looping until one of the players gives a quit command.

Note that the `processCommand` method returns `true` if the command is "quit". The `takeTurn` method in turn returns `true` to the code which calls it, which is the game loop.

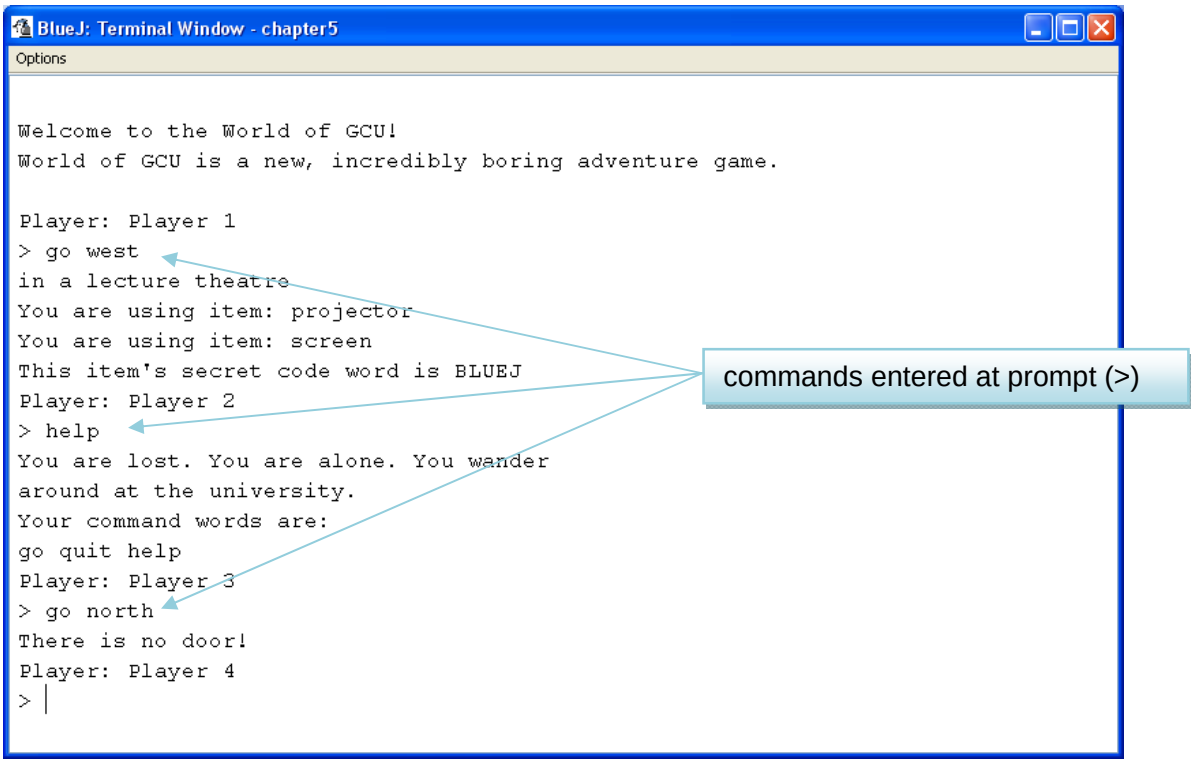
The game loop can then use the value returned by `takeTurn` to set the value of `finished`, the boolean variable it uses as a flag to stop the loop executing:

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

    // Enter the main command loop.
    // Here we repeatedly read commands and execute them until game is over
    boolean finished = false;
    do
    {
        for(int i=0;i<NUM_PLAYERS;i++)
        {
            System.out.println("Player: " + players[i].getName());
            boolean quitRequested = players[i].takeTurn();
            if(quitRequested)
            {
                finished = true;
            }
        }
    } while (!finished);

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

Running the game

We can run the game simply by right-clicking on the `Game` class in the BlueJ class diagram and selecting the `main` method. The output appears in the BlueJ terminal window.

Here is an example of game play



However, you do not expect users of your application to run it in BlueJ. Applications are usually run by **clicking on an icon** (for applications with a graphical user interface) or **typing a command at a command prompt**. We can package the game project so that it can be run at a system command prompt.

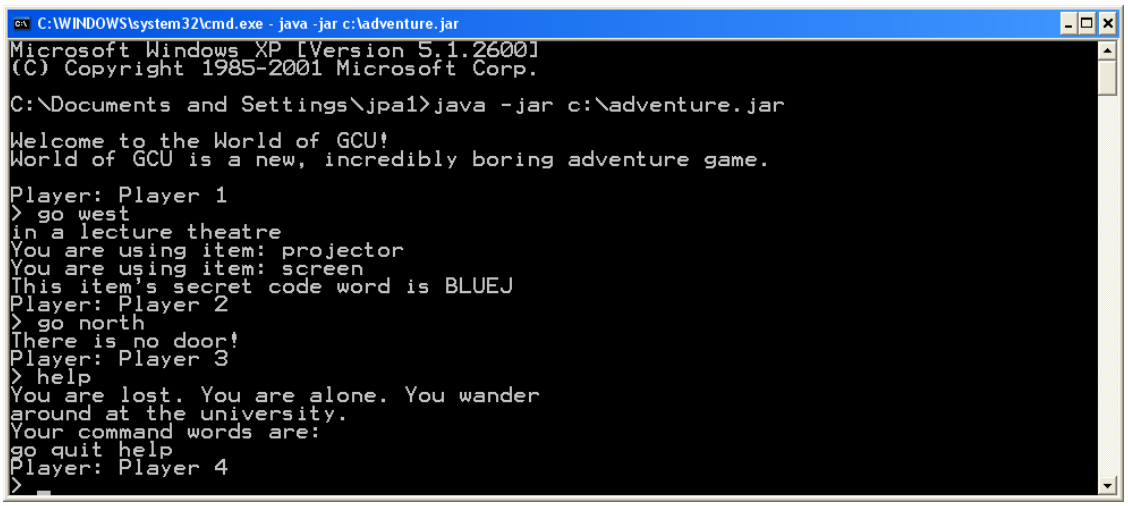
We select the Project > Create Jar File... menu option in BlueJ. This will package the contents of the project into a single, executable file, called a **Jar**. This is similar to a Windows .exe file.

The `main` method, which is the entry point which the operating system needs to launch the application, is in the `Game` class, so you need to specify that this is the main class in the Create Jar File dialog.

We can then name the jar file and save it in a suitable location, for example C:\adventure.jar.

The application can then be run by entering the command:

java –jar c:\adventure.jar



Compiling and running Java programs without an IDE

Throughout this module we have used the BlueJ IDE (Integrated Development Environment) to help manage the process of editing, compiling, testing, debugging and deploying Java applications. Most programmers use an IDE because it helps them to do their job and to be more productive. BlueJ is designed to help you to learn about object-oriented programming, while more advanced IDEs like NetBeans and Eclipse (for Java) and Visual Studio (for C#) will provide support as you develop and apply your skills.

It is, however, useful to know how to work “without a tightrope”. The Java JDK provides a range of command prompt tools which, together with a simple text editor, can be used to create and run Java programs without an IDE. We have just looked at one example, the java command, which can be used to execute a JAR file created with BlueJ.

Here, we will look at how the adventure game application can be compiled and run using command prompt tools. There are also many other tools in the JDK, including the javadoc tool for creating documentation.

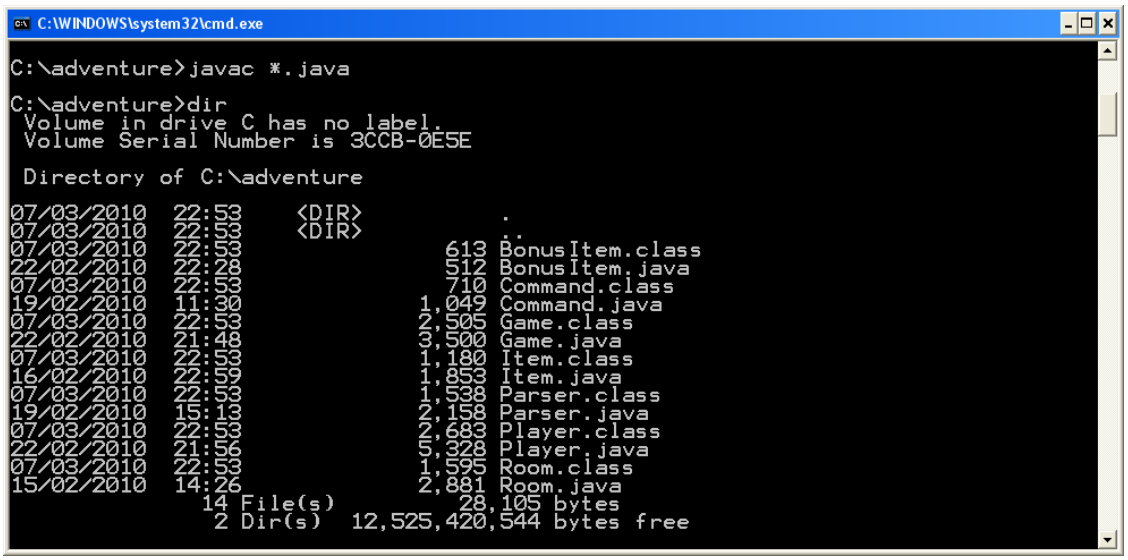
Compiling

The Java compiler is called **javac**. To compile a Java source file, for example Game.java, you use the command:

javac Game.java
In the figure below, the Java source files for the game are in a folder called *C:\adventure*, and this is the current working directory. The command:

javac *.java

uses the wildcard character * to select all Java source files in the folder and compile them. A compiled **.class file** is created for each class in the Java source files.



Setting paths

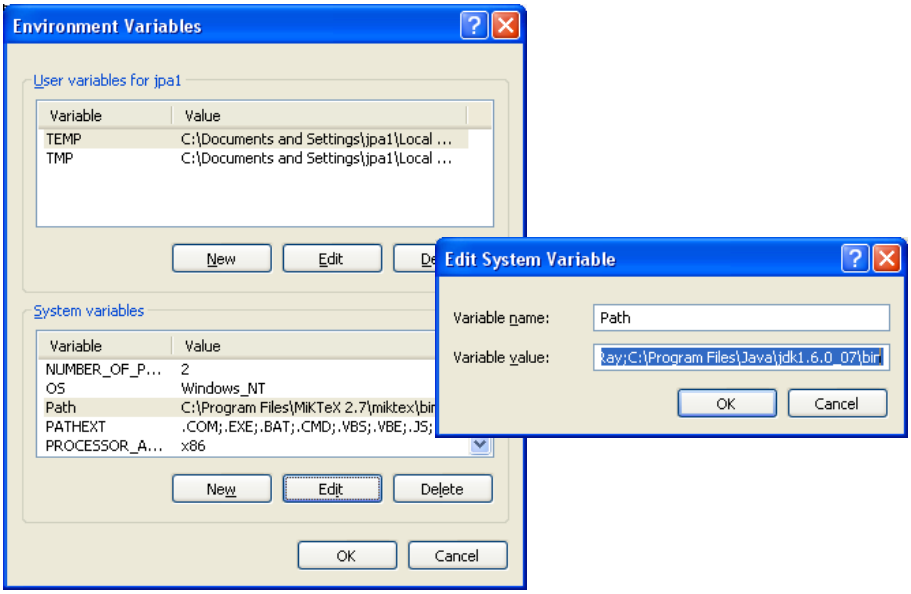
Note that for this to work, the folder which contains the file javac.exe needs to be in the current **path**. Javac.exe is usually in a folder called **bin** within the JDK installation folder.

In Windows, the PATH environment variable contains a list of folders which Windows will search through when it is asked to execute a file which is not in the current folder. Other operating systems which support Java have similar environment variables which need to be set.

We can set the PATH in Windows from the command line, or from the **Environment Variables** window, which is accessed by opening the System control panel, selecting the Advanced tab and clicking the Environment Variables button.

In the figure below, the path has been modified by appending the **bin** folder in the Java installation folder to the path.

***existing path*;C:\Program Files\Java\jdk1.6.0_07\bin**

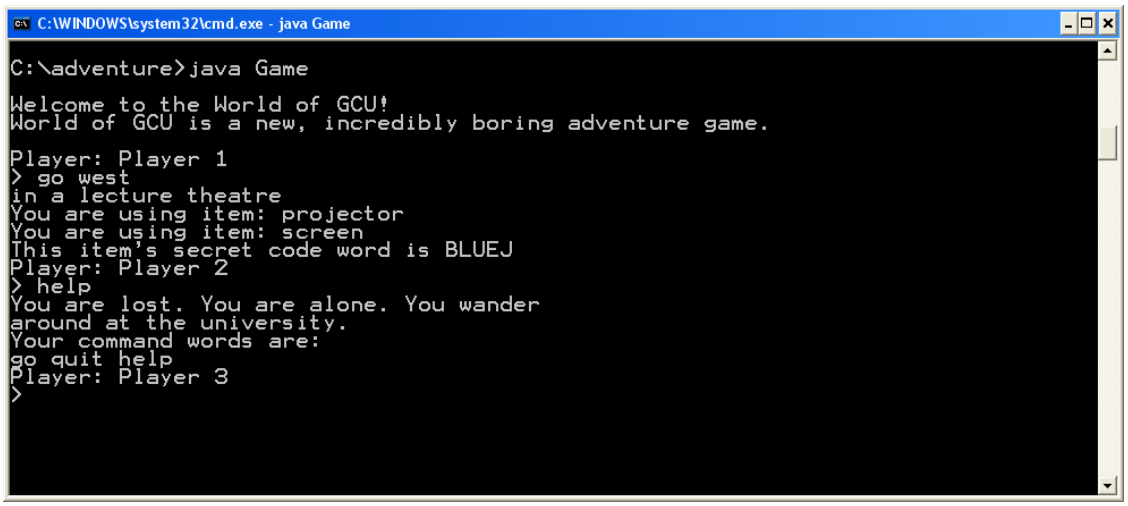


Note that there is also an environment variable **CLASSPATH**, which Java uses to search for compiled class files which may be needed when a program runs. We don't need to add anything in this example, as all the required classes are in the current folder, or are API classes which are included in the CLASSPATH by default.

Running

We can run the program by using the **java** command, specifying the name of the class which contains the main method. The command is:

java Game



When the Game class executes, it also requires the class files for the other classes in the game, for example Player.class and Room.class, which are in the current folder.

The **java** command is the same one we used earlier to execute a JAR file. The –jar option is required to execute a JAR. Note that the java command can also be used to create a JAR file. Deploying an application as a single JAR is more convenient than as a collection of separate class files.

What's next?

That's as far as we are going to go with this adventure game. We could add more features to make it a (much) more interesting game, but the basic structure of the game is there.

So what else do we need to learn about programming? Now that you know the basics, here's a few examples of exciting things you may go on to learn during your course:

- How to write programs with graphical interfaces
- How to write programs with web page interfaces
- How to write graphics-based games
- How to write programs which work with databases
- How to write programs which communicate over networks
- How to use other languages, such as C# or C++
- How to use more advanced development tools, such as NetBeans or Visual Studio