

## 4. Enhancing a class: Room

**Programming techniques in this chapter:** library classes, using API documentation

**Class associations in this chapter:** self relationships

The Java platform	1
Using library classes	3
Using the ArrayList library class	6
Testing the new Room class	9
Linking rooms together1	1
More changes to Room? Test Again!1	3
Creating the game "world" 1	4
What's next?	6

# The Java platform

Java is not just a programming language. It also provides:

- the Java Application Programming Interface (API) which is a set of library classes which Java developers can use within their own applications
- the Java Virtual Machine (JVM) which is the platform that allows Java programs to run on a computer

### Java APIs

There are actually many Java APIs, both official and third-party. We will look here only at classes which are provided within the official core Java API, which is contained within the Java Development Kit (JDK). The current version of the core API is known as **Java SE 7** – SE stands for Standard Edition. The Java SE API provides classes which help with many aspects of programming, for example for working with databases, for creating graphical user interfaces, for working with networks, etc. These classes are organised in **packages**, each of which contains a set of related classes.

The documentation for the Java SE API can be found at:

http://docs.oracle.com/javase/7/docs/api/

This website provides Javadoc pages for all the classes in the API, similar to the documentation that we did for our own classes in the previous chapter. It is **virtually impossible** to write a "real" Java program without referring to the API documentation –



there is simply too much information there for any developer to remember. Understanding how to use the API documentation is a crucial skill for all Java developers.

🥹 Overview (Java Platform Sl	E 7 ) - Mozilla Firefox				
<u>Eile E</u> dit <u>V</u> iew Hi <u>s</u> tory <u>B</u> ook	Elle Edit View Higtory Bookmarks Tools Help				
Overview (Java Platform SE 7 )					
+ http://docs.oracle.	com/javase/7/docs/api/	🟫 - C 🚼 - java 🔎 🍙			
Java™ Platform Standard Ed. 7	Overview Package Class Use Tre	Java™ Plafform e Deprecated Index Help Standard Ed. 7 Standard Ed. 7			
All Classes	Prev Next Frames No Frames				
Packages	.lava™ Platform_Star	adard Edition 7			
java.applet java.awt java.awt.color	API Specification				
All Classes	This document is the API specification f See: Description	or the Java™ Platform, Standard Edition.			
AbstractAction AbstractAnnotationValueVis AbstractAnnotationValueVis AbstractBorder	Packages Package	Description			
AbstractButton AbstractCellEditor	java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.			
AbstractColorChooserPane AbstractDocument	java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.			
AbstractDocument.Attribute AbstractDocument.Content	java.awt.color	Provides classes for color spaces.			
AbstractDocument.Elemen AbstractElementVisitor6	java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.			
AbstractExecutorService AbstractInterruptbleChanni AbstractLayoutCache AbstractLayoutCache	java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the OUI.			
AbstractList AbstractListModel AbstractMap	java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.			
AbstractMap.SimpleEntry 👽	java.awt.font	Provides classes and interface relating to fonts.			
	<.				
x		zotero 😻 Contra 🖬 🔒 🤗			

There are actually two different versions of the Java 7 SE platform:

### The Java Runtime Environment (JRE)

This provides the JVM and the core Java classes. If you want to simply run existing Java programs on a system, then it is sufficient to have a JRE installed on the system.

### The Java Software Development Kit (JDK or Java SDK)

This includes the JRE, together with compilers and debuggers and other development tools. If you want to develop Java programs, a JDK must be installed on your system.

### NOTE

Most object-oriented programming languages have a similar set of library classes. For example, Microsoft's C# and Visual Basic.NET are supported by the library classes in the **.NET Framework**, with web-based documentation on the MSDN (Microsoft Developer Network) website.



## The Java Virtual Machine

A Java source code file must be compiled to an intermediate language called **bytecode** before it can actually be run. Bytecode is interpreted by the **Java Virtual Machine** (JVM), which translates it into the machine code for the specific computer system it runs on, for example for an Intel x86 compatible processor on a PC. The bytecode is exactly the same for all systems.

You write and compile your program once, and it will run on any platform for which a JVM is available.

Java source files have a .java extension.

Bytecode (compiled) files have a .class extension.

**Advantage:** Java programs are portable between platforms with no additional programming effort – this is valuable as many organisations use a variety of computer systems.

**Disadvantage:** Java can be slower than other languages as it is interpreted rather than native machine code. It is possible to compile Java programs to native code for specific processors, and some Java development tools allow this, but the cross platform capability is then lost.

#### NOTE

Other programming languages often have similar platforms for running programmes. For example, C# and Visual Basic.NET source code is compiled to an intermediate language called **Microsoft Intermediate Language** (MSIL), which is in turn interpreted by a component called the **Common Language Runtime** (CLR).

# **Using library classes**

In this section we will look at a situation where the use of a Java SE API library class can improve a program and make it easier to write.

#### **Problems with arrays**

The Room class implemented in the previous chapter used an array to store its Item objects. Arrays are a core feature of the Java language. However, an array is not always the best way to store a collection of objects.

The Room class illustrates some problems with using arrays:



- The array size is fixed what happens if we want to add more items than the maximum number?
- We need to keep track separately of the number of items which have been stored in the array look at this code, from the getItem method of Room:

```
while(!found && i<numberOfItems)
{
    if(items[i].getDescription().equals(description))
    ...
}</pre>
```

What would happen if we did not keep track of numberOfItems, and tried to search through the whole array, like this?

```
while(!found && i<MAX_ITEMS)
{
    if(items[i].getDescription().equals(description))
    ...
}</pre>
```

If the array was not full, then some array elements would be **null references** - references to objects which have not been instantiated and do not yet exist. The call to items[i].getDescription() would then cause an error (actually an error called a **null pointer exception**) as you can't call a method of an object when the object does not exist.

• The code for adding items to and especially for removing items from the array is a bit clumsy and may involve moving elements around within the array.

All of these problems can be avoided by using a **library class** called ArrayList.

### The ArrayList class

ArrayList is a library class which is part of the JavaSE API. The name sounds quite similar to an array, but ArrayList has a number of very useful features which make it a better choice for use in the Room class.

Let's find the documentation for the ArrayList class:





Don't worry about understanding *all* of the content of the documentation – you will understand more of it as you learn more about programming. We will, however, be able to use the documentation now to get enough information to work out how to use ArrayList instead of an array in the Room class.

If we scroll down the main content frame (or click the <u>METHOD</u>) link we can find a list of the methods of the library class. This tells us what we can do with an instance of the class. Part of the list is shown below:

Method Summary				
Methods				
Modifier and Type	Method and Description			
boolean	add (E_e) Appends the specified element to the end of this list.			
void	add (int index, E element) Inserts the specified element at the specified position in this list.			
boolean	addAll (Collection extends E c) Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.			
boolean	addBll (int index, Collection extends E c) Inserts all of the elements in the specified collection into this list, starting at the specified position.			
void	<b>clear</b> ( ) Removes all of the elements from this list.			
Object	<b>clone</b> () Returns a shallow copy of this ArrayList instance.			
boolean	<b>contains (Object</b> o) Returns true if this list contains the specified element.			



There are many methods available, but the ones we will find most useful here are add, get, remove and indexOf.

#### NOTE

There are two versions of add listed. A class can have more than one method with the same name as long as the **signature** is different. The signature is the combination of the **return type** and the **list of parameters** required. The signature of the first version of add specifies a return type of boolean and a single parameter. The inclusion of several methods with the same name in a class is known as **method overloading**.

Similarly, we can find lists of **public fields** and **constructors** for an ArrayList.

r leid Summary			
Fields inherited from class java.util.AbstractList			
modCount			
Constructor Summary Constructors			
Constructor and Description			
Constructor and Description ArrayList () Constructs an empty list with an initial capacity of ten.			
Constructor and Description         ArrayList ()         Constructs an empty list with an initial capacity of ten.         ArrayList (Collection extends E c)         Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.			

## Using the ArrayList library class

We are now ready to start modifying Room to use an ArrayList instead of an array.

#### Importing the ArrayList class

Firstly, in order to use a library class, we need to add a line of code at the start of the Java file, before the class name, to **import** the class (or its package). If we don't do this, then the compiler will complain that it does not know about the library class. ArrayList belongs to the package java.util, so we need to specify the full name of the class, including the package name.

import java.util.ArrayList;



#### **Declaring an ArrayList**

The items field is now of type ArrayList. We don't need the MAX\_ITEMS constant or numberOfItems field any more. Why not? Well, an ArrayList is clever. It can **tell you how many** items it is currently holding – you just call its size method. Also, it does not have a maximum size. It can **resize itself automatically** to hold as many items as you add to it. If you keep adding, it will keep resizing as required.

The ArrayList library class is designed to store **any kind of objects**. Note that when we declare an ArrayList, we can specify what kind of objects it will store. ArrayList<Item> specifies an ArrayList which stores only Item objects.

The items field is declared in the modified Room class as follows:

```
public class Room
{
    // a description of the room
    private String description;
    // the items in the room
    private ArrayList<Item> items;
```

Note that the return type of the getItems method needs to change to match the new field type:

```
public ArrayList<Item> getItems()
{
    return items;
}
```

#### **Constructing an ArrayList**

The constructor of Room now needs to create a new ArrayList by calling an appropriate constructor of the ArrayList class. You can see from the documentation for the class that there is a default constructor which takes no parameters. Note that the initial capacity is 10, but this will change automatically if it needs to.

```
public Room(String description)
{
    this.description = description;
    items = new ArrayList<Item>();
}
```

#### Using methods of an ArrayList

We will really see the benefit of replacing the array of items with an ArrayList when we look at the methods for adding, finding and removing items from the Room.



Adding an Item is very simple. We just call the add method of ArrayList – no need to check whether we have reached a maximum number of items, or to keep track of the number of items.

```
public void addItem(Item newItem)
{
    items.add(newItem);
}
```

**Getting** a specified Item is slightly more complicated. We need to create a target Item object with the required value of description, and call the indexOf method of ArrayList to tell us the position in the list of the Item which matches the target Item. If there is no match for the target Item, indexOf returns a value of -1.

The get method of ArrayList then returns the Item at that position in the list.

```
public Item getItem(String description)
{
    Item target = new Item(description);
    int pos = items.indexOf(target);
    if (pos!=-1)
        return items.get(pos);
    else
        return null;
}
```

Removing an item is very similar, using the remove method of ArrayList.

```
public void removeItem(String description) {
    Item target = new Item(description);
    int pos = items.indexOf(target);
    if (pos!=-1)
        items.remove(pos);
}
```

#### NOTE

To allow indexOf to work for an ArrayList of Item objects, the Item class needs to have a new method called equals – you can see the equals method in the full code for this chapter which you can download from your course website.



The object diagram for the new version of Room is actually simpler than before:



Actually, if we looked inside the ArrayList object, we would find that it is considerably more complicated than an array (although, as the name suggests, there is an array in there). However, we don't need to look inside– all we need to know is that the object will handle the job of storing any items we give it, we don't need to know how it does it. This is an example of **information hiding** – a class hides everything about itself other than the **public fields and methods** it provides to allow other classes to use it. This is a key benefit of library classes, which can be **re-used** over and over again in different projects.

# **Testing the new Room class**

### Compiling

The modified version of the Room class should compile successfully. However, if we compile the whole project, the compiler will find an error in the Player class, in its takeTurn method:

```
Incompatible types - found java.util.ArrayList<Item> but expected
Item[]
```

This is because takeTurn in Player calls getItems of the current room to get the set of items which are available to be used in the room. We need to modify Player to use the **correct return type**. This is the only change – the for loop works exactly the same way as before.

```
public void takeTurn()
{
    System.out.println("Player " + name + " taking turn...");
    ArrayList<Item> items = currentRoom.getItems(); // changed type
    for(Item it : items)
    {
        if(it!=null)
            it.use();
    }
}
```



### Testing

Unit testing is very useful for making sure that changes in the way a class works "under the hood" don't change the way it behaves. Although the new Room class stores its items in a different way from before, a Room should still appear exactly the same to other objects as the previous version (apart from the change to the return type of getItems).

If this is true, then the RoomTest unit test class we used in the previous chapter should still pass. You can try this with the new version of the game project which you can download to convince yourself that it works.

We have had to change Player slightly, so we should test that too. We can test Player exactly as before. The sequence of steps described in chapter 3 for creating test objects to test Player can be repeated by creating a new test class, PlayerTest, with the following setup method.

```
protected void setUp()
{
    room1 = new Room("kitchen");
    item1 = new Item("cooker");
    item2 = new Item("fridge");
    item3 = new Item("knife");
    room1.addItem(item1);
    room1.addItem(item2);
    room1.addItem(item3);
    player1 = new Player("Joe");
    player1.setCurrentRoom(room1);
}
```

We can then repeat the test by right-clicking on PlayerTest and selecting Test Fixture to Object Bench



This creates our test objects in the Object Bench, and we can call the takeTurn method of the Player object and observe the output as before.

Note that this is a semi-automated test – the test objects are created automatically but the actual test is manual. Both fully-automated and semi-automated tests can be useful – fully automated tests are particularly good for testing a full project with a large number of classes.



# Linking rooms together

So far we can only have one room in the game because once a player has been placed in a room there is **no way out of that room**. We will need to make it possible for rooms to have **exits**. An exit should lead into another room, so that players can move from one room into another. This will allow us to join rooms together to design a game "world".

This means that a room must be linked to, or associated with, the other room or rooms next to it in the game world. This is shown in the class diagram by a **self-relationship**. A Room object can be associated with other Room objects.



### A new code pattern

This is an example of the "self" code pattern.

#### CODE PATTERN: "SELF"

**Problem:** how do you implement a self- relationship, where an object is linked to other objects of the same class

Solution: the class has a field whose type is the same as the name of the class



Here's the code for this:

```
public class Room
{
    // a description of the room
    private String description;
    // the items in the room
    private ArrayList<Item> items;
    // the exit from the room
    private Room exit;
```

This is still pretty limiting, as room could only have **one exit**. We want to give the game players a choice of which room to move into, so we will need to allow a room to have several exits.

How about using an ArrayList? This would be possible. However, there may be another library class which would be a better choice here. The Java SE API provides many classes for storing collections of objects in different ways (this is known as the **Collections Framework**).

It would be useful to be able to store each exit along with a **label** to identify it. This will make it easy to select a room based on the player's choice. For example, a room might have exits labelled "east", "west", and so on. The player will then be able to choose to "go east", or "go west", or go in whatever direction the available labelled exits allow.

#### Using a HashMap

The HashMap library class is a good choice here. You can look up the documentation for this and see if you agree. A HashMap stores objects (as many as needed) along with a label, or **key**, for each one. A HashMap<String, Room> will store Room objects each with a String as its key.

First, we must import the HashMap class:

import java.util.HashMap;

We can then declare the exits in Room like this:

```
// the exits from the room
private HashMap<String,Room> exits;
```

We can construct the HashMap like this:

```
exits = new HashMap<String,Room>();
```

What else do we need to do? Well, when we are building the game world we will need to be able to **set the exits** for each room so that the rooms are linked together the way we

```
M1G413283: Introduction to Programming 2
```



want them to be. When the player chooses which direction to go, we will need to be able to **get the required room** which is the exit in the chosen direction, so that the player can use the items in the room. These tasks may sound difficult, but they are actually really easy with a HashMap.

### Getting objects into and out of a HashMap

The documentation for HashMap shows the following useful methods:

```
get (Object key)
Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
```

```
<u>put</u>(K key, V value)
```

Associates the specified value with the specified key in this map.

We can use these to help add setExit and getExit methods to our Room class:

```
public void setExit(String direction, Room neighbor)
{
    // adds a Room object to the HashMap with a String label
    exits.put(direction, neighbor);
}
public Room getExit(String direction)
{
    // gets the Room object stored with the specified label
    return exits.get(direction);
}
```

## More changes to Room? Test Again!

We can add another test to RoomTest to test the new capability which has been added to Room. The new test will create some more Room objects, and set these as the exits of the test Room. It will then check that the correct Room is returned by getExit. Note that setUp is shared by all test methods and must create all the test objects for the class.

```
protected void setUp()
{
    room1 = new Room("kitchen");
    item1 = new Item("cooker");
    item2 = new Item("fridge");
    item3 = new Item("knife");
    room1.addItem(item1);
    room1.addItem(item2);
    room1.addItem(item3);
```



```
room2 = new Room("dining room"); // other rooms declared as fields
room3 = new Room("hall");
room4 = new Room("study");
room1.setExit("east",room2);
room1.setExit("north",room3); // north exit is hall
room1.setExit("west", room4);
}
public void testGetExit()
{
    Room target = room1.getExit("north"); // north exit should be hall
    assertEquals("hall", target.getDescription());
}
```

When we run the tests in BlueJ, the result of two test methods is now shown:

🌁 BlueJ: Te	st Results	
✔ RoomTest.te ✔ RoomTest.te	stGetItem stGetExit	
Runs: 2/2	X Errors: 0	X Failures: 0

# Creating the game "world"

We can now add a method to the Game class to create the game "world" as a set of linked rooms. In the game, all players will start in the same room and can then navigate through the world by moving from room to room. The Game object only needs to store a reference to the starting Room object – each room, as we have seen, will know about the other rooms it is linked to.

```
public class Game
{
    // the starting room
    private Room startingRoom;
```



Here is a map of a possible game world.



This world can be created by the following getRooms method which is called from the setUp method when a new Game is created.

```
private void createRooms()
   {
        Room outside, theatre, pub, lab, office;
        // create the rooms and put some items in them
        outside = new Room("outside the main entrance of the university");
        outside.addItem(new Item("phone"));
        outside.addItem(new Item("bin"));
        theatre = new Room("in a lecture theatre");
        theatre.addItem(new Item("projector"));
        theatre.addItem(new Item("screen"));
        pub = new Room("in the campus pub");
        pub.addItem(new Item("fruit machine"));
        lab = new Room("in a computing lab");
        lab.addItem(new Item("computer"));
        lab.addItem(new Item("printer"));
        office = new Room("in the computing admin office");
        office.addItem(new Item("computer"));
        office.addItem(new Item("filing cabinet"));
        // initialise room exits
        outside.setExit("west", theatre);
        outside.setExit("south", lab);
        outside.setExit("east", pub);
        theatre.setExit("east", outside);
```



```
pub.setExit("west", outside);
lab.setExit("north", outside);
lab.setExit("west", office);
office.setExit("east", lab);
startingRoom = outside; // start game outside
}
```

Note that these exits are all designed to be two-way doors – for example the east exit of the admin office is the computer lab, while the west exit of the lab is the office. This will allow players to move freely around the world.



## What's next?

The players are still stuck in the starting room! In the final chapter we will complete the game by implementing a Command class which will allow players to perform actions such as moving from room to room. We will also use inheritance to allow rooms to contain different types of item.