3. Creating classes: Room and Item

Programming techniques in this chapter:
stepwise refinement, unit testing, documentation, enhanced for loop

Class associations in this chapter:
“has-a” relationships, “uses-a” relationships

Interaction between rooms and items

In this chapter we will create Java code to implement the `Room` and `Item` classes. The link between a room and its items is similar in some ways to that between the game and its players. Each room can hold several items, and there is a **“has-a”** relationship between `Room` and `Item`.

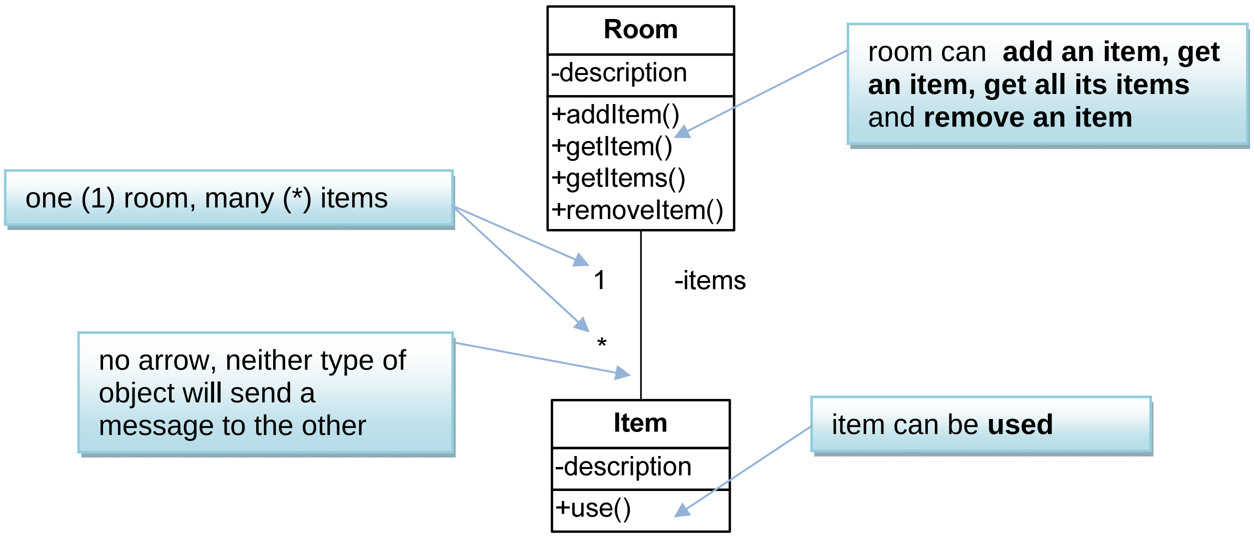
There are some differences, however:

- There will be more than one room in the game world
- Each room may have a different number of items
- We would like to be able to add and remove items from a room as the game progresses

The interaction will need to be a bit different too. The `Item` class will have a behaviour called `use`, which will perform whatever action the item is used for. Another object can then send a message to an item by calling the `use` method. However, it would **not make sense** for the **room to use the item**. What kind of object do you think should use an item?

The room simply has the job of holding the items, and providing these items to other objects to use. It can either provide one item (`getItem`) or its whole set of items (`getItems`).

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.


Room
-description
+addItem()
+getItem()
+getItems()
+removeItem()
room can add an item, get an item, get all its items and remove an item
one (1) room, many (*) items
1
-items
*
no arrow, neither type of object will send a message to the other
Item
-description
+use()
item can be used

The Item class in Java

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

```
/**
 * Class Item
 * Represents an item, in the game
 *
 * @version 1.0
 */
public class Item
{
    // a description of the item
    private String description;

    /**
     * Constructor for objects of type Item
     *
     * @param description   value for the description example property
     */
    public Item(String description)
    {
        this.description = description;
    }
    /**
     * gets the value of description
     *
     * @return    the value of description
     */
    public String getDescription()
    {
        return description;
    }

    /**
     * sets the value of description
     *
     * @param  description   the new value of description
     */
    public void setDescription(String description)
    {
        this.description = description;
    }

    /**
     * use the item by asking it to perform some action
     */
    public void use()
    {
        System.out.format("You are using item: %1$s\n", description);
    }
}
```

Testing the Item class

You can try creating this class for yourself in BlueJ in the same way as you tested the `Player` class previously.

Implementing the interaction

If you look through the code for the `Item` class, there is no mention of the `Room` class. That is because an `Item` object does not need to communicate with the `Room` it is in.

As we said before, the `Room` does not need to communicate with its items. However, it will need to have the ability to **store** its items. This interaction can be implemented using the same "HAS-A(ARRAY)" pattern which we used for `Game` and `Player`. Remember that the solution for that pattern was:

"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."

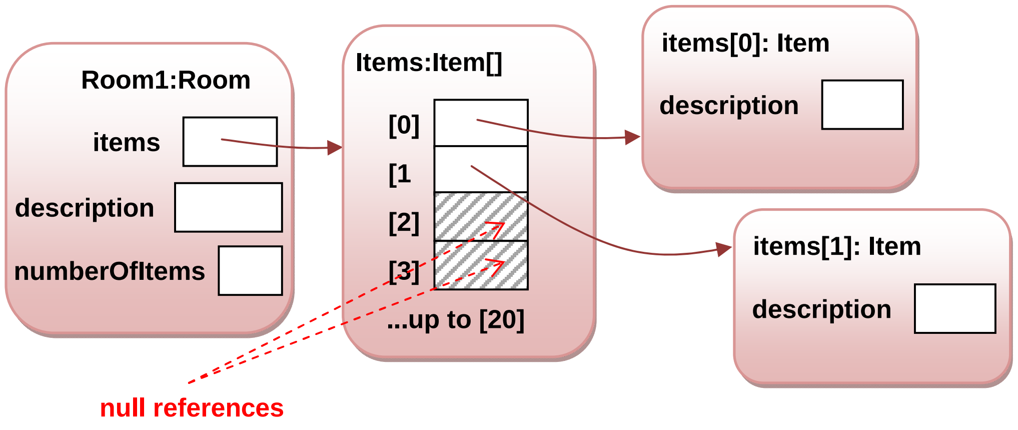
The outline of the Room class looks like this.

```
public class Room
{
    // a description of the room
    private String description;
    // the maximum number of ITEMS
    private static final int MAX_ITEMS = 20;
    // the number of items currently in the room
    private int numberOfItems;
    // the items in the room
    private Item[] items;

    public Room(String description)
    {
        this.description = description;
        items = new Item[MAX_ITEMS];
        numberOfItems = 0;
    }

    public String getDescription()
    {
        return description;
    }

    public void addItem(Item newItem)
    {
        ...
    }

    public Item[] getItems()
    {
        return items;
    }

    public Item getItem(String description)
    {
         ...
    }

    public boolean removeItem(String description)
    {
         ...
    }
}
```

We need to specify a size for the array, which is the constant MAX_ITEMS. However, the array may hold any number of objects up to the maximum, so we also need to keep track of the **number of items which have been added.** This will let us make sure that when we add a new item, it goes into the right place in the array.

The array when it is first created contains null references – that is each element in the array should point to an Item object, but no Item objects have been created yet. When the first Item is added, the first element of the array will point to it, while the remaining elements are still null references.

An object diagram for a room and its items might look like this (the array contains two Items here):

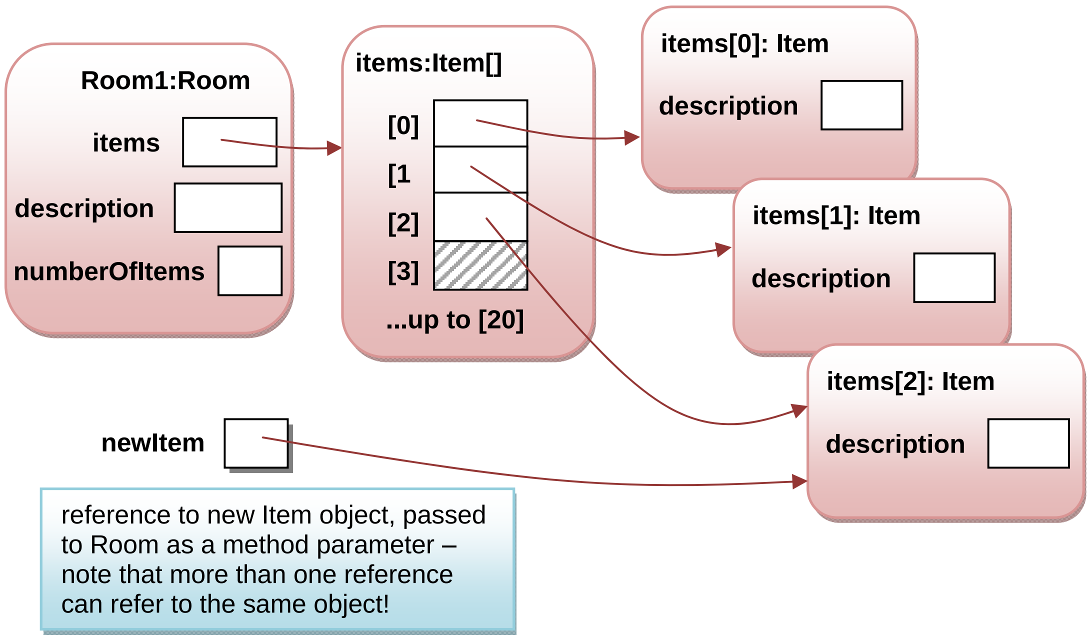


Adding an `Item` to a `Room` – the `addItem` method

Now we need to fill in the details of the methods of the `Room` class. First, the `addItem` method. This simply takes an `Item` object and sets the next available array element in `items` to refer to that object.

```java
public void addItem(Item newItem)
{
      if(numberOfItems < MAX_ITEMS)         // check array is not full
      {
          items[numberOfItems] = newItem;
          numberOfItems++;                  // increment by one
      }
}
```

The object diagram would look like this after this method runs:



An example of the use of this method will be within the `setup` method of the `Game`, which might have code like this:

```
Item newItem = new Item("torch");  // creates a new Item object
startingRoom.addItem(newItem);     // sets next available array element in
                                   // startingRoom.items to refer to new Item
```

The `getItem` and `removeItem` methods are a little bit more complicated, and will require a bit of thought to work out the **algorithms** which are needed to implement them. We will look at `getItem` here, and leave `removeItem` as an exercise.

Algorithms and stepwise refinement

An algorithm is a list of well-defined steps for completing a task. We need to work out what steps are required to search for a particular Item and to remove a particular Item.

The `getItem` method must take a string as a parameter, and return the `Item` in the `items` array which has that value of description. How do we find a particular element in an array? The simplest way to do this is to start at the beginning of the array, and check each element in turn to see if it is the one we want. This is called a **linear search**.

We can write the algorithm for the search using **pseudocode**. Pseudocode is not a real programming language – it is simply a way of thinking about and writing down what the steps for an algorithm are without worrying about the details of the actual code until we are clear about how it will work.

The algorithm for `getItem` needs to repeatedly check items until the required one is found or we run out of items, and a while loop is ideal for this. Here's a first attempt:

```
1. while (target not found and end of array not reached)
2.      check next item
3. return target item
```

NOTE

There are no rules for pseudocode, as it is not a real language. It's just a way of writing down steps in an algorithm while you're designing it. Not everyone uses pseudocode – it's up to you whether you find it useful or not.

The step in line 2 looks as if it needs a bit more thought. How do we check the item and what do we do when the item is the target item? We'll need to keep count of how far through the array we get so that the correct item can be returned. We'll also need to set a **flag** which will indicate that the target item has been found.

Let's **refine** this algorithm to put a bit more detail in it. The numbering here helps us keep track of how the lines from the first version have been refined:

```
1.1  count = 0
1.2  set target not found
1.3. while (not found and end of array not reached)
2.1      if item is target
2.2              set target found
2.3      increment count
3.1  return item from array with index = count
```

This process of adding more detail to an algorithm is called **stepwise refinement**. We have done two steps here, and we're pretty much ready to write the real Java code. A more complicated algorithm might require more refinement steps.

These lines look as though they can be translated directly into Java. Here is the code for `getItem`:

```
public Item getItem(String description)
{
      int i = 0;
      boolean found = false;
      while(!found && i<numberOfItems)
      {
          if(items[i].getDescription().equals(description))
          {
              found = true;
          }
          i++;
      }
      return items[i];
}
```

This code looks OK, and it will compile. However, it has at least two serious flaws. Can you see what they are?[1]

Before we move on, we really need to do some testing of the *Room* class to check that the methods really do what they are supposed to do.

Unit testing

The process of testing an individual class is known as **unit testing**. This contrasts with testing the complete program. Unit testing is vitally important in object oriented programming. Unit tests can be repeated as we continue to develop a class to make sure that we don't inadvertently break a part of a class which was already working correctly.

We've already done a little bit of unit testing. We used BlueJ to run some tests on the first versions of the *Player* and *Game* classes. Good unit tests actually need some careful thought to make sure that they test a class thoroughly in all possible circumstances.

So far, the *Room* class has methods to add an item and to get a specified item. Here are some test cases that we should run. If any of these actions do not produce the desired result, we will need to revise the code and try again.

- Add items to the array
- Add sufficient items to fill the array
- Try to add an item when the array is full (the desired result is that this should not work)
- Get an item which is in the array

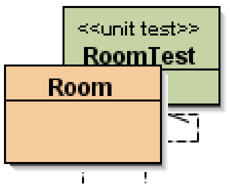
[1] Variable i will have wrong value as it is incremented after target found ; shouldn't try to return an array element if the target is not found

- Try to get an item which is not in the array when the array is not full, and when it is full

Creating a test class

The best way to perform unit testing on a class is to use a test class, A test class is a special kind of class which defines the way a specific model class will be tested. BlueJ can help you create test classes. It makes use of a test framework called JUnit to do so.

If you right-click on the `Room` class in the BlueJ class diagram you can select Create Test Class from the popup menu. A new class called `RoomTest` will be added to the project – it will be coloured green in the diagram to show that it is a test class.



Open the `RoomTest` class for editing. You will see that it has a method called `setup()`. This method can be edited so that it sets up some test objects. Add instance variables and code in the setup method as follows. This will create some test objects – a `Room` and three `Items`. It then adds the Items to the `Room`.

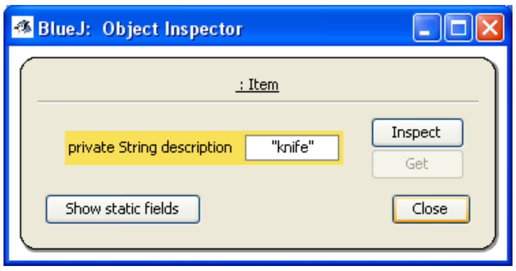
```
    private Room room1;
    private Item item1;
    private Item item2;
    private Item item3;
...
    protected void setUp()
    {
             room1 = new Room("kitchen");
             item1 = new Item("cooker");
             item2 = new Item("fridge");
             item3 = new Item("knife");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

This saves us from creating these objects manually every time a test is run. If we now right-click on the `RoomTest` class in the class diagram, we can select the Test Fixture to Object Bench option. The `Room` and three `Item` objects now appear in the Object Bench. You can inspect the `Room` to see that the items are there in the `items` array.

Running a test

Let’s run one of our tests – *Get an item which is in the array.*

Right click on the Room object and select getItem(string description) . Enter the text “fridge” in the method call dialog and click OK. Click Inspect to have a look at the Item which is returned in the Method Result dialog.


BlueJ: Object Inspector

: Item

private String description "knife"

Wait a minute! That’s the wrong item. Our Room class has failed this test dismally. The getItem method does return an Item, but it is the wrong one.

Look at the code for getItem – can you see why this is happening? The following version of the method fixes the problem:

```
public Item getItem(String description)
{
      int i = 0;
      boolean found = false;
      Item target = null;
      while(!found && i<numberOfItems)
      {
          if(items[i].getDescription().equals(description))
          {
              target = items[i];
              found = true;
          }
          i++;
      }
      return target;
}
```

Now we can run the same test on the modified Room class – it should pass this time.

Test methods and automated testing

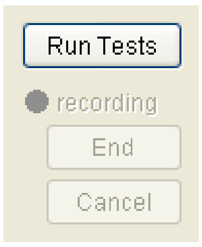
The RoomTest class we have created helps us by setting up the same test objects in the object bench each time we want to run a test. The actual test was done manually. If we want to do a lot of tests, this can become very time-consuming. It is often useful to have a series of tests which are run automatically. We can add test methods to the RoomTest class which BlueJ can then run automatically.

Let’s add a test method to run the same test as before, but this time to do it automatically. Add the following method to RoomTest.

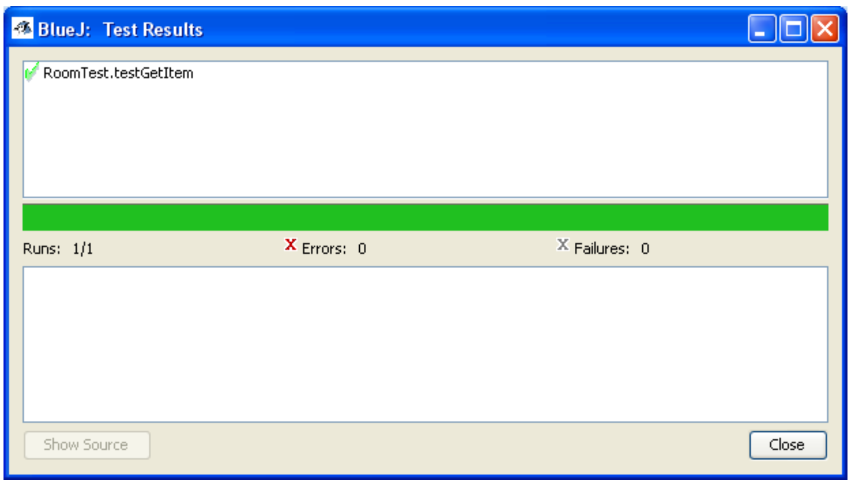
```
    public void testGetItem()
    {
        Item target = room1.getItem("fridge");
        assertEquals("fridge", target.getDescription());
    }
```

`assertEquals` is a special method used by BlueJ to decide whether a test is passed or failed. It checks that the actual description value of the Item which is returned (`target.getDescription()`) is the same as the expected value, “fridge”. If not, then the wrong item has been returned, and the test should fail.

BlueJ runs tests automatically when you click the Run Tests button. It runs all test methods in all test classes which have been added to the current project. This is very useful when you have a large project with many classes. Sometimes making a change to the code in one class can cause another related class not to work correctly. Testing the whole project regularly ensures that this kind of problem is discovered quickly. This is called **regression testing**.



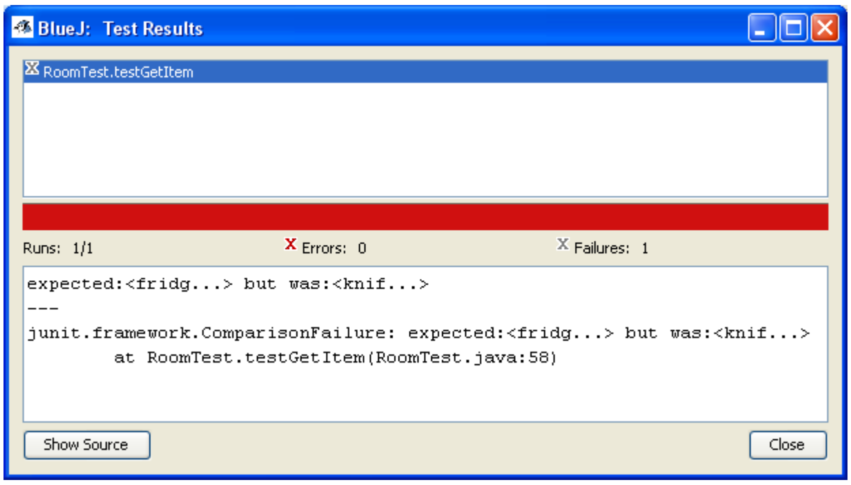
At the moment we only have one test class with one test method. If we click Run Tests, the `testGetItem` test method will run, and the result is shown in the Test Results window. The solid bar is coloured green to indicate a pass, and changes to red if any test does not pass.



RoomTest.testGetItem

Runs: 1/1 Errors: 0 Failures: 0

If we run the test on a version of `Room` with the first, incorrect, version of the `getItem` method, then the test fails and we can see the details of the failure in the Test Results window:



```
expected:<fridg...> but was:<knif...>
---
junit.framework.ComparisonFailure: expected:<fridg...> but was:<knif...>
	at RoomTest.testGetItem(RoomTest.java:58)
```

NOTE

BlueJ uses a popular unit test framework called **JUnit** to run tests. JUnit is an important tool for Java programming, and is also built into many other Java IDEs.

Documentation

A Java object in an object-oriented program is likely to be used by other objects. For example, in the completed game, an `Item` object will be used by a `Player` object. The **interface** of a class specifies how objects of that class may be used.

The interface of a class consists of:

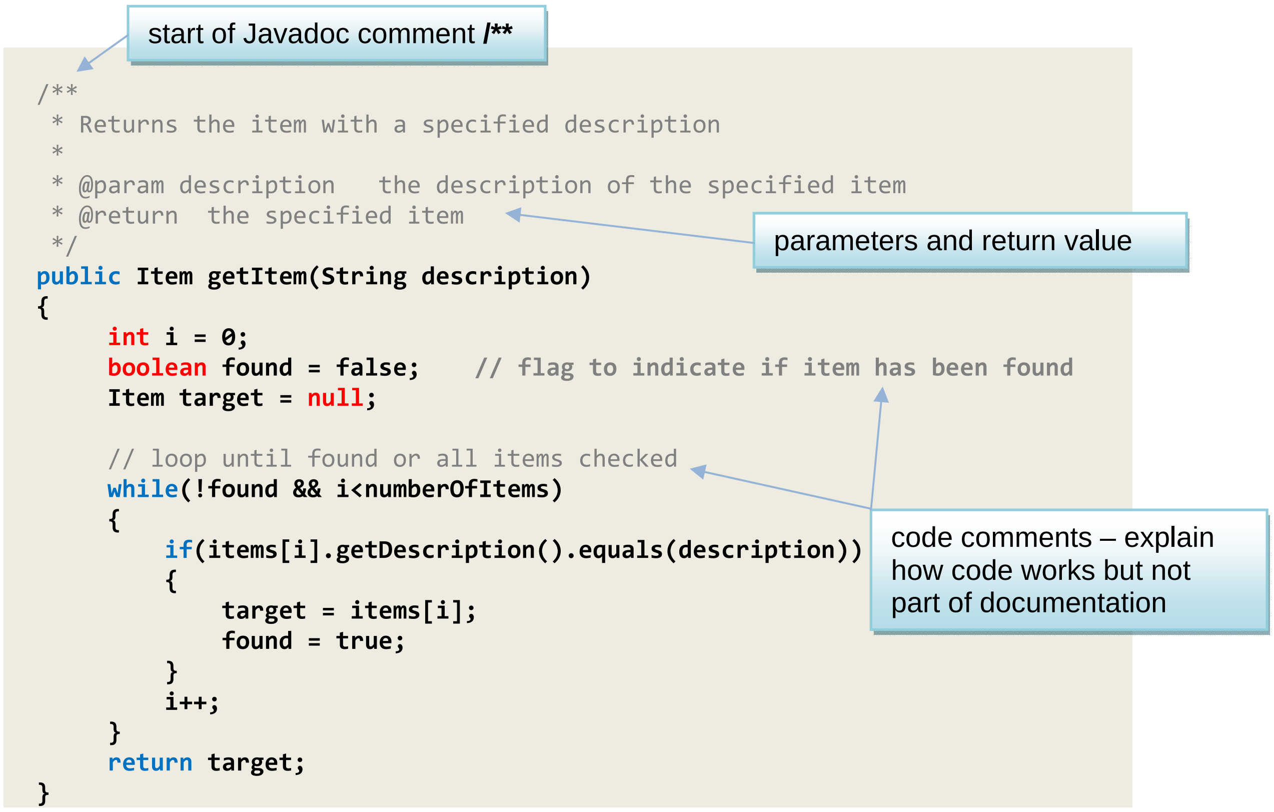
- **Public fields** – fields declared with the `public` key word
- **Public methods** – methods defined with the `public` key word

It does not include

- **Private fields** – if values of `private` fields need to be accessed or updated by other objects then there should be public getter and/or setter methods
- **Private methods** – some methods may be used by the public methods in a class but are not themselves available for other objects to use

It is helpful to designers of classes which use your class if you take time to carefully **document the interface** of your class. Documentation is done by writing **Javadoc**

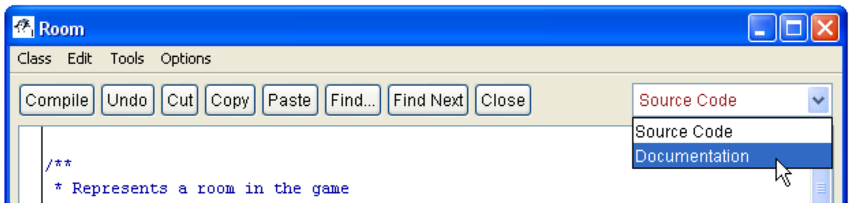
comments in your code. An example of documentation for the `getItem` method in `Room` is shown below:


start of Javadoc comment /**
parameters and return value
code comments – explain how code works but not part of documentation

```
/**
 * Returns the item with a specified description
 *
 * @param description   the description of the specified item
 * @return  the specified item
 */
public Item getItem(String description)
{
     int i = 0;
     boolean found = false;    // flag to indicate if item has been found
     Item target = null;

     // loop until found or all items checked
     while(!found && i<numberOfItems)
     {
         if(items[i].getDescription().equals(description))
         {
             target = items[i];
             found = true;
         }
         i++;
     }
     return target;
}
```

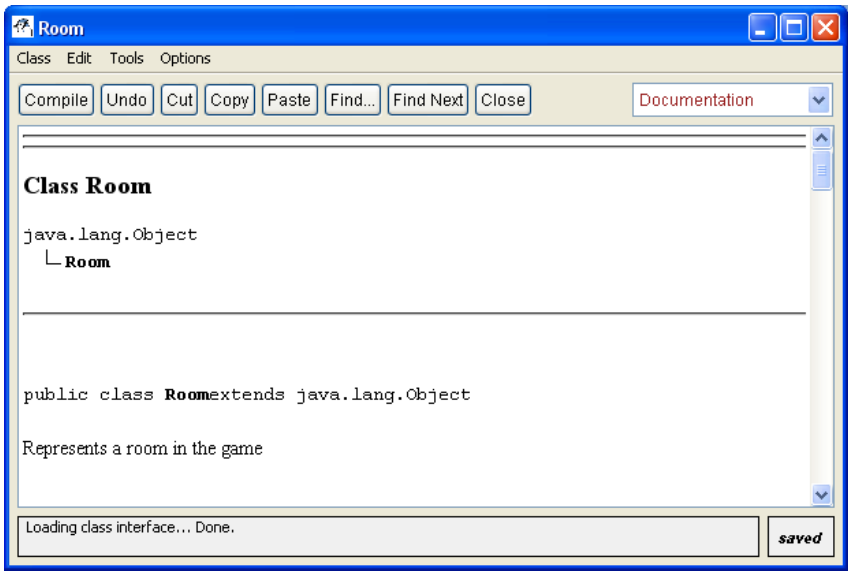
Viewing documentation

Javadoc comments can be used by the **Javadoc tool** to automatically generate a set of HTML pages which document a single class or an entire programme. They are also used by the BlueJ editor when you choose to view a class as **Documentation** instead of **Source Code**.



```
/**
 * Represents a room in the game
```

BlueJ then shows the HTML documentation page for the class.



Class Room

```
java.lang.Object
  └Room
```

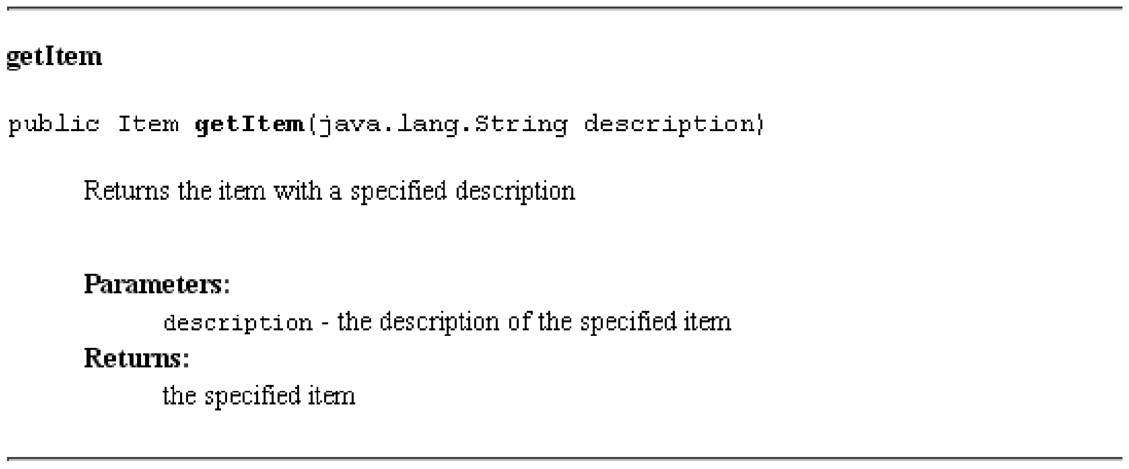
```
public class Roomextends java.lang.Object
```

Represents a room in the game

Loading class interface... Done.

saved

The Javadoc comment in the listing above produces the following content in the documentation page:



getItem

```
public Item getItem(java.lang.String description)
```

Returns the item with a specified description

Parameters:
`description` - the description of the specified item

Returns:
the specified item

Code comments

Not all comments in code are used for documentation. Ordinary **code comments** like those indicated in the listing (the ones which don't start with /**) are there to explain how the code works. Those comments are there to help someone who may, for example, have to modify the code later on. They are **not** there to help the programmer who is writing code which **uses** the `Room` class.

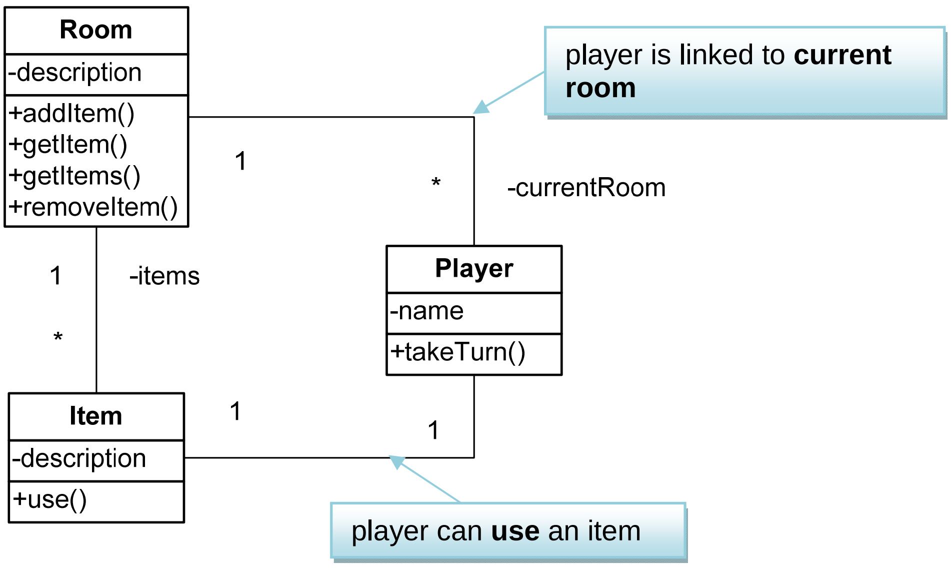
To use the `Room` class it is only necessary to know that it has a `getItem` method which takes a description as a parameter and returns an `Item` – it is not necessary to know how the getItem method actually does its job. This is a bit like driving a car – you need to know, for example, that it has a brake pedal which causes the car to slow down if you press it. You don't need to know how the brake mechanism works.

Code comments do not appear in the documentation. Private fields are often described with code comments to explain their purpose.

The Player class - using the Item and Room classes

We have now created and tested initial versions of the `Room` and `Item` classes. These classes exist in the game because a `Player` needs to be located in a `Room` and can use Items in the room. The `Player` class therefore needs to be able to interact with `Room` and `Item`.

Let's add the `Player` class to the class diagram which was shown earlier in this chapter:



Player and Room

A `Player` object needs to maintain an association with the current room in which the player is located. This is an example of the **"has-a" code pattern** which we have seen before:

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.

What message will a `Player` object need to send to a `Room` object? Well, the room stores an array of items, so the player may need to ask the room for to provide it with a list of the available items, and a reference to the specific `Item` object it wants to use.

The association between `Player` and `Room` will therefore be implemented by having a field of type `Room` in the `Player` class.

```
public class Player {

    // the player's name
    private String name;
    // the room in which the player is currently located
    private Room currentRoom;
```

Player and Item

A `Player` object does not need to **own** any items (these belong to a room, not a player), but it may need to **use** an `Item` object. This is an example of a new coding pattern:

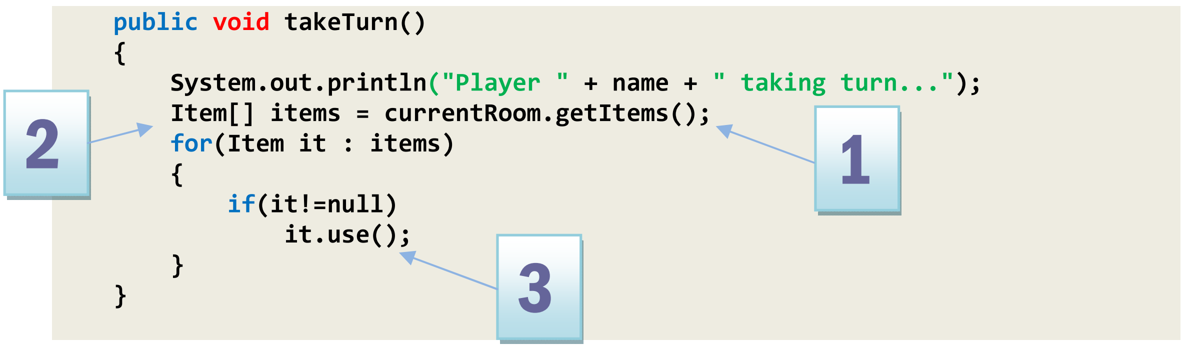
CODE PATTERN: ***"USES-A"***

Problem: how do you implement a "uses-a" relationship, where an object needs to send a message to another object with which it has no ongoing relationship.

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.

This type of association is a bit like booking a holiday through a travel agent. You **use** the agent to making the booking, and you then own that booking – however, you have no link to the agent once the booking has been made.

The association is implemented in the takeTurn method of Player:



```
    public void takeTurn()
    {
        System.out.println("Player " + name + " taking turn...");
        Item[] items = currentRoom.getItems();
        for(Item it : items)
        {
            if(it!=null)
                it.use();
        }
    }
```

Note the following features of this method:

1. A message is sent to the currentRoom object, calling its getItems method
2. An array of Item objects is obtained as a result – this is a local variable, and each Item in the array is itself accessed individually as a local variable.
3. Each Item object is used in turn, by calling its use method

The enhanced for loop

Note in the above code the **simplified version of the for loop** which is useful for stepping through all the elements in an array.

```
        for(Item it : items)
        {
            // do something with the loop variable, it
        }
```

In this code, items is the array, and the loop steps through the elements of the array. The loop variable, it, always refers to the current element of the array each time round the loop. The loop variable type is the same as the type of the data stored in the array, Item in this case. There is no need to specify an end condition or to initialise and increment a counter variable

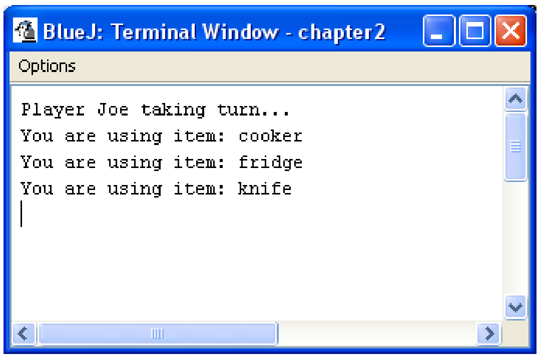
Testing the Player class

You can test the Player class as follows:

- Create a Room object
- Create some Item objects and add these to the Room you have created
- (these could be the same Room and Item objects as we used to test Room)
- Create a Player object
- Call the setCurrentRoom method of the Player, providing the Room you have created as its parameter

- Call the `takeTurn` method of the `Player` object

The output should be something like the following. Look at the code for `Item` and `Player` to see if you can work out which method produces each line of output.



```
Player Joe taking turn...
You are using item: cooker
You are using item: fridge
You are using item: knife
```

NOTE

The complete code for the classes we have created so far can be downloaded from your course website.

What's next?

In the next chapter we will use Java library classes to improve the implementation of `Room`, and make it possible for rooms to be connected together to make a "world"