9. DEVELOPING, TESTING AND DOCUMENTING CLASSES

Increment 2: Interaction between rooms and items

In this increment we will develop the GCU adventure game further to meet the following additional requirement:

- *The player should be able to use any items which are located in a room*

To do this we need to create Java code to implement the `Room` and `Item` classes and make it possible for a room to contain items. The link between a room and its items is similar in some ways to that between the game and the player. Each room can hold several items, and there is a **"has-a"** relationship between `Room` and `Item`.

However, the main difference is that while a game has a single player, a room can have several items.

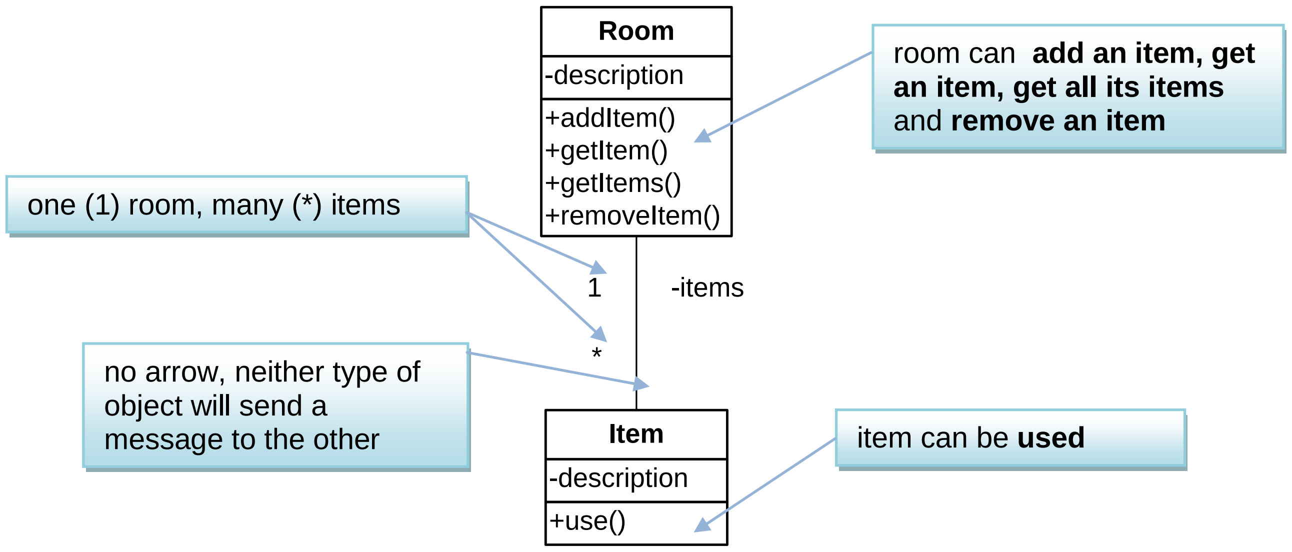
We also need to note that:

- 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 `Player` object has a behaviour `takeTurn`. The `Game` object sends messages to the `Player` object to tell it to perform this behaviour. 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 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 testing this class for yourself in BlueJ in the same way as you tested the `Player` class previously. You can find the classes for this increment in the *adventure* project for lecture 8 on GCULearn.

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" pattern that 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 whose type is the name of the other class."

However, as you have seen, the relationship between `Room` and `Item` is a little different as it is a one-to-many relationship. So how do we implement this? We could have several fields, each of type `Item`. This might not be a very good solution, however:

- If we allow a large number of items in a room, then we would need many similar fields in the `Room` class
- What if we decide later to change the number of items allowed – then we would have to add a new field or fields to the class

This is a situation where we want to work with many objects of the same type. As you saw in Lecture 6, **arrays** are useful when we want to do this. With an array, we can have **one field** that refers to an array of Item objects. If we want to change the number of items allowed we just have to **change the size of the array**.

Using an array is a common solution when a "has-a" relationship is also a **one-to-many** relationship. This leads to a new pattern that is really just a slight variation on the one you have seen 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.

Room class implementation

The outline of the `Room` class looks like this. Note that The `Room` class has a field `items` which is an array of objects of type `Item`.

```
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)
    {
         ...
    }
}
```

Defining the number of items allowed in a room

We need to specify a size for the array, which is the **constant** `MAX_ITEMS`. Like a variable, a constant allows you to specify name that refers to a value, but unlike a variable its value will **never change** while the program is running. By convention constants are given names that are all capitals.

In this code `MAX_ITEMS` is assigned the value 20, so that a room can contain at most 20 items. Note that the line of code that defines `MAX_ITEMS` contains a couple of unfamiliar key words which we will look at in more detail shortly.

It is useful to use a named constant instead of the value itself in your code because you can assign a value to the constant once in class and refer to it by name anywhere it is used in the class code. That means that if you want to change the code to use a different value you only need to make the change once, and don't have to change every place in the code that the value is used. In the `Room` class, if you want to change the design so that more items can be contained all you need to do is change the value assigned to `MAX_ITEMS`.

So, we have specified the maximum number of items in a room using the array size. However, rooms will not all have the same number of items. The array may hold any number of objects up to the maximum, so we also need to keep track of the **number of items that have been added,** using the field `numberOfItems`. This will let us make sure that when we add a new item, it goes into the right place in the array.

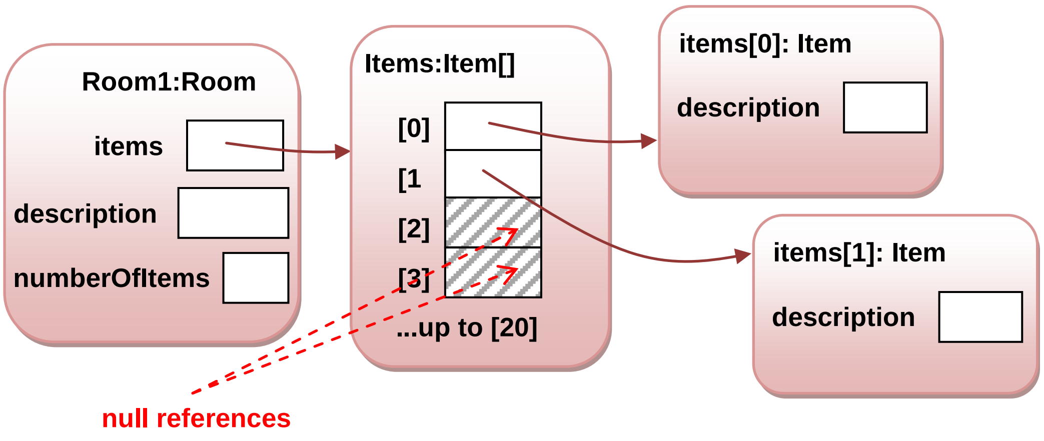
Null references

Note that the array `items` is an array of objects. The array is created in the constructor of `Room`. However, the constructor does not actually create any `Item` objects. The array when it is first created contains **null references**. A null reference is a reference that can refer to an object but no object has been assigned to it yet. It points to "nothing". So, each element in the items array can point to an `Item` object, but no `Item` objects have been created yet.

Instead of being created when the room is created, the items can be created later and added to the room. When the first Item is added, the first element of the `items` array will point to it, while the remaining elements are still null references.

What objects do we have?

An object diagram for a room and its items might look like this (the array contains two items here and the remaining array elements are null references). Note that the `items` field refers to the array, which itself is an object that contains references to `Item` objects.



Static, final and constants

Look at this line of code from the `Room` class that defines the value of `MAX_ITEMS`. `MAX_ITEMS` is a constant, and this is how constants are typically defined in Java:

```
    private static final int MAX_ITEMS = 20;
```

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 and the static key word

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

```
public class MyClass {
     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 (assuming it is not declared private) you need to have an instance of the class, i.e. an object.

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

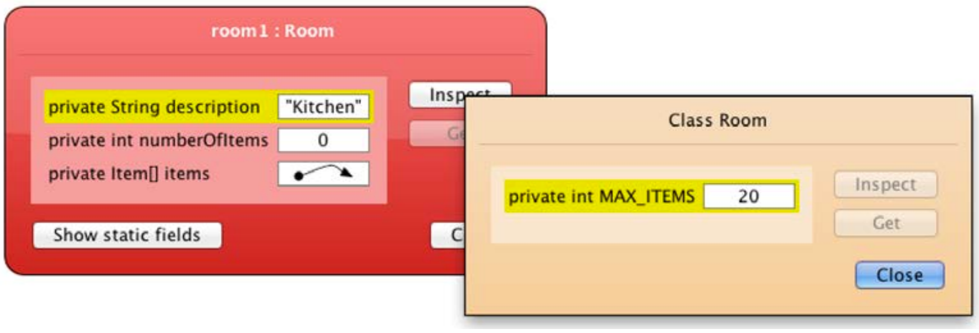
If a variable is declared with the **static** keyword, it is a **class variable**. There is one copy of each class variable **shared between all instances** of the class.

```
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;
```

In BlueJ you can inspect class variables but you need to click the **Show static fields** button in the object inspector window to see them. This is what you can see when inspecting a `Room` object:



private String description "Kitchen"
private int numberOfItems 0
private Item[] items
private int MAX_ITEMS 20

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

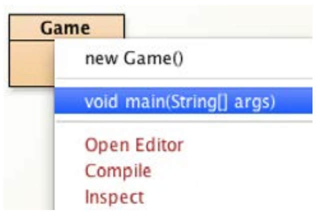
Class methods, on the other hand, cannot access the instance variables declared within the class (unless they create a new object and access them through the object). Class methods can access class variables.

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

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

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

In BlueJ you can call a class method from the class in the main window area without having to have an object on the object bench. Note that you have already been using a (rather special) class method – the **main** method is always defined as a class method in one of the classes in a project:



Class methods provide a way of providing specific functionality **without the need to create an object**. One common usage of class methods is as factory methods, which can allow objects to be instantiated without using constructors.

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

Constants and the final key word

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;
    ...
}
```

Class constants

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

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

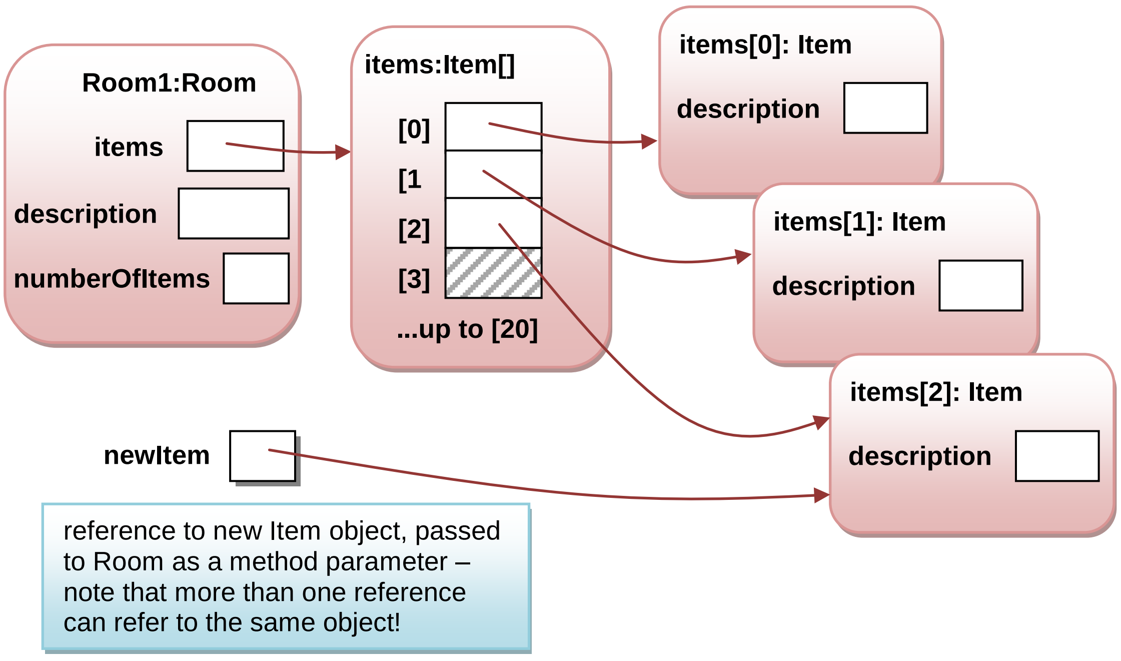
We often use the combination **static final** to define a **class constant.**

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.

```
  public void addItem(Item newItem)
  {
        if(numberOfItems < MAX_ITEMS)  // check array 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.

Designing a search algorithm

As you learned previously, 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. The search 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
```

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** that will signal 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
```

As you have seen earlier, 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 try to write any more code, we really need to do some testing of the `Room` class to check that the methods really do what they are supposed to do.

[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

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.

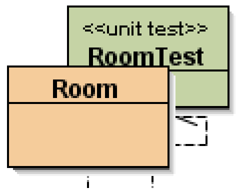
So far, the `Room` class has methods to add an item and to get a specified item. Here are some test cases that we should run. If any of these actions do not produce the desired result, we will need to revise the code and try again.

- 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
- 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 popular unit 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;
...
```

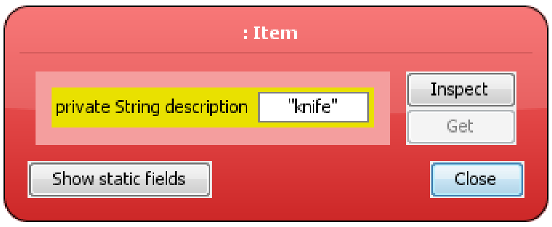
```
    protected void setUp()
    {
            room1 = new Room("kitchen");
            item1 = new Item("cooker");
            item2 = new Item("fridge");
            item3 = new Item("knife");
            room1.addItem(item1);
            room1.addItem(item2);
            room1.addItem(item3);
 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

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 that is returned in the Method Result dialog.



private String description "knife"

Wait a minute! That's the wrong item. Our `Room` class has **failed this test** rather 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

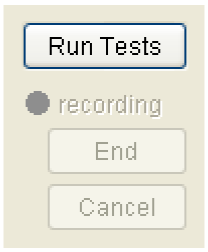
The `RoomTest` class we have created helps us by setting up the same test objects in the object bench each time we want to run a test. The actual test was done manually. If we want to do a lot of tests, this can become very time-consuming. It is often useful to have a series of tests which are run automatically.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

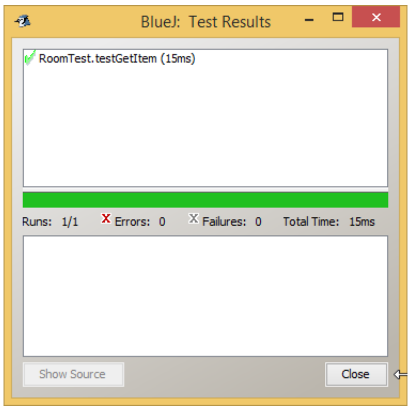
```
    public void testGetItem()
    {
        Item target = room1.getItem("fridge");
        assertEquals("fridge", target.getDescription());
    }
```

`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**.



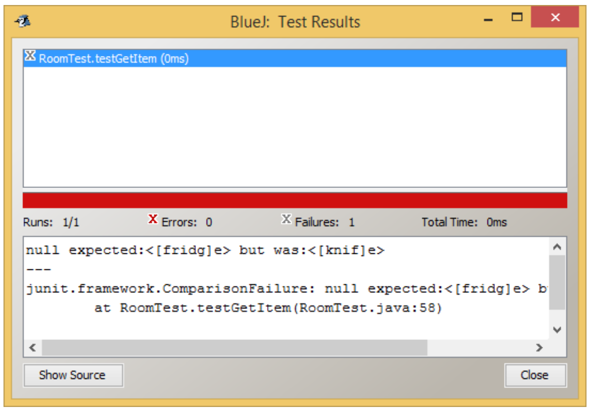
At the moment we only have one test class with one test method. If we click Run Tests, the `testGetItem` test method will run, and the result is shown in the Test Results window. The solid bar is coloured green to indicate a pass, and changes to red if any test does not pass.



RoomTest.testGetItem (15ms)

Runs: 1/1 Errors: 0 Failures: 0 Total Time: 15ms

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:



RoomTest.testGetItem (0ms)

Runs: 1/1 Errors: 0 Failures: 1 Total Time: 0ms

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

Exceptions

Testing can assure you that your code works correctly under the conditions that you test for. However, unexpected things can sometimes happen while a program is running. Code that normally runs with no problem can cause the program to terminate unexpectedly, or **crash**, under certain **exceptional** circumstances which you didn't anticipate when writing or testing the code. Crashes can frustrate users or, more seriously, cause them to lose work or data.

How does Java respond to exceptional circumstances? As an experiment, open the Code Pad in BlueJ and enter the following lines:

```
int x = 0;
1/x
```

The second line evaluates the expression `1/x`, which is fine for any value of `x` except zero. Unfortunately, in this case `x` is in fact zero, so the expression can't be evaluated. You will see a message like this:

```
Exception: java.lang.ArithmeticException (/ by zero)
```

The Java platform has detected an attempt to divide by zero and has **thrown an exception** as a result. An exception is an object that contains information about the error condition that has occurred. Java can detect many different error conditions, and can throw the right type of exception in each case. This one is an `ArithmeticException`: if you try to read beyond the end of an array it will throw an `IndexOutOfBoundsException`; if a disk failure occurs while reading from a file it will throw an `IOException`; and so on.

By default, a program will terminate (i.e. it will crash) when an exception is thrown as it runs. However, the purpose of exceptions is to give a chance to deal with the error condition and allow the program to continue to run. You can write code to **catch an exception** and take some appropriate action, such as informing the user what has happened. If you have a section of code in which an error condition may occur you can enclose it in a **try-catch statement**, which has the simplest form:

```
try {
     statement(s)
} catch (exceptiontype name) {
     statements(s)
}
```

For example, the following version of the code inside the `addItem` method in the `Room` class doesn't check whether the items array is full. The version you saw earlier makes sure the next array position to be filled, `numberOfItems`, is not greater than the array size `MAX_ITEMS`. Instead, here the code that adds the item in a specified position is inside a **try block**. If the user tries to add an item when the array is full `numberOfItems`, will be larger than the maximum index of the array and an `IndexOutOfBoundsException` will be thrown. However, the program will not crash, as this exception is caught and the code in the **catch block** is run – it just prints a message to the terminal.

```
try
{
     items[numberOfItems] = newItem; // error may happen
     numberOfItems++;
}
catch(IndexOutOfBoundsException e)
{
     System.out.println("Cannot add
          at position " + e.getMessage());
}
```

In this case you could deal with the possibility of adding too many items in two different ways: either by checking before adding is allowed or by always allowing adding and catching an exception if too many items are added. In some situations, such as reading from a file, it is impossible to check or predict errors and exceptions must be used. What about dividing by zero – should you use a try-catch statement where a calculation might in some circumstances divide by zero?

It is important to note that exceptions are not just about preventing the program crashing. They are very useful for providing a consistent way of **making the user aware** of unexpected situations that arise as the program runs.

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 **/****

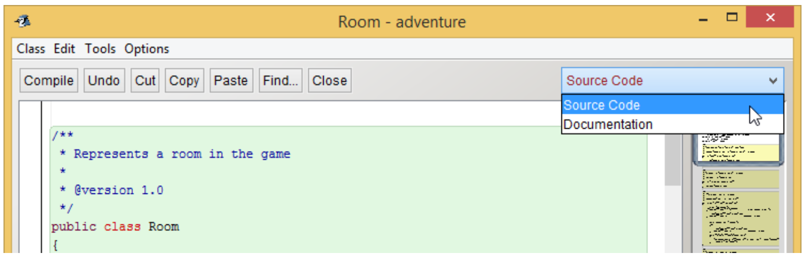
```
/**
 * Returns the item with a specified description
 *
 * @param description description of the specified item
 * @return  the specified item
 */
public Item getItem(String description)
{
     int i = 0;
     boolean found = false; // flag to indicate item found
     Item target = null;

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

parameters and return value

code comments – explain how code works but not part of documentation

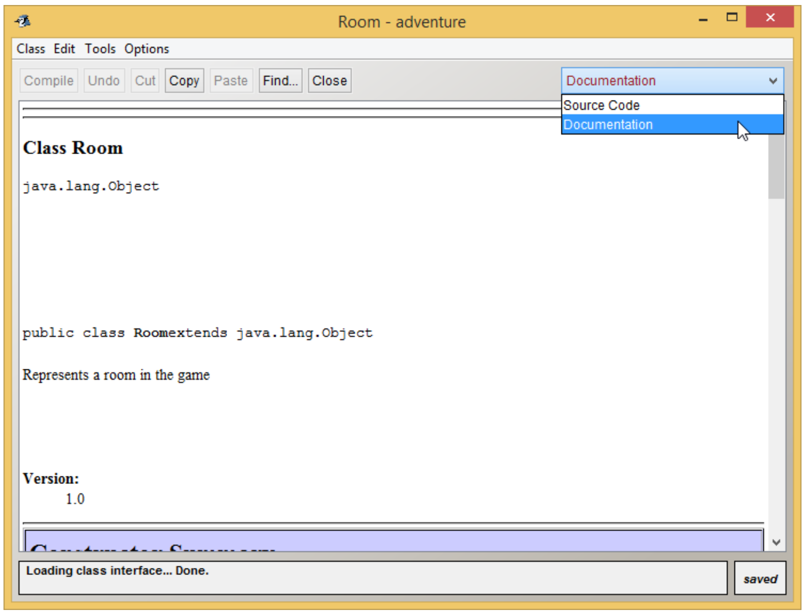
Viewing documentation

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
 *
 * @version 1.0
 */
public class Room
{
```

BlueJ then shows the HTML documentation page for the class.



Class Room

```
java.lang.Object
```

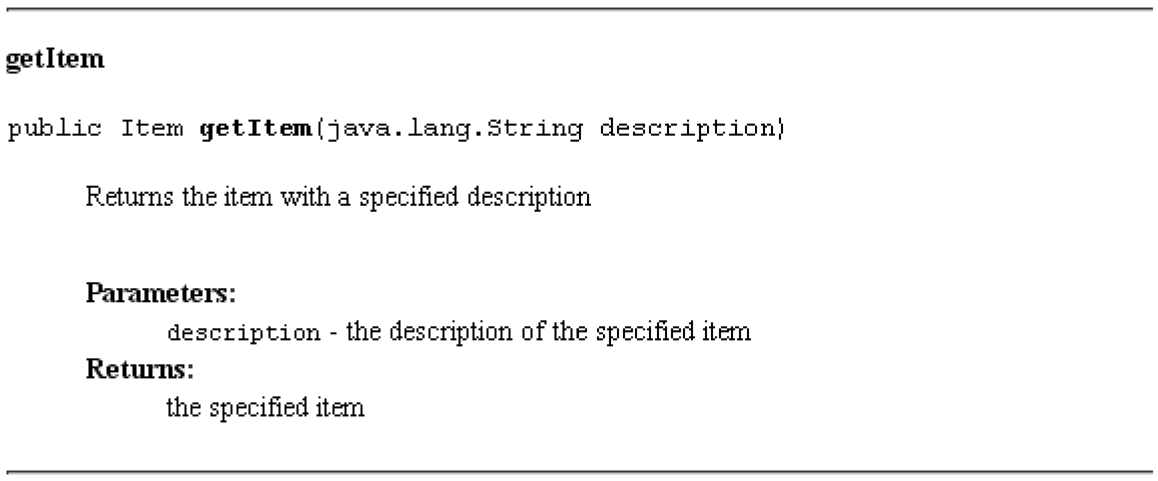
```
public class Roomextends java.lang.Object
```

Represents a room in the game

Version:
1.0

Loading class interface... Done.

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.

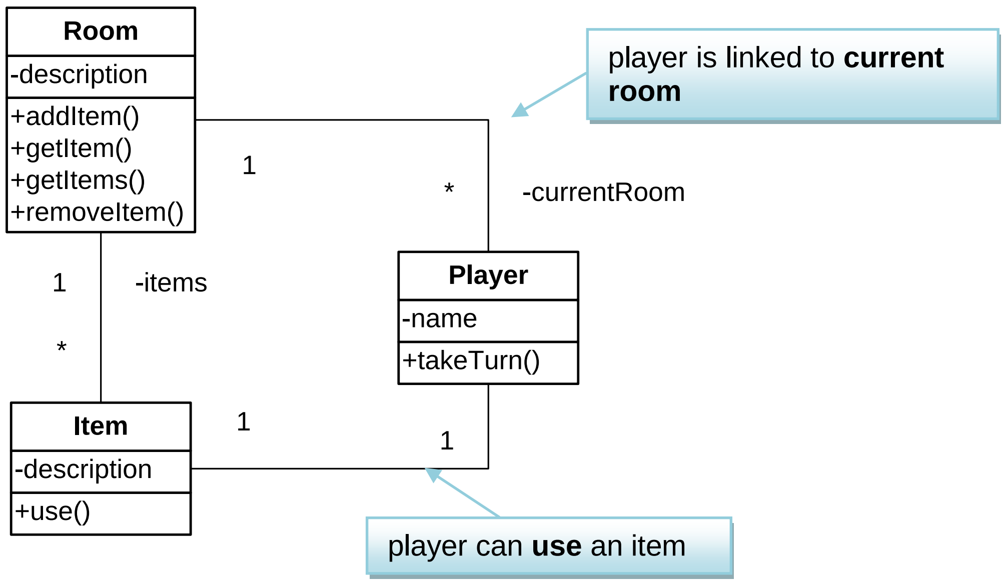
To use the `Room` class it is only necessary to know that it has a `getItem` method which takes a description as a parameter and returns an `Item` – it is not necessary to know how the `getItem` method actually does its job. This is a bit like driving a car – you need to know, for example, that it has 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:

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:

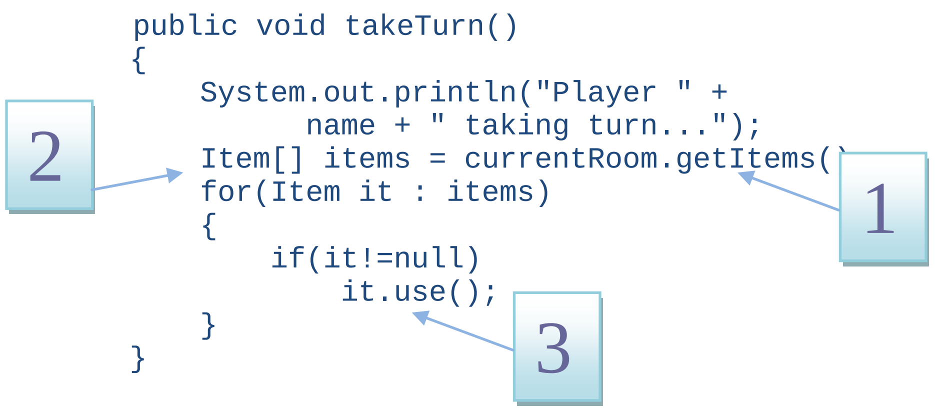
CODE PATTERN: ***"USES-A"***

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
}
```

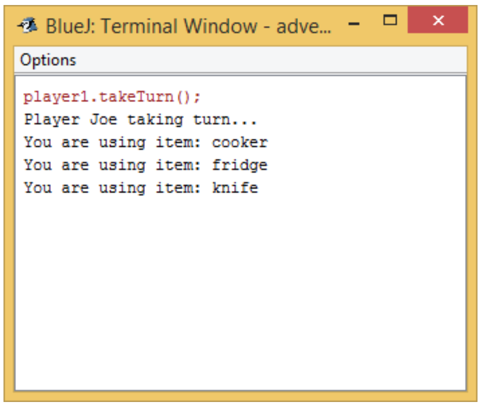
In this code, `items` is the array, and the loop steps through the elements of the array. The loop variable, `it`, always refers to the current element of the array each time round the loop. The loop variable type is the same as the type of the data stored in the array, `Item` in this case. There is 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.



```
player1.takeTurn();
Player Joe taking turn...
You are using item: cooker
You are using item: fridge
You are using item: knife
```

Wrap up

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

One-to-many “has-a” relationship, Constants, Linear search, Unit testing, Exceptions, Class documentation, “uses-a” relationship, Enhanced for loop

In the next lecture you will see the third increment of the game. 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”