Introduction To Programming
With Greenfoot

Object-Oriented Programming in Java
With Games and Simulations
Ch 2

Michael Kölling
KENT
UNIVERSITY OF KENT
2.1 The Little Crab scenario

In the previous chapter, we have discussed how to use existing Greenfoot scenarios: We have created objects, invoked methods, and played a game. Now, we want to start to make our own game.
Start Greenfoot

Scenario Open...

Q

Assignments
Mr. Kadri
Greenfoot
Scenarios

Select the ‘crab’ scenario from the Greenfoot sample scenarios.
Create a crab in the same way we created a wombat, place it in the world, and click **run**. Nothing will happen; we now need to program the crab to move.
On the right, you see the classes in this scenario. We notice that there is the usual Greenfoot Actor class, a class called Animal, and the Crab class.

The hierarchy (denoted by the arrows) indicates an is-a relationship (also called inheritance):

A crab is an animal, and an animal is an actor. (It follows then, that a crab also is an actor.)

Initially, we will work only with the Crab class. We will talk a little more about the Actor and Animal classes later on.

The crab does not do anything when Greenfoot runs. This is because there is no program code in the definition of the Crab class that specifies what the crab should do.
2.2 Make the crab move

We will now make the crab do something.

Open the editor for the crab by selecting *Open editor* from its menu.

In the editor, you will see an empty *act* method:

```java
import greenfoot.*; // (World, Actor, GreenfootImage, and Greenfoot)
/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Animal
{
    public void act()
    {
        // Add your action code here.
    }
}
```
import greenfoot.*; // (World, Actor, GreenfootImage, and Greenfoot)
/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Animal {
    public void act()
    {
        // Add your action code here.
    }
}

This is a standard Java class definition. It defines what a Crab can do.

Within this class definition, we can see what is called the act method.
Here, we can add some code that determines the actions of the crab. We can replace the grey text in the middle with a command. One such command is

```java
move();
```

The act method should then look like this:

```java
public void act()
{
    move();
}
```

**Note** that it has to be written exactly as shown, including the parentheses and the semicolon.

The instruction `move()` is called a method call.

A **method** is an action that an object knows how to do (here, the object is the crab) and a method call is an instruction telling the crab to do it.

The parentheses are part of the method call. Instructions like this are ended with a semicolon.
Close the editor and click the **Compile all** button.

**Exercise 2.1** Place a crab into the world. Try clicking the **Act** button.

Also click **Run** and play with the **Speed** slider.

**Exercise 2.2** Place multiple crabs into the world. Run the scenario. What do you observe?
2.3 Turning

The crab also understands a *turn* instruction. Here is what it looks like:

\[
\text{turn}(5); \\
\]

The number 5 in the instruction specifies how many degrees the crab should turn. This is called a *parameter*.

We could also use other numbers, for example:

\[
\text{turn}(23); \\
\]

The degree value is specified out of 360 degrees, so any value between 0 and 359 can be used. (Turning 360 would turn all the way around, so it is the same as turning 0 degrees, or not turning at all.)

How do we turn left?
Use a –ve angle eg: turn(-5)
If we want to turn instead of moving, we can replace the `move()` instruction with a `turn(5)` instruction. The act method then looks like this:

```java
public void act()
{
    //move();
    turn(5);
}
```

**Exercise 2.3** Replace `move()` with `turn(5)` in your scenario. Try it out. Also, try values other than 5 and see what it look like. Remember: every time after you change your source code, you must compile again.
The next thing we can try is to move and turn together. The act method can hold more than one instruction – we can just write multiple instructions in a row.

```java
import greenfoot.*; // (World, Actor, GreenfootImage, and Greenfoot)
/**
 * This class defines a crab. Crabs live on the beach.
 */
public class Crab extends Animal
{
    public void act()
    {
        move();
        turn(5);
    }
}
```

**Exercise 2.4** Try it out: use a `move()` and `turn(N)` instruction in your crab’s act method. Try different values for N.
The number between the parentheses in the turn instruction – i.e. the 5 in turn(5) – is called a **parameter**. A parameter is an additional bit of information that we have to provide when we call some methods.

Some methods, like `move`, expect no parameters. They are happy to just execute as soon as we write the `move()` instruction. Other methods, such as `turn`, want more information: *How much should I turn?* In this case, we have to provide that information in the form of a parameter value between the parentheses, for instance `turn(17)`.

Compile, place the crab in the world again, and try it out (click *Run*).

Try placing multiple crabs.
When we write program code, we have to be very careful: every single character counts. Getting one small thing wrong will result in our program not working. Usually, it will not compile.

If, for example, we forget to write the semicolon after the move() instruction, we will be told about it when we try to compile.

**Exercise 2.5** Open your editor to show the crab’s source code, and remove the semicolon after move(). Then compile. Also experiment with other errors, such as misspelling move or making other random changes to the code. Make sure to change it all back after this exercise.

As we can see with this exercise, if we get one small detail wrong, Greenfoot will open the editor, highlight a line, and display a message at the bottom of the editor window.
2.4 Dealing with screen edges

When we made the crabs move and turn in the previous sections, they got stuck when they reached the edge of the screen.

Now, we shall improve this behavior so that the crab notices that it has reached the world edge and turns around. The question is: How can we do that?
Reminder

The move and turn methods come from the Animal class.

A crab is an animal (signified by the arrow that goes from Crab to Animal in the class diagram), therefore it can do whatever an animal can do.

Our Animal class knows how to move and turn – that is why our crab can also do it. This is called inheritance:

The Crab class inherits all the abilities (methods) from the Animal class.

A subclass inherits all methods from a superclass. That means that it has and can use all methods that its superclass defines.

The question now is: What else can our animals do?
To investigate this, we can open the editor for the Animal class. The editor can display two different views: It can show the source code (as we have seen for the Crab class) or it can show the documentation. The view can be switched using a popup selection menu in the top right corner of the editor window. We now want to look at the Animal class in the documentation view.
Exercise 2.7 Open the editor for the Animal class. Switch to documentation view. Find the list of methods for this class (the “Method Summary”). How many methods does this class have?
## Constructor Summary

**Animal()**
Constructor for Animal - nothing to do.

## Method Summary

**void act()**
Act - empty method.

**boolean atWorldEdge()**
Test if we are close to one of the edges of the world.

**boolean canSee(java.lang.class cls)**
Return true if we can see an object of class 'cls' right where we are.

**void eat(java.lang.class cls)**
Try to eat an object of class 'cls'.

**void move()**
Move forward in the current direction.

**void turn(int angle)**
Turn 'angle' degrees towards the right (clockwise).

### Methods inherited from class greenfoot.Actor

- addedToWorld, getHeight, getImage, getIntersectingObjects, getNeighbours, `getOngoingGesture`, getOneIntersectingObject, getOneObjectOffset, `getImageOffset`, `getWorldSize`, `getTimeOffset`, `getWorldScale`
Method signature

If we look at the method summary, we can see all the methods that the Animal class provides.

Among them are three methods that are especially interesting to us at the moment.

- boolean `atWorldEdge()`
  Test if we are close to one of the edges of the world.

- void `move()`
  Move forward in the current direction.

- void `turn(int angle)`
  Turn 'angle' degrees towards the right (clockwise).

Here, we see three signatures of methods.

Each method signature starts with a return type, and is followed by the method name and the parameter list.
The move and turn methods are the ones we have used in the previous sections. If we look at their parameter definitions, we can see what we observed before: move has no parameters (the parentheses are empty), and turn expects one parameter of type `int` (a whole number) for the angle.

We can also see that both methods have `void` as their return type. This means that they both do not return any values.

```
boolean atWorldEdge()
    Test if we are close to one of the edges of the world.

void move()
    Move forward in the current direction.

void turn(int angle)
    Turn 'angle' degrees towards the right (clockwise).
```
The signature for `atWorldEdge` is a little different. It is

```java
boolean atWorldEdge()
```

This method has no parameters (there is nothing between the parentheses), but it specifies a return value: `boolean`. It is a type that can hold two possible values: `true` or `false`.

Calling methods that have return values (where the return type is not `void`) is like asking a question.

If we use the `atWorldEdge()` method, the method will respond with either `true` (Yes!) or `false` (No!).

Thus, we can use this method to check whether we are at the edge of the world.
Exercise 2.8 Create a crab. Right-click it, and find the boolean `atWorldEdge()` method (it is in the “inherited from Animal” submenu, since the crab inherited this method from the Animal class). Call this method. What does it return?

Exercise 2.9 Let the crab run to the edge of the screen (or move it there manually), and then call the `atWorldEdge()` method again. What does it return now?
if-statement

The if-statement is part of the Java language that makes it possible to execute commands only if some condition is true. For example, here we want to turn only if we are near the edge of the world.

The general form of an if-statement is this:

```java
if ( condition )
{
    instruction;
    instruction;
    ...
}
```

In place of the `condition` can be any expression that is either true or false (such as our `atWorldEdge()` method call), and the `instructions` will only be executed if the condition is true. There can be one or more instructions.

If the condition is false, the instructions are just skipped, and execution continues under the closing curly bracket of the if-statement.
public void act() {
    if (atWorldEdge()) {
        turn(17);
    }
    move();
}

Using `atWorldEdge()` , we can modify our code to turn when we reach the world edge.

We can now combine this method with an `if-statement` to write the code shown above.

Note that our move() method call is outside the if-statement, so it will be executed in any case. In other words: If we are at the edge of the world, we turn and then move; if we are not at the edge of the world, we just move.

**Exercise 2.10** Try it out! Type in the code discussed above, and see if you can make your crabs turn at the edge of the screen. Pay close attention to the opening and closing brackets – it happens easily to miss one or have one too many.
Concept summary

- A **method call** is an instruction that tells an object to perform an action. The action is defined by a method of the object.

- Additional information can be passed to some methods within the parentheses. The value passed is called a **parameter**.

- When a class is compiled, a first check for errors is done. If an error is found, an **error message** is displayed.

- A subclass **inherits** all methods from a superclass. That means that it has and can use all methods that its superclass defines.

- Calling methods with **void return types** issues a command. Calling methods with **non-void return types** asks a question.

- An **if-statement** can be used to write instructions that are only executed when a certain condition is true.