*Introduction to Java via Greenfoot*

*Aquarium Lab*

# Main Objectives

* Become familiar with the Greenfoot IDE.
* Discover what methods are and learn how to use them.
* Learn the basics of writing conditionals.
* Learn the basics of primitive variables and fields.
* Learn how to create object variables.

# Prior Knowledge

This lab is intended to help students start with Java. No prior knowledge is needed, but teacher guidance is always welcome! A lot of (learning) mistakes occur on the first days of students writing code.

# The Greenfoot IDE

Greenfoot is an integrated development environment designed to help teach the object orientated nature of Java in an intuitive, visual, and interactive format. Students will use standard Java code but will learn to use APIs that have streamlined the visual, audio, and interactive elements.

# Getting Started

Install the Greenfoot IDE: <https://www.greenfoot.org/download>

* Greenfoot went through many changes beginning with the 3.0.0 release. Version 2.4.2, although older, tends to be one of the most stable versions but is missing a few of the newer additions.

In your Aquarium folder, run the project file with the Greenfoot icon to begin.

# Demonstration

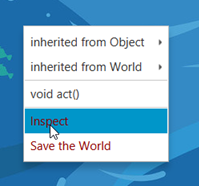
View a sample demonstration of what the Aquarium lab could look like here:

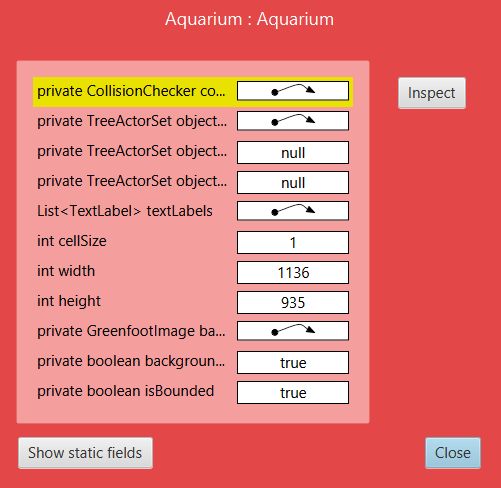
<https://www.greenfoot.org/scenarios/23659>

# Explore

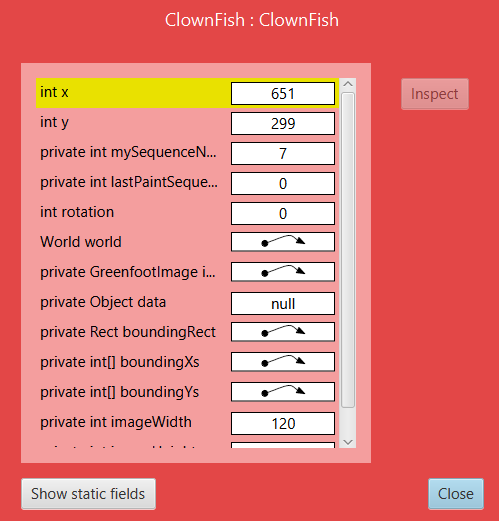
There are several elements on the screen for you to explore. The main left area has a picture of the current World and the current Actors on the World. At the moment, no Actors are on the World itself. On the right side are boxes showing the ‘classes’, which will contain the code for the various elements (classes) of your project. You will be writing code for the Aquarium-World itself, code for the ClownFish-Actor, and the StarFish-Actor, as well as many other Actors that will be introduced later. Finally there are some buttons on the bottom that you can feel free to play with, but you will see what they do later.

Right click the world (Aquarium picture, not the class) and notice a menu of things it can already do. Try the “inspect” option to see a multitude of information that the World knows about its current ‘state’, most of which are not helpful at the moment, but you should be able to identify how wide and tall the World is. Do these numbers make sense to you?





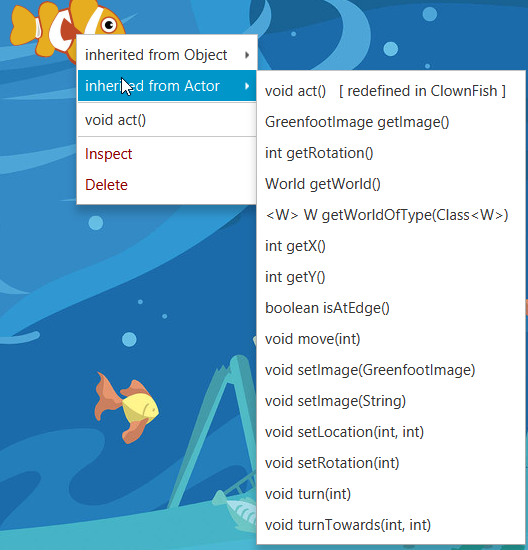
Right click either ClownFish or StarFish class on the right and select the option ‘new ClownFish()’ or ‘new StarFish()’. The word ‘new’ builds objects and you will notice. You can drop your fish on the screen somewhere and right-click it to display a menu of options about the fish. Try inspecting it to see information the fish (actor) knows about itself already, pay special attention to the ‘x’ and ‘y’ coordinates of your Fish. Try dragging your Fish around and checking the (x,y) coordinate of it.



## Coordinate Questions

1. In math, the ‘origin’ is at (0,0). Where is (0,0) in Greenfoot?
2. How big is this current world?
3. What are the coordinates of the four corners of the screen?
4. If the screen was 5000 wide and 4000 tall, what would the bottom right coordinate be?
5. What is the major difference between the coordinates in a math class and those on Greenfoot?

Right click your Fish and look at the menu ‘Inherited from Actor’. This shows a list of all ‘methods’ your Fish already knows simply because it is a ‘subclass’ of Actor. In Greenfoot, most classes, like ClownFish or StarFish, will either be a subclass of World or Actor, depending on if they are the background itself or items that move around on top of the background. Play around with the methods.



The ones you should become the most familiar with early are: getRotation, getX, getY, isAtEdge, move, setLocation, setRotation, turn, and turnTowards. Try each of these methods multiple times until you understand what they do. These methods will be your primary way to make the actors interact on the screen. The questions below are helpful in pinpointing some key concepts before moving further into Greenfoot. They establish some ideas of how directions and movement will work, while also highlighting some concepts of methods that are important to know before moving further.

## Actor Method Questions

1. Some of the methods have the word ‘int’ either before the name of the method or in the ( )s. What is int likely short for?
2. Some of the methods cause a pop up to appear that allows you to type in information. By looking at the list of methods above, how can you easily identify if you will need to type information in or not when you try to use a method?
3. When using the move method, what happens if you give it a positive number versus a negative number?
4. Very specifically, which direction does move with the number 50 make you go? Are you sure?
5. When using the turn method, what happens if you give it a positive number versus a negative number?
6. What does the number you type in to turn represent?
7. Add at least four total Fish to your screen. Use setRotation to make one face east, another south, another west, another north. What numbers did you use?
8. Tell each of the four Fish from the last question to move 50. Was your earlier answer correct?
9. What specific direction would a Fish face if they were told to setRotation to: 180? 45? -45? -90? 720?
10. Describe the difference between turn, setRotation, and turnTowards.
11. Which method would make sense to rename teleport?
12. Some of the methods cause a pop up to appear after the method has been chosen which shows you information. This is called ‘returning’ information to you. By looking at the list of methods above, how can you easily identify if a method will return information to you?
13. What does the word ‘void’ appear to mean?
14. What does the word ‘boolean’ appear to mean?

# Learning about Methods

Right clicking the screen or the actors to do things would end up being tedious and impractical for most situations. We want to learn how to make the different fish use the methods automatically and use them in such a way that each fish feels like it has its own personality. To do so, we need to learn how to write the code to call a method.

First, look at the anatomy of a few methods as shown when right clicking one of the Fish:

Return type Method Name ( Parameters )

int getY ( )

boolean isAtEdge ( )

void move ( int )

void setLocation ( int, int )

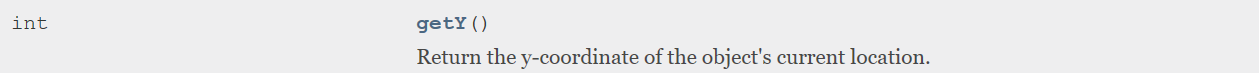
The return type determines what kind of answer the method gives back to you. Void means no answer will be coming back. The method name determines what the method is called. Good programmers will try to name methods intuitively. The parameters determine any information that the method needs to be sent for it to run correctly. Some methods do not need extra information to run correctly, while others do. setLocation, for instance, needs to be told both an X and a Y coordinate to teleport to.

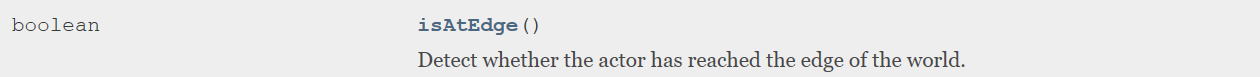
These are just a few methods amongst many, but the format does not change much even in the most complicated situations. Where can we find a list of all possible methods without right clicking the screen? Welcome to the API: <https://www.greenfoot.org/files/javadoc/>

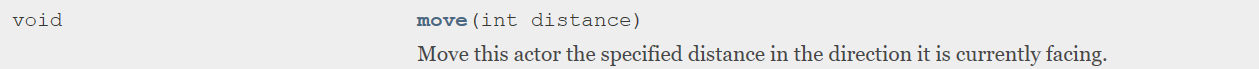
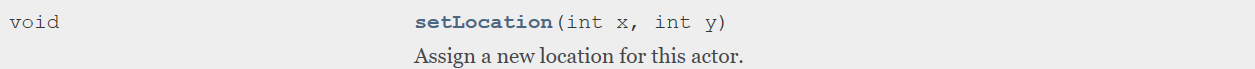
All Java classes (the code / blueprint for things you will learn to write) have an API. Consider the API like a formula-sheet from a math class. You do not need to memorize all of it, you just need to be able to look up the formula and use it when needed. This project will be using methods from the Actor, Greenfoot, and World classes, all of which you will see as sub menus in the API.

You can see the same methods under the Actor API with a bit more information as above:

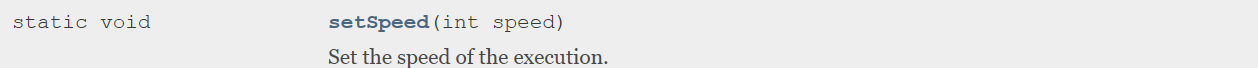
Return type Method Name ( Parameters )





While under the Greenfoot API you would see new methods, for instance:



# Calling Methods

To call (use) a method you will need to decide these four things:

1. Where is the method found?
2. What is it called?
3. Does it need information (arguments) from me?
4. If it returns information, what do I want to do with the answer it gave me?

## Calling your own methods

General syntax for calling methods that your class owns itself or has inherited from its parent class:

*methodName( argument1, argument2, … )*

*or*

*this.methodName( argument1, argument2, … )*

*Examples for ClownFish to use some of its own methods:*

*move( 5 );*

*setRotation( 90 );*

*setLocation( getX(), 0 ); Can you see two different method calls here?*

*or*

*this.move( 5 );*

*this.setRotation( 90 );*

*this.setLocation( this.getX(), 0 ); Can you see two different method calls here?*

## Calling static methods

General syntax for calling methods that are ‘static’ you need to attach the ClassName of where the static method was found in front of the methodName and use a period to separate the two.

*ClassName.methodName( argument1, argument2, … )*

*Examples for ClownFish to use some of Greenfoot’s static methods:*

*Greenfoot.delay( 500 );*

*Greenfoot.stop();*

## Calling another object’s methods

General syntax for calling methods on other objects instead of yourself:

*variableName.methodName( argument1, argument2, … )*

*Examples for ClownFish to use some methods of a StarFish which happens to be named stormy:*

*stormy.move( 5 );*

*stormy.setRotation( 270 );*

## Important Note

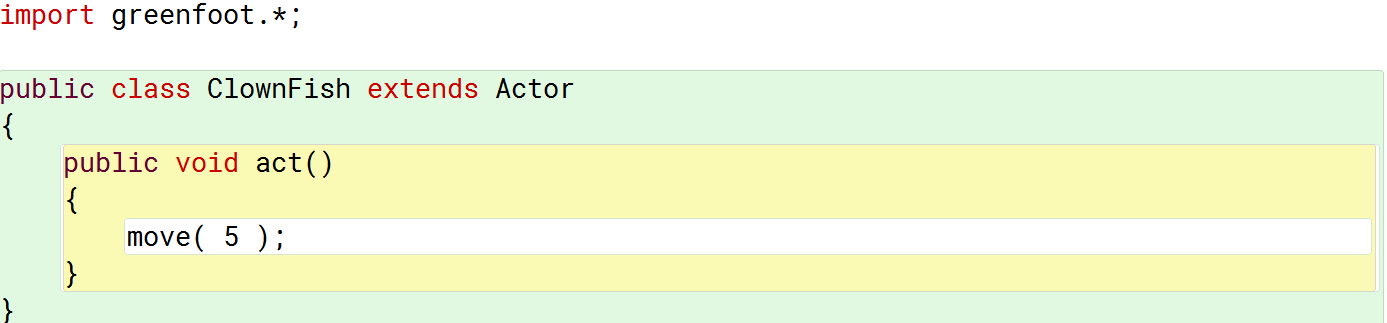
Notice all the example method calls end in with a semicolon (;). Java uses the semicolon to represent the end of a current statement. Alternately, some sections of code use curly braces to represent the beginning and ending of larger (block) statements instead of semicolons.

Now we get to the main part of this lab. Writing our first lines of Java code by calling (using) the methods without having to right click each the slow way.

We will begin with the ClownFish. If you double-click the ClownFish class name on the right, the code will open up. Our code will currently go between the two { }s under the public void act() at the start.

# 

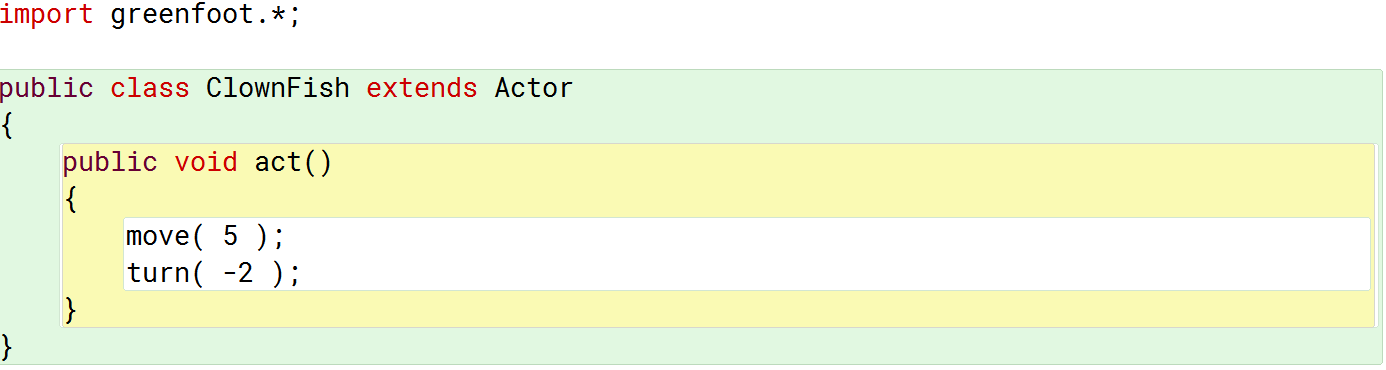
Add a method call in the area shown to tell the ClownFish to move forward 5 pixels.



Return back to the Aquarium program itself and place a ClownFish on the screen. The code we wrote is in the act method, so if you right click the fish and tell it to act it should now move a small amount forward. Build several more ClownFish (hint, hold the shift key down and you can keep building). Right clicking each fish and telling them to act will only move that one fish. But, now you can play with the buttons on the bottom of the screen:



Return to your ClownFish code and try adding below the move( 5 ); a method call for the ClownFish to turn a little bit. Try testing it by placing more ClownFish and clicking Run.



## Method Calling Questions

1. What would the fish do if you added setLocation( 0, 0 ); just above the move(5); call?
2. What would the fish do if you replaced all of the code in the act method with   
   setLocation( getX(), getY() );
3. What would the fish do if you replaced all of the code in the act method with  
   setLocation( getX() – 5, getY() );
4. What line of code could make the fish move towards the top of the screen no matter what direction it was facing?
5. What two lines of code could be used to make the fish face northeast and then move towards that direction?
6. What two lines of code would make the fish face the top left corner of the screen and then move towards that direction?
7. What line of code could make the fish teleport to the same y coordinate it already has but all the way to the left side of the screen?
8. What line of code could be used to duplicate the effect of turn( 5 ); without using the turn method?

# Conditionals

There are many times throughout programming where your software will need to branch its code into different options depending on certain situations. These situations are called conditionals, where some question may come up to be either true or false. Based on the answer to that question, your code will branch to do something different.

The most common conditional is an if statement. An if statement will check a condition to see if it is true, if so it will have a block of code it could then run. The syntax is:

if ( *some condition(s)* )

{

*some code to do if the condition(s) were true*

}

Anything that generates a true or false can be used as the condition. This includes any sort of *boolean* expression or method call, since booleans are true or false values. It also includes any sort of mathematical comparisons which generate trues or falses, which would include concepts like less than, greater than, equal to, not equal to, or various others. The symbols for mathematical comparisions are:

== Are the two values equal?

!= Are the two values not equal?

< Is the first value less than the second?

> Is the first value greater than the second?

<= Is the first value less than or equal to the second?

>= Is the first value greater than or equal to the second?

It is also possible to ask more than one condition at a time by combining two conditions with either an and symbol, &&, or an or symbol, ||.

There are many examples of conditions that can be seen below. Study each to try to grow your understanding of possible questions. Using if statements will be a major part of writing code in Java.

There were several methods pre-built into Greenfoot that had boolean results. Here are some examples of early ones that will help make your fish smarter or more interactive:

if ( getX() < 100 )

{

setLocation( 1100, getY() );

}

*This will determine if the Actor is close (within 100 pixels) of the left side of the World. If so, it will teleport to the right side of the screen with the same Y coordinate.*

if ( isAtEdge() )

{

turn( 180 );

}

*This will determine if the Actor is touching any sides of the World. If so, it will turn and face the opposite direction.*

if ( Greenfoot.isKeyDown( “up” ) )

{  
 move( 1 );

}

This uses one of the static methods Greenfoot knows and determines if the up-arrow on the keyboard is currently pressed down, if so the fish will move forward in its current direction. The word “up” is in quotes because it is a String. You will learn more about Strings the further you go, but for now just think that a typical String will be in quotes and is a word or phrase.

if ( Greenfoot.mousePressed( this ) && isTouching( StarFish.class ) )

{

removeTouching( StarFish.class );

}

This uses one of the static methods Greenfoot knows and determines if someone running the Aquarium simulation put their mouse over the Actor and pressed a mouse button. However, it also uses one of the && symbols, which means not only does someone have to press the mouse button on the Actor, but the Actor also has to be touching a StarFish. If both of those are not true, then nothing will happen. If both of those are true, then the StarFish the Actor is touching will be removed from the screen. Notice the StarFish.class is a new addition and is needed any time the API identifies java.lang.Class<?> cls as a needed argument.

if ( getY() <= 35 || getY() >= 900 )

{

setLocation( 600, 400 );

setRotation( 0 );

}

This uses two different questions and combines the questions with an || (or). If either of the questions are true, the Actor will first teleport to the (600,400) coordinate and then face due east.

One last point before moving on to exercises and letting you try building the Aquarium more in depth. if statements have a secondary, optional part, called else. An else statement allows a block of code to occur if the if-statement was false.

if ( *some condition(s)* )

{

*some code to do if the condition(s) were true*

}

else

{

some code to do if the condition(s) were false

}

Example:

if ( isAtEdge() )

{

turn( 180 );

}

else

{

move( 10 );

}

This has two different outcomes. Either the Actor will turn 180 degrees if it is at the edge of the screen, or it will move forward 10 pixels.

*First Sea Animal: ClownFish.*

Edit the code in your ClownFish to make it accomplish the following results. Each ClownFish should move forward a little bit each act. When the ClownFish is near the edge of the right side of the screen, the ClownFish should teleport to the left side of the screen at the same Y coordinate it already had.

Here is an example visual time lapse of the ClownFish moving and ultimately reappearing on the left side after it hits the right side of the screen:











Try this exercise and test it by putting several different ClownFish on the screen to see if they all work.

*Second Sea Animal: StarFish.*

Edit the code in your StarFish to make it accomplish the following results. Each StarFish should move forward a little bit each act. Each time the StarFish is near any edge of the screen, the StarFish should turn a random amount of degrees.

### New Knowledge: How do you generate a random number?

The APCSA exam requires students to learn Math.random() to generate random numbers. There are many ways to generate random numbers in Java. Greenfoot has a built-in way if you would like to use it, you will find it under the Greenfoot API as one of the static methods. Another way is to use something called the Random class, which you could look up on the Internet yourself. However, if you will be taking the APCSA exam, you may want to learn the Math.random() option. This is a small introduction that should be done more in depth at a later time:

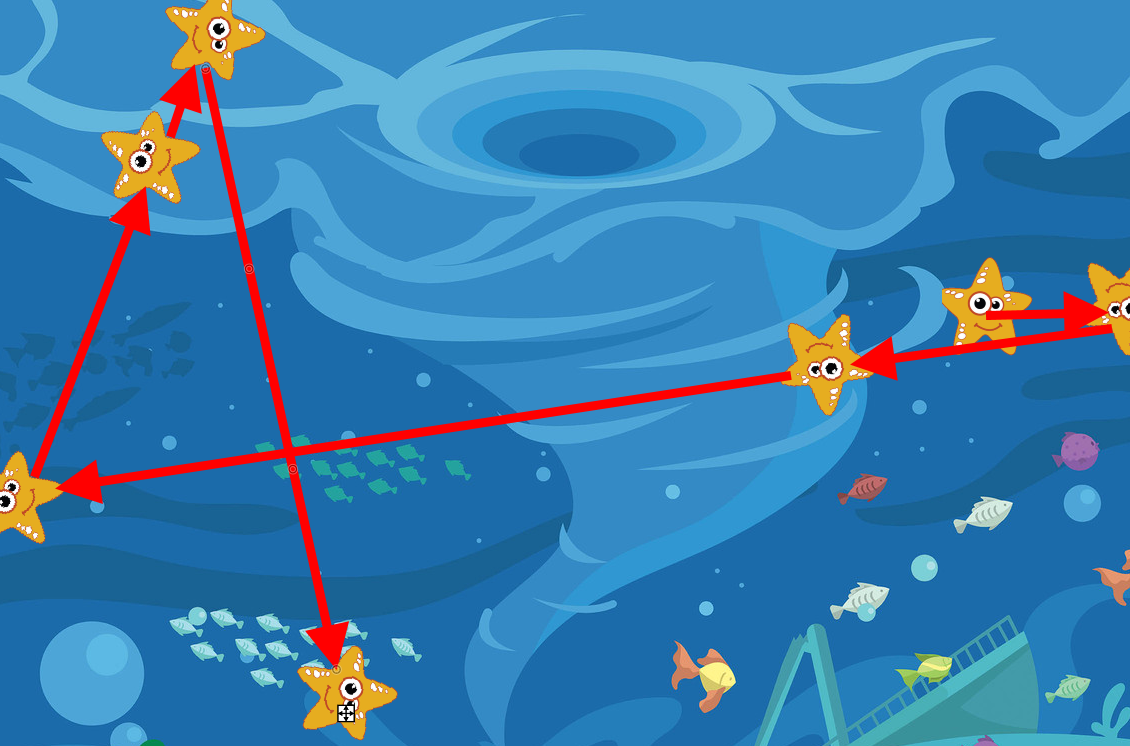
* Math.random() generates a number between [0, 1). These are not integers, they are doubles (real numbers).
* If you want a random integer, Math.random() can be manipulated to give back an integer with the following formula:

(int)( Math.random() \* number of desired results ) + smallest result

* An example usage would be an Actor which wishes to move forward 10-15 pixels randomly.

move( (int)(Math.random() \* 6) + 10 );

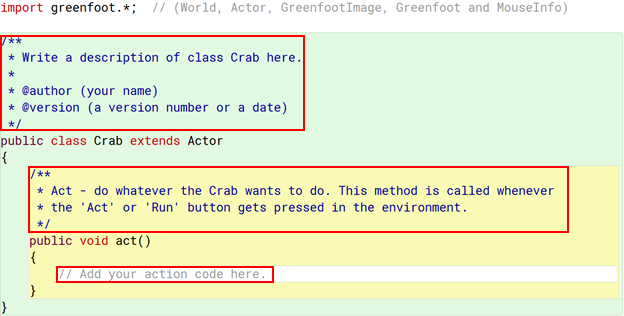
Notice there are 6 numbers from 10-15, {10, 11, 12, 13, 14, 15}, and the smallest desired was 10.



*Third Sea Animal: Crab.*

Our third ‘fish’ will be a Crab instead. The Crab will walk until it hits the edge of the screen then turn around and walk the other way until it hits the edge of the screen, back and forth. This is relatively trivial, and you can attempt it now, but you will likely run into one problem with the way the Crab looks.

From now on you will need to build your own classes from the beginning to the end. The Crab will be another type of Actor. To build another type of Actor, right click the Actor class and choose ‘New subclass’. Type Crab as the name of the new class (we will always uppercase our class names) then select the supplied crab image.



### New Knowledge: Comments

There are three types of ‘comments’ that exist in Java. Comments are special text that programmers add while coding to make notes to themselves or to others who will be using the class in the future. There is a large variety of reasons to write comments, but one thing remains always true – the computer ignores the comments when running your code. Comments are purely for humans.

Three types of comments:

1. Single line comments. Any line of text can be commented out by adding a // in front of it.  
   You can see this in the act method. The // does not have to be the first item in the line of text but everything after it is ignored by the computer.
2. Block comments. If many lines of text are to be commented out, a programmer can add a /\* where the comment is to start and later put a \*/ for where it will end. Anything between the /\* and the \*/ is ignored by the computer.
3. Javadoc comments. These special comments build the APIs for Java classes, the same as the Greenfoot APIs you have already looked at. These start with a /\*\* and end with a \*/ and use special words in the middle to make the API build correctly. These particular comments are not on the APCSA exam and can be learned on their own later. You will see these in two of the highlighted boxes above. Feel free to delete them or edit them as you wish.

### New Knowledge: Variables

Variables are used to store information, either long term or short term, while your program is running. There are an infinite amount of types of information that could be stored in a program, so you will need to think about what your needs on as you design your software. Early on we will be working with a few variables types called ‘primitive variables’. These are the building blocks of more complicated information later.

The main primitive types:

int – Stores integers (whole numbers), both positive and negative.

double – Stores decimals (real numbers), both positive and negative.

boolean – Stores a true or false value.

char – Stores a single typed character (not on the APCSA exam).

Variables can temporarily exist (local variables), or permanently exist (fields). While the variable exists it can be updated to store new information or you can access the current information in the variable.

To make a primitive variable, follow this syntax:

typeOfVariable nameOfVariable;

or

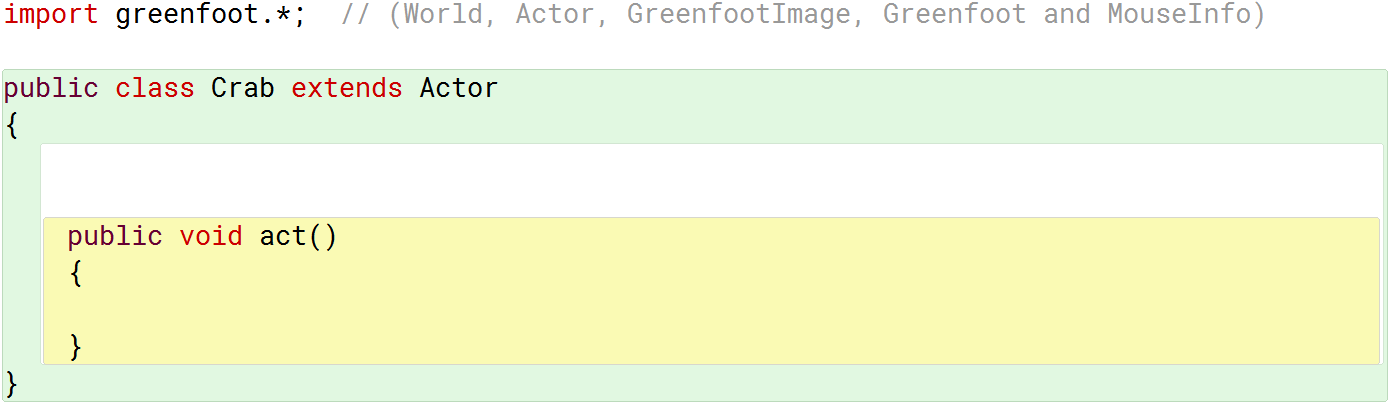
typeOfVariable nameOfVariable = valueOfVariable;

To use the value a variable is storing, just type the name of the variable. To update the value a variable is storing, type the name of the variable, followed by an equal sign, then put the new value on the right side.

Here are some examples:

| int speed = 5;  if ( Greenfoot.isKeyDown( “space” ) )  {  speed = speed + 1;  } | boolean moving = true;  if ( isAtEdge() && moving )  {  moving = false;  } |
| --- | --- |

To make a variable permanent, it must be created as a field. To create a field, make your variable at the top of your class in the area shown, and (for now) put the word private in front of it.



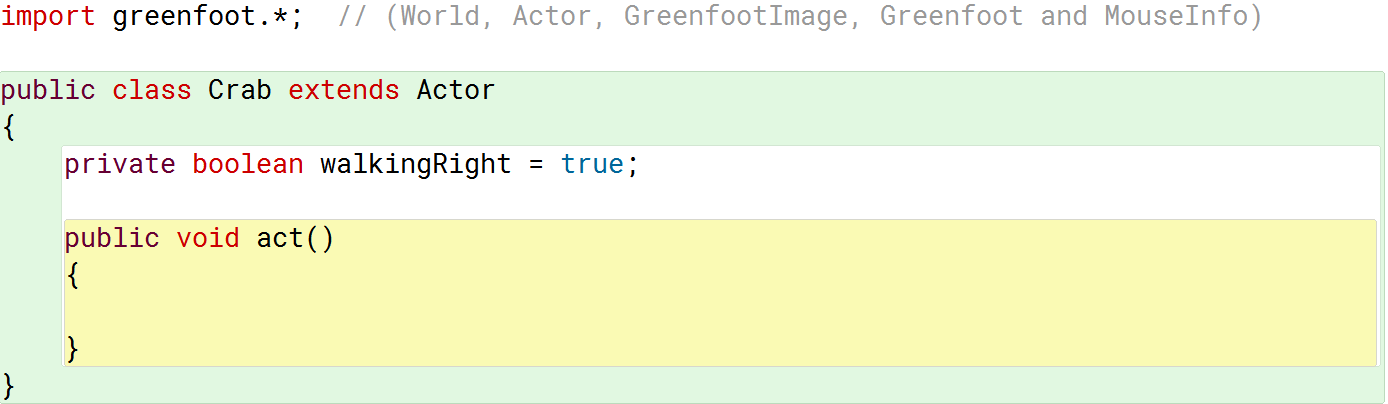
## The Crab

Why do we need to learn about variables to make the Crab? The intention of the crab is to walk forward until it hits a wall, then walk the other way, back and forth. However, if we just simply ask if the Crab has hit the edge of the screen then turn 180 degrees, and keep going, what will the crab look like? When it is walking one direction it will look correct, but when walking the other way, it will be upside down.



We can fix this by making a variable keeping track of if the crab should be walking right or left at the moment. If the crab is walking left, we could make it move in a positive X direction. If it is walking right, we could make it move in a negative X direction. We would never need the crab to turn if we do this.

What kind of variable would we need to keep track of this? Well, we are either walking right or we are not. That is only two options, which makes a boolean variable a great choice. A crab also needs to keep this knowledge permanently known, so it should be a field.

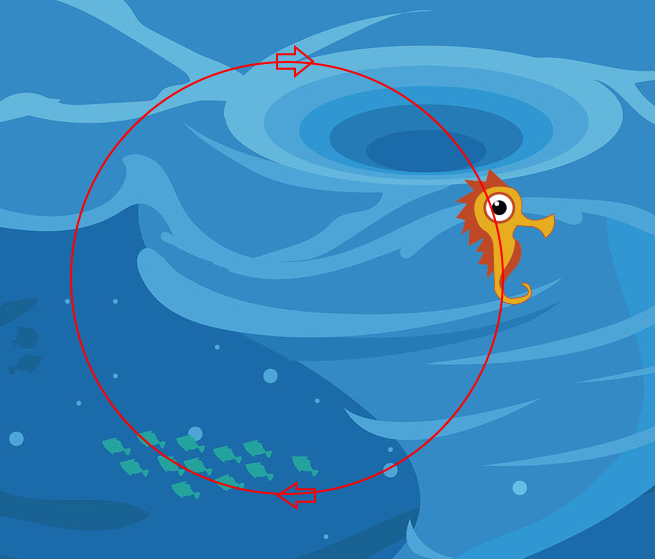


Your task is now to figure out how you could use that variable correctly. If the current value of walkingRight is true, then the crab should move in a positive X direction, otherwise the crab should move in a negative X direction. Finally, if the Crab is hitting the edge of the screen, what should you do with the walkingRight variable?

*Fourth Sea Animal: SeaHorse.*

Our fourth sea animal will be a SeaHorse. The SeaHorse is going to walk in a big circle. This could be accomplished simply with a move and a turn, but just like the Crab, there will end up being a problem” the SeaHorse will be upside down while walking around the circle.

You can attempt to figure this out on your own, or you can look further below for an idea.



Idea: What if you keep a field that keeps track of the angle the SeaHorse would be at if it were to be facing a certain direction right this moment? Each act, you could increase this angle by one, make the SeaHorse temporarily look that way, move forward, then look due east again (east is the default direction of all Actors).

*Fifth Sea Animal: Octopus.*

The Octopus is the first animal where you will add some interactive elements to it, and it will thus have a bit more complicated code. The Octopus is going to ‘fall’ through the water, as if gravity is pulling it faster down, then it will bounce up when it hits the ground. You will also make the Octopus propel up the aquarium by pushing a button on the keyboard.

Goals:

* Keep track of how fast (velocity) the octopus is current moving (a field).
* Determine if the Y value is very low on the screen, if so, set the octopus’s velocity to a negative value to make it ‘bounce’.
* Determine if a button (space) is pressed on the keyboard and if the octopus is moving fast enough downwards, if so, set the octopus’s velocity to a negative value. Also, use the Greenfoot static method playSound to play the sound file “blop.mp3”.
* If the first two questions were both false, slightly increase the velocity.
* Make the octopus teleport slightly, keeping the same X coordinate, but adding its old Y coordinate together with the current value of velocity.

### New Knowledge: SmoothMover

Greenfoot’s regular movement methods take ints, which means the slowest an Actor can move is 0 or 1 pixel each act. What if you wanted to move 0.5 pixels? Or 15.345 pixels? There is a relatively simple solution: SmoothMover. SmoothMover is a class you can import that includes the capability of moving with decimals (doubles). To get SmoothMover on your main Greenfoot window click ‘Edit’, ‘Import Class’, then choose ‘SmoothMover’.

Using SmoothMover: Instead of right clicking Actor and building a new subclass, right click SmoothMover and build a new subclass. The only difference for your purposes of SmoothMover and Actor is that SmoothMover’s move and setLocation methods can now take doubles.

The biggest change for SmoothMover is two new methods, getExactX() and getExactY(). Regular getX() and getY() are still available, but they return the closest int values for the X and Y of your SmoothMover. getExactX() and getExactY() will return the double values for the X and Y of your SmoothMover and are thus more accurate.

### Specific Numbers

You can try playing with the numbers yourself for the Octopus, but here are numbers that worked well:

* When the Octopus hits the ground, set the velocity value to -5.
* When the space bar is pressed AND the velocity is greater than 3, set the velocity to -6.
* If neither of those were true, add 0.1 to the velocity.

*Sixth Sea Animal: Turtle.*

The Turtle is going to be the largest set of code for all the sea animals. Follow the goals as best you can.

Goals:

* Keep track of how fast (velocity) the Turtle is current moving (a field).
* If the up arrow is pressed make the Turtle go slightly up the screen without rotating.
* If the down arrow is pressed make the Turtle go slightly down the screen without rotating.
* If the right arrow is pressed increase the velocity value by a small amount.
* If the left arrow is pressed decrease the velocity value by a small amount.
* Move the turtle forward by its current velocity amount.
* Set the velocity to a slightly smaller number (closer to zero) by multiplying its current value by a decimal. This is the concept of friction which makes any velocity value get closer to zero.
* If the velocity is positive, change the image to “turtleRight.png”, check the API for the needed method.
* If the velocity is negative, change the image to “turtleLeft.png”.

### Specific Numbers

You can try playing with the numbers yourself for the Turtle, but here are numbers that worked well:

* Up and down movement can be a constant 2 pixels.
* Increase or decrease the velocity by 0.5 when the right or left arrows are pressed.
* Multiply the velocity by 0.95 to apply friction.

*Final Sea Item: Food.*

The Food item is not a sea animal, instead the sea animals will want to eat it. The Food will periodically be added to the Aquarium. All the Food needs to do is fall down the screen slowly, and if it hits the bottom of the screen it needs to be removed from the Aquarium. However, there happen to be six different images for the Food: “food0.png”, “food1.png”, …, “food5.png”. It would seem wasteful to write six different Food classes when the only difference between one food and another is the image name it will use.

## New Knowledge: Asking the World to use one of its Methods

Sometimes an Actor needs to ask the World to do something for it. Early on it was mentioned to have another object to use one of its methods, you would need to know a variable name for that object.

Here is a quick rundown of how to do this for a World:

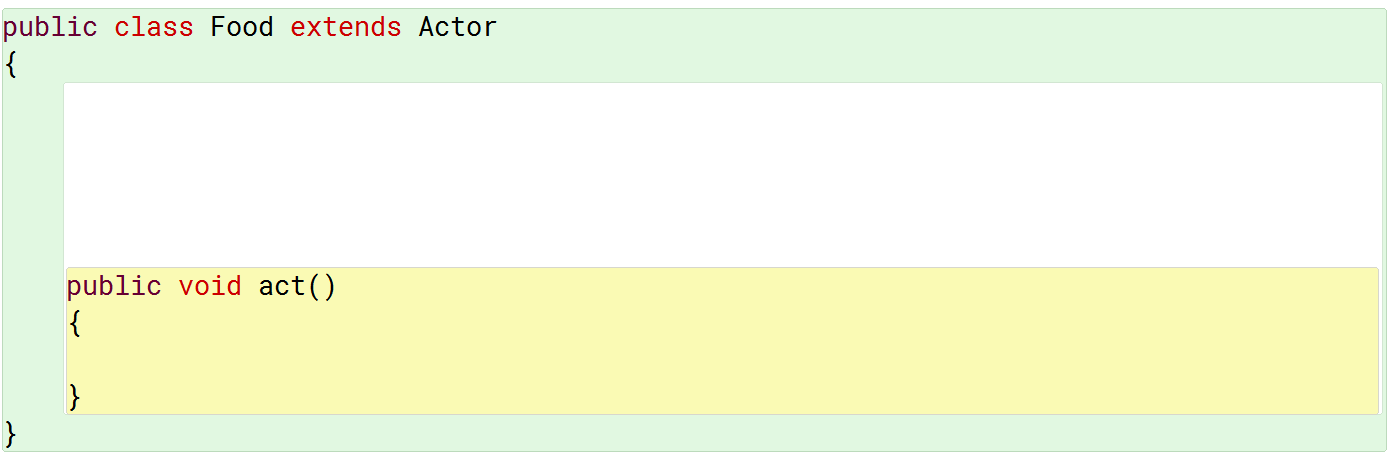
* Build a World variable and store in it your Actor’s World: World myWorld = getWorld();
* Ask that World variable to use one of its methods: myWorld.removeObject( this );
* The example shown asks the world to remove the current object (this = the current object).

### New Knowledge: Constructors

A Constructor is a special type of method that is used to build objects from the class ‘blueprints’. Each time you right click one of the Sea Animal types and hit ‘new ClownFish()’ or ‘new Crab()’ you are invoking the constructor for those particular types. A constructor constructs, so it builds items, just like a construction crew takes the blueprints of a house (a class) and then builds a dozen identical physical houses (the objects).

If you want a particular object to have specific code be triggered the moment it is created, then you will need to write a constructor. This topic will be more elaborated on the further you go into learning how to program, but at the moment just think of it as which code you want to immediately occur when an object is being created.

### Location of Constructors



### Writing a Constructor

The syntax for writing a constructor is always the same to start, but will modify as you learn more:

public ClassName ( parameter1, parameter2, … )

{

}

The ClassName must match exactly the actual name of the class you are working in. If you were writing code for the Turtle class, it would be public Turtle( ), if you were writing code for a class called PaperTowel, it would be public PaperTowel( ).

For the moment we can ignore the parameter options for Constructors and instead focus on an example of a constructor you could add to your Crab class.



When any Crab is built it will now randomly decide if it goingRight immediately or not. It has a 50% chance of the direction it will go. How is that true? Remember Math.random() gives a number from [0,1), what percent of those numbers are less than 0.5?

## The Food

Can you write a constructor for the Food class that will set its image randomly to “food0.png”, “food1.png”, …, or “food5.png” the moment the Food is built? Can you make the food fall down the screen without rotating it and to be removed from the world when it hits the bottom of the screen?

*The Aquarium Itself.*

All World types in Greenfoot will always have a constructor showing. The first line of that constructor will always be super( , , ); or some variation of that for reasons that you will learn when constructors are visited in more depth. That super line currently tells how many pixels big the World will be.

You may decide to add more code to the World’s constructor so that other actions are taken the moment the World itself is built. A common use of the World’s constructor is to add the starting Actors onto the screen so that they no longer have to be manually added. To do this, you will need to learn how to build Objects with code rather than right-clicking them and adding them yourself.

## New Knowledge: Building Objects (Calling Constructors)

To build an object you will need to use the keyword new to call the constructor of the object type you wish to build. You can either build this object and put it into a variable so that you can use the object multiple times, or you can build the object and use it just one time (like add it to the screen in Greenfoot).

### Option 1: Build an Object so that you can use it several times immediately.

Example: StarFish sam = new StarFish(); Builds a StarFish and puts it into a variable, sam.

addObject( sam, 300, 200 ); Uses the World method addObject to place

the StarFish sam at the coordinate 300, 200.

sam.turn( (int)(Math.random() \* 360 ) ); Tells sam to turn from 0-359 degrees.

### Option 2: Build an Object and immediately use it once. This is relatively useful for placing objects.

Example: addObject( new StarFish(), 300, 200 ); Builds a StarFish with no name and puts it

at the coordinate 300, 200.

## The Aquarium

Add at least one of each of the different Sea Animals in the Aquarium’s constructor at different locations around the screen. Make sure your code is below the super( 1136, 935, 1); line.

In the act method for the Aquarium, write an if statement that is only true 5% of the time. When the statement is true, build a Food item and place it at the very top of the screen with a random X coordinate.

*Options!*

There are always ways to make a program better, here are some you can try:

* Choose some of the sea animals and have them ‘eat’ (remove) any food they run into. You can play the “pling.wav” sound when this happens.
* Play the “waterDrops.wav” sound when the program starts. There is a way to loop this sound and make it keep playing if you learn how to use GreenfootSound.
* Create a few classes like ClownFishButton or CrabButton. Place them on the screen somewhere when the program starts. In the act method for each of these buttons have them wait until the mouse is pressed on them. When the buttons are pressed, create an object of the correct type and place it on the screen at some random X and Y coordinate.
* Try designing your own sea animal with your own rules for how it should move.
* Try making a health variable for one of the animals (Turtle?). Every act the health variable should go down, but when the Turtle eats food the health variable goes up. If the Turtle’s health ever reaches zero, have the Turtle be removed from the Aquarium.
* Try using some of the other methods not yet used here just for trial and error purposes. Some you may be able to figure out, some you could use the internet to help understand, and others you may wish to wait on until they are explained later.

# Answers to Question Sets

## Coordinate Potential Answers

1. In math, the ‘origin’ is at (0,0). Where is (0,0) in Greenfoot?
2. How big is this current world?
3. What are the coordinates of the four corners of the screen?
4. If the screen was 5000 wide and 4000 tall, what would the bottom right coordinate be?
5. What is the major difference between the coordinates in a math class and those on Greenfoot?
6. *(0,0) is the top left corner of the screen, much like images are defined in image editors.*
7. *1136 wide and 935 tall.*
8. *(0,0) is top left. (1135, 0) is top right. (0, 934) is bottom left. (1135, 934) is bottom right. Notice how the coordinates are one short of the width and height due to 0 being one of the coordinates.*
9. *(4999, 3999).*
10. *The Y values are inverted. Zero is at the top and positives Y values are below that.*

## Actor Method Potential Answers

1. Some of the methods have the word ‘int’ either before the name of the method or in the ( )s. What is int likely short for?
2. Some of the methods cause a pop up to appear that allows you to type in information. By looking at the list of methods above, how can you easily identify if you will need to type information in or not when you try to use a method?
3. When using the move method, what happens if you give it a positive number versus a negative number?
4. Very specifically, which direction does move with the number 50 make you go? Are you sure?
5. When using the turn method, what happens if you give it a positive number versus a negative number?
6. What does the number you type in to turn represent?
7. Add at least four total Fish to your screen. Use setRotation to make one face east, another south, another west, another north. What numbers did you use?
8. Tell each of the four Fish from the last question to move 50. Was your earlier answer correct?
9. What specific direction would a Fish face if they were told to setRotation to: 180? 45? -45? -90? 720?
10. Describe the difference between turn, setRotation, and turnTowards.
11. Which method would make sense to rename teleport?
12. Some of the methods cause a pop up to appear after the method has been chosen which shows you information. This is called ‘returning’ information to you. By looking at the list of methods above, how can you easily identify if a method will return information to you?
13. What does the word ‘void’ appear to mean?
14. What does the word ‘boolean’ appear to mean?
15. *Integer (whole numbers, both positive and negative).*
16. *The parenthesis ( )s are either empty or not. If they are empty, no information needs to be provided, if they are not then information needs to be provided.*
17. *Positives makes it move forward, negative makes it move backwards. This is often mistaken as students originally may perceive positive numbers as making actors move right and negative making them move left, but the following questions should cement the difference.*
18. *Forward, although it may appear to the right if no turning has occurred.*
19. *Positive makes the actors turn clockwise (to their right) and negatives makes them turn counterclockwise (to their left).*
20. *The number is a degree amount.*
21. *East is 0 (or 360, or 720, or -360, or any other multiple of 360).*

*South is 90 (or 450, or -270, etcetera).*

*West is 180 (or -180, 540, etcetera).*

*North is 270 (or -90, -450, etcetera).*

1. *Each fish should move in their version of forward.*
2. *(180, West), (45, Southeast), (-45, Northeast), (-90, North), (720, East).*
3. *turn is relative to the direction an actor is already facing. Turning a small number will nudge the actor slightly left or right of their current direction. setRotation forces the actor to face a very specific direction and has nothing at all to do with the direction the actor used to face. turnTowards tells an actor to face a particular coordinate on the screen. Multiple actors turning towards the same coordinate may have different angles they are now facing depending on where they started on the screen at.*
4. *setLocation would be the most like a teleport command.*
5. *To the left of the method’s name are things like int or World or boolean. These tell you information will be ‘returned’.*
6. *The word ‘void’ is used to represent a method will not ‘return’ an answer.*
7. *Boolean represents a true or false value.*

## Method Calling Potential Answers

1. What would the fish do if you added setLocation( 0, 0 ); just above the move(5); call?
2. What would the fish do if you replaced all of the code in the act method with   
   setLocation( getX(), getY() );
3. What would the fish do if you replaced all of the code in the act method with  
   setLocation( getX() – 5, getY() );
4. What line of code could make the fish move slightly towards the top of the screen no matter what direction it was facing?
5. What two lines of code could be used to make the fish face directly northeast and then move towards that direction?
6. What two lines of code would make the fish face the top left corner of the screen and then move towards that direction?
7. What line of code could make the fish teleport to the same y coordinate it already has but all the way to the left side of the screen?
8. What line of code could be used to duplicate the effect of turn( 5 ); without using the turn method?
9. *The fish would teleport to the top left corner of the screen -each- act, then move and turn a little. This would make the fish look like it is stuck in the corner but keep turning in place because it keeps teleporting over and over.*
10. *Setting the location of the fish to the same place it already is will make the fish not move at all.*
11. *Setting the location of the fish to the same y coordinate but a slightly less x coordinate would make the first go slightly left even though it could be facing any direction.*
12. *setLocation( getX(), getY() – 5 ); The number itself is not important, as long as it is negative. Remember, a smaller Y value is closer to the top of the screen. The X coordinate needed to use the same X coordinate it already had so it would not move left or right.*
13. *setRotation( -45 );  
    move( 5 );*
14. *turnTowards( 0, 0 );*

*move( 5 );*

1. *setLocation( 0, getY() );*
2. *setRotation( getRotation() + 5 ); Despite this being the answer, the turn method is a better choice because it already does this work. We do not want to rewrite code that already exists usually.*