The Snake Game Java Case Study

John Latham

January 27, 2014

Contents

1 Introduction **2**

2 Learning outcomes **2**

3 Description of the game **2**

4 The classes of the program **4**

5 The laboratory exercise and optional extra features **6**

6 High level design, and development approach **6**

7 Concept: model, view and control **8**

8 Separating behaviour and image: design policy **9**

9 Development of `Direction` **12**
- 9.1 Variable and method interface designs 12
- 9.2 Code for `Direction` 13

10 Development of `Cell` and `CellImage` **15**
- 10.1 Variable and method interface designs 17
- 10.2 Code for `Cell` 20

11 Development of `Game` and `GameImage` **25**
- 11.1 Method interface designs 25

12 Development of `SpeedController` and `SpeedControllerImage` **28**
- 12.1 Variable and method interface designs 29

13 Development of `AboutBox` **30**

14 Development of `GameGUI` **30**
- 14.1 Method interface designs 30

15 Development of the top level class `Snake` **30**

16 Conclusion **31**

1 Introduction

This case study presents much of the development of a program to play a snake game, similar to that found on certain old mobile phones. The core game playing functionality is actually left as a staged laboratory exercise.

2 Learning outcomes

The intentions of this case study are as follows.

- To reinforce many of the Java and programming concepts you have already met.
- To provide valuable experience of the design and implementation of a large program.
- To provide a framework for a more challenging, and thus rewarding, laboratory exercise.

Whilst there is no substitute for writing your own programs, watching the development of another's is still an extremely effective way of collecting design and programming experience, particularly as the programs can be a little more challenging than those you would be able to write yourself at this stage. (Caveat: this is only true for case studies that you are *not* supposed to be doing yourself – watching someone else develop code that you are supposed to be doing on your own is disastrous for your learning!)

How much you get from this case study depends on how much you put in. Ideally you should carefully follow every part of it and check your depth of understanding at every opportunity. All confusions or misunderstandings which it may reveal must be dealt with immediately.

3 Description of the game

The game is similar to snake on mobile phones (which is similar to a game played on Unix over 30 years ago), but with some 'improvements'.

The game is played by one player, who has the objective of obtaining the highest score possible. The player is in control of a snake which is constantly moving around a square field of cells. The length of the snake is a whole number of cells. At any time, the snake moves in one of the 4 directions, parallel to a side of the square, and the player can change the direction using the 4 arrow keys of the keyboard. If the snake crashes into a side, or into itself, then it is dead, and the game stops.

Also in the field is a single piece of food. When the head of the snake hits this, the food is eaten, and the snake becomes one cell longer. This event increases the score of the player. At

the same time, a new piece of food is placed in a randomly chosen cell somewhere in the field, which was previously clear (empty).

The game has a score message bar, informing the player what is the current score, and also a single message which changes from time to time. For example, when food is eaten the player is informed how much score was just obtained, and when the snake crashes a suitable message is shown.

The player can make the game go slower or faster, and can alter the speed during the game, by pressing ‘s’ to slow down and ‘f’ to speed up. The score obtained when a piece of food is eaten is proportional to the speed setting. The game can also be paused and resumed by pressing ‘p’ and ‘r’ respectively. The game will be in a paused state when it starts. The speed controller can be placed in interactive mode by the player pressing ‘i’ – this enables the player to see the current speed and also alter it via buttons.

At any time, the player can end the game and start a new one, simply by pressing ‘a’ on the keyboard. The most common use of this will be after the snake has crashed, so as to continue playing with a new game.

When the snake is dead, the player has the option to ‘cheat’ by making it come back to life so play can continue. However, this costs the player half of his or her score. The cheat feature is invoked when the player presses ‘c’ on the keyboard.

The player has the option of enabling trees to appear in the field. This feature is toggled on and off by pressing ‘t’ on the keyboard. When trees become enabled, a single tree appears in a randomly chosen clear cell. Then, each time the food is eaten, another tree appears somewhere, and so on. This makes the game more interesting, because the snake will die if it crashes into a tree. The game is thus more difficult, and so if trees are enabled when the food is eaten, the score obtained is multiplied by the number of trees in the field at the time. When the trees feature is toggled off, all the trees vanish.

Each time the snake crashes into something, it will not die immediately. Instead, a ‘crash countdown’ begins, and reduces by one each move time of the game. The player sees this count down via the score message bar. If the snake is moved away before the count down reaches zero, it has escaped death.

The game offers a single ‘about’ and ‘help’ box, popped up and down by pressing ‘h’.

There is no requirement for keeping track of high scores.

Figure 1 shows a sneak preview of the finished game program.

Figure 1: A screen dump of a game. The optional extra feature of a gutter trail is visible behind the snake.

4 The classes of the program

As is usual in the object oriented approach, the program will be developed as a number of separate classes. Identifying the appropriate classes of a program is really an art which can slowly be learned from experience. And like the result of any art, the quality of any choice of classes is subjective (although most people will agree there are many more bad choices then good ones!).

By thinking about our understanding of the game, we can form a list of classes which will appear in the program.

Class list for Snake Game: main model classes	
Class	Description
`Cell`	The game grid is made up of cells. Each cell has a type and a number of attributes. For example, a cell could be clear, or it could be a snake head facing in a particular direction, etc..
`Direction`	The snake in the game moves in one of four directions. This non-instantiable class provides those four directions, and various methods to manipulate them (such as obtaining the opposite of a direction).
`Game`	This is the main behaviour of the game play. Its primary purpose is to provide a method for making one move of the game.

We will also require some other classes to support the operating framework, including the following.

Class list for Snake Game: operating support classes	
Class	Description
`GameGUI`	This will provide a user interface so that the player can see the snake and the food, etc., and be able to operate the game via the keyboard. It will also be the driving force for the game, that is, it shall contain a method to play the game by repeatedly invoking the move method from the `Game` class.
`SpeedController`	A speed controller is something which controls the speed of the game, to make it easier or harder for the player as they choose. This will be achieved rather simply by the program calling a method to cause a delay between every move. A controller will have various speed settings, the ability to increase or decrease speed, and to pause and resume the game.
`AboutBox`	This will provide a pop-up box which can display information about the program and help on how to use it.

Finally, we shall need a class to contain the main method.

Class list for Snake Game: top level class	
Class	Description
`Snake`	This is the top level class, containing only the main method. This shall simply create an instance of `GameGUI`, invoke its game playing method and print out the score once the game has finished.

5 The laboratory exercise and optional extra features

The laboratory exercise associated with this case study will involve you creating the `Game` class, without needing to change the other classes of the program.

The rest of the program offers flexibility for you to implement optional extra features. This flexibility is achieved by two mechanisms. 1) any key presses not recognised by the `GameGUI` class are passed directly to a method called `optionalExtraInterface()` in the `Game` class. 2) a large number of additional values of cell are available in the `Cell` class. These all appear as different shades of the field background colour.

Particular suggested optional extras are as follows. 1) Make the food be able to move by itself and run away from the snake. 2) Make the snake be able to leave grass or gutter trails behind it, which the food cannot move across but which fade slowly away. 3) Make the snake have the ability to blast away a tree just in front of it.

You can also think of your own extras, such as an auto-play demonstration mode, where the snake steers itself, or having two snakes with the user switching control between them.

6 High level design, and development approach

In this case study, we shall develop the program incrementally, class by class in a **bottom-up** manner, that is, we shall attempt to fully design and implement each class before designing any class which we think will need it. One reason for developing in this order is that it will be easier for you to follow the discussion. Identifying the dependency relationships between classes before they are designed is not easy, but the ability comes with experience. We obtain them by thinking very abstractly about the classes from the main class downwards, i.e. in a **top-down** manner.

Here is the list of classes we have identified so far: `Cell`, `Direction`, `Game`, `GameGUI`, `SpeedController`, `AboutBox`, and `Snake`.

`Snake` will be the top level, it will create a `GameGUI` and invoke its `playGame()` method. So, the main method will contain a code fragment something like the following.

```
get the gameSize from the command line (default 18)
GameGUI gameGUI = new GameGUI(gameSize)
int score = gameGUI.playGame()
output the score
```

A `GameGUI` will create an instance of `Game` and `SpeedController`. Its `playGame()` method will look something like this.

```
set initial game state
while user has not asked to quit
{
  speedController.delay(by some amount)
  moveValue
    = some function of (speedController.getCurrentSpeed())
  game.move(moveValue)
  update the game's image
}
return game.getScore()
```

A `GameGUI` will also interpret the key presses from the player, and invoke the appropriate methods: e.g, to get the `SpeedController` to change speed or pause and resume; to create and show the `AboutBox`, to change the direction of the snake in the `Game`, etc..

A `Game` will primarily offer the method `move()`. This will cause the game to change state, progressing by one move. The new state will depend on the settings of the old state. Where is the head of the snake? What direction is it facing? Where is the food? Etc.. There will also be a method `setSnakeDirection()` which will be called from the `GameGUI` when the player presses one of the arrow keys. There will be other methods too, such as `getScore()`, `cheat()` and `toggleTrees()`. The state of the game will be represented by a number of instance variables, the most obvious one being a 2-dimensional array of `Cells` to make up the field.

A `SpeedController` will have a number of methods, such as `speedUp()`, `slowDown()`, `pause()`, `resume()` and, most importantly, `delay()`.

A `Cell` will store information, such as is there a piece of snake in the cell? If so, what direction is it facing? Or, is there a piece of food here? Or a tree? Etc.. It will have **accessor method**s to access and **mutator method**s to mutate the information it stores, so the state of a cell can be seen and changed.

The `Direction` class will be non-instantiable, it will simply provide 4 directions, and some useful methods to manipulate them. For example, the method `xDelta()` will return what value you need to add to an x coordinate, if you want to make a movement in a certain direction given as its argument. (The amount returned will be -1, 0 or 1.) This combined with the similar `yDelta()` will make the new position of the head of the snake easy to compute from the direction it is facing.

An `AboutBox` will simply pop up to display some text given to it when it is created, and provide a dismiss button to make it go away. This will be used to provide some help to the player.

Let us analyse the anticipated dependencies between these classes.

- `Snake` is the top level class. This will build an instance of `GameGUI`.

- `GameGUI` will need an `AboutBox` to show help text, a `SpeedController` to regulate the speed of the game, and a `Game` to play.
- `AboutBox` will not require any other of these classes.
- `SpeedController` will not require any other of these classes.
- `Game` will require `Cell` for the field of cells and `Direction` for the direction of the snake.
- `Cell` will require `Direction` so it can store which direction the pieces of snake are facing.
- `Direction` will not require any other of these classes.

So, we plan to develop the classes in the following order: `Direction`, `Cell`, `Game`, `SpeedController`, `AboutBox`, `GameGUI`, and finally `Snake`.

7 Concept: model, view and control

In general, programs have some kind of behaviour and an interface connecting the behaviour to the outside world. The behaviour is often referred to as the **model**, as it is this that must implement the real world functionality required of the program.

The interface to a program can be divided into two categories: output and input, often referred to as **view** and **control**, respectively.

There is much debate in the O.O. world about these three functions of a program. The so-called **M.V.C.** argument (standing for **model**, **view** and **control**) suggests there is much to be gained by separating the 3 functions into different classes in the final program. On the other hand, some people prefer to combine them – there are no hard and fast rules.

Perhaps the main reason for separating the parts is that it more easily allows certain parts to be changed without affecting the others. For example, somebody might find a better way of modelling the real world requirements, but is quite happy with the existing user interface. Or someone might make a better interface for an existing program which is otherwise quite acceptable. The inherent flexibility can be exploited from the initial design too. For example, a program might have a number of different interfaces to support different operating environments, whilst having only one model part. Perhaps the most obvious examples are those programs which have both a text and a graphical interface.

The most persuasive argument for separating the model from the other two parts in a learning environment is that it permits us to study quite complex programs long before we need to discover the details of *complex* graphical user interfaces: we can simply hide the specifics of the control and view parts of the program at this point in time.

Once we separate the model from the view and control, we need to find a way to link them together so that they work properly. For example, the user pressing a button in a control object needs to cause something to happen in a model object, and then whatever state change that results in must be shown by a view object. There are various strategies for keeping track of the relationship between corresponding instances of these classes, and we shall not attempt to describe them all here. There is no best way to do it for all cases, and there are alternative approaches within the context of a single program. One of the factors affecting the choice is whether a model object can have more than one associated instance of a view or control object, or vice versa. In some programs, there is a one to one correspondence, but in others we might wish to have more than one view and control objects for one model object, and sometimes we might wish to have no view objects, etc.. Less commonly, we might wish to have an instance of a view class without a corresponding model instance.

8 Separating behaviour and image: design policy

This program will need to **model** the behaviour of a snake moving around a field of cells, and eating food etc.. It must also provide a **view** of that behaviour, so that the player can easily see the snake, the food and the trees behaving in ‘real time’. It must additionally provide a **control** mechanism for the player to be able to steer the snake, change the speed, etc..

We will keep a clean separation between the model and the view classes. This means, for example, we shall have a class called `Cell` and a separate one called `CellImage`. The former will be concerned with the model of a cell, and the latter merely about allowing the picture of a cell to appear on the screen. Similarly, the class `Game` will be concerned with the behaviour of a game which consists of cells, whereas `GameImage` will be responsible for allowing images of games to be seen on the screen (and will be composed of `CellImages`).

However, we are not necessarily going to separate the view from the control: indeed these often fit together nicely in a graphical user interface context. The controlling input from the user will be handled by the `GameGUI` class, which will also provide a view for the score message bar, and will contain the game image.

Let us make a list of the extra classes we have just identified.

Class list for Snake Game: image classes	
Class	Description
`CellImage`	Provides images for cells. Each instance of `CellImage` will have a corresponding instance of `Cell`, which it will store as an instance variable, and it will provide the image of that cell.
`GameImage`	Provides images for games. Each instance of `GameImage` will have a corresponding instance of `Game`, which it will store as an instance variable, and it will provide the image of that game.

Class list for Snake Game: image classes	
Class	Description
`SpeedControllerImage`	Provides the image and control for speed controllers.

Having made the decision to develop separate classes for model and view, we next must figure out the way we would like to do it. For example, for each instance of `Cell` which we create, we will also need to create an instance of `CellImage`. We must decide on a strategy for keeping track of the relationship between corresponding instances of these classes. There are numerous ways we could do this. The approach we have chosen in this case study has actually been taken so that *you* can develop the `Game` class without having to worry about view and control classes! So, if we were not in such a learning context, we *might* have taken a different approach.

Here is the policy we shall use in this case study.

- An instance of `GameGUI` will be created by the main method. This will create an instance of `Game` and then also an instance of `GameImage`. A reference to the game instance will be passed to the instance of the game image, so that the game image knows which game to provide a view for.
- The instance of `GameImage` will construct an instance of `CellImage` for every cell in the game it has been given. It will keep track of all these cell images, and also combine them to produce the image of the game.
- When an instance of `CellImage` is created, it is passed a reference to the instance of `Cell` for which it must provide an image.
- The instance of the class `GameImage` will have an `update()` method. This will be invoked after every move, and at other times, by the instance of `GameGUI`. This method will invoke a similar `update()` method on each instance of `CellImage`, which will repaint the image of the corresponding cell, if the cell has been changed since the image was last painted.

As suggested above, the attraction of this strategy is that the model classes for the game and the cells need to have absolutely no knowledge of the image classes. This is particularly persuasive given that the `Game` class is to be written by you!

A second attraction is that we can also hide most of the details of the image and control classes for this current discussion.

Figure 2 shows how the classes will be used in the program. The diagram is simplified, and in particular ignores the class `Direction`.

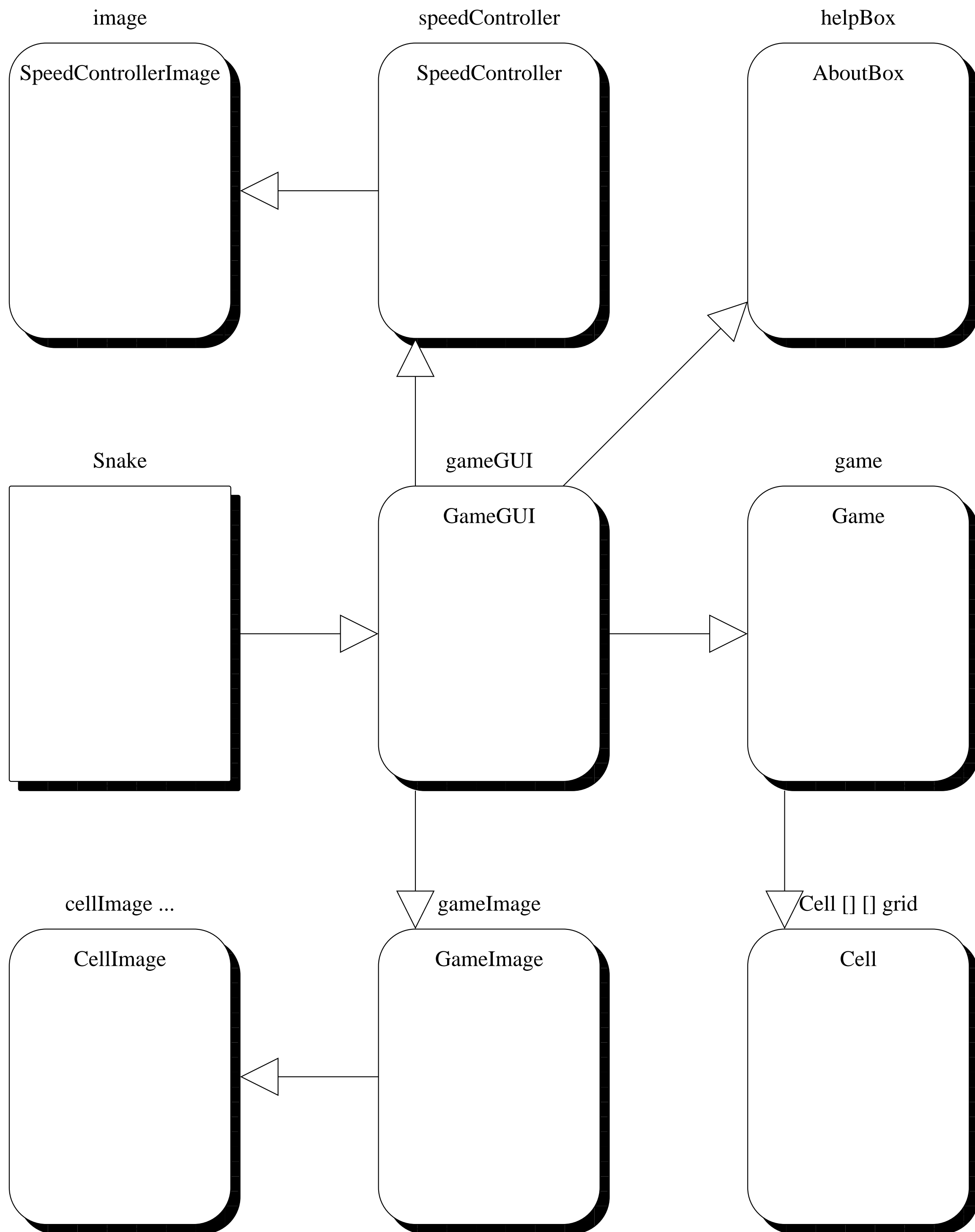

Figure 2: Simplified diagram showing the most significant classes of the program. The class `Direction` is not shown.

9 Development of `Direction`

Now we can start the development of the program. The first class on our list is `Direction`. This will provide the 4 directions for snake movement, together with some useful methods for manipulating directions. We do not intend that the class will be instantiated – all its public items will be accessed statically.

The primary purpose of this class is to provide 4 constant values, one each for the 4 directions that a snake can move in. We shall also have a 5th direction, to model no movement. One of the common ways of implementing a small range of constant values is to model them as integers, and we shall use this approach to implement the 5 directions. However, it should be noted that while this technique is quite commonly used, it has a significant danger. Can you think what it is?

9.1 Variable and method interface designs

`Direction` will have the following public variables.

Variable interfaces for class `Direction`.

Variable	Type	Description
`NONE`	`int`	A static final integer value representing the direction that is a model of no direction.
`NORTH`	`int`	A static final integer value representing the direction that is north.
`EAST`	`int`	A static final integer value representing the direction that is east.
`SOUTH`	`int`	A static final integer value representing the direction that is south.
`WEST`	`int`	A static final integer value representing the direction that is west.

The class will also have the following public methods.

Method interfaces for class `Direction`.

Method	Return	Arguments	Description
`opposite`	`int`	`int`	Static (class) method that returns the opposite direction to the given one.
`xDelta`	`int`	`int`	Static (class) method that takes an integer representing a direction, and returns -1, 0, or 1: this being the change needed to the x component of a cell co-ordinate in order to move in the given direction.

Method interfaces for class `Direction`.			
Method	Return	Arguments	Description
`yDelta`	`int`	`int`	Static (class) method that takes an integer representing a direction, and returns -1, 0, or 1: this being the change needed to the y component of a cell co-ordinate in order to move in the given direction.
`rightTurn`	`int`	`int`	Static (class) method that takes an integer representing a direction, and returns the integer representing the direction that is a right turn from the given one.
`leftTurn`	`int`	`int`	Static (class) method that takes an integer representing a direction, and returns the integer representing the direction that is a left turn from the given one.

9.2 Code for `Direction`

Next we develop the code for the class. We need to choose an integer to represent each of the direction values we need to provide. The most common way of doing this is simply to count upwards from 0, directly assigning the values to each variable.

```
public class Direction
{
  public static final int NONE = 0;
  public static final int NORTH = 1;
  public static final int EAST = 2;
  public static final int SOUTH = 3;
  public static final int WEST = 4;
```

The method to find the opposite of a direction simply requires 5 cases.

```
  public static int opposite(int direction)
  {
    switch (direction)
    {
      case NORTH: return SOUTH;
      case SOUTH: return NORTH;
      case WEST:  return EAST;
      case EAST:  return WEST;
      default:    return direction;
    } // switch
  } // opposite
```

We could instead have used a formula exploiting the actual values of the directions – see if you can figure out what it would be.

The method to find the x delta to move in a certain direction also simply requires 5 cases. It is at this point that we need to fix the relationship between coordinates and directions. We arbitrarily choose to have cell coordinates in the form (x,y), with (0,0) as the top left corner cell, and with the direction `NORTH` being upwards, `SOUTH` being downwards, `WEST` being leftwards and finally `EAST` being rightwards.

```
public static int xDelta(int direction)
{
  switch (direction)
  {
    case NORTH: return 0;
    case SOUTH: return 0;
    case WEST:  return -1;
    case EAST:  return 1;
    default:    return 0;
  } // switch
} // xDelta
```

The method to find the y delta is similar.

```
public static int yDelta(int direction)
{
  switch (direction)
  {
    case NORTH: return -1;
    case SOUTH: return 1;
    case WEST:  return 0;
    case EAST:  return 0;
    default:    return 0;
  } // switch
} // yDelta
```

Turning left or right again involves 5 cases.

```
public static int rightTurn(int direction)
{
  switch (direction)
  {
    case NORTH: return EAST;
    case EAST:  return SOUTH;
```

```
      case SOUTH: return WEST;
      case WEST:  return NORTH;
      default:    return NONE;
    } // switch
  } // rightTurn


  public static int leftTurn(int direction)
  {
    switch (direction)
    {
      case NORTH: return WEST;
      case WEST:  return SOUTH;
      case SOUTH: return EAST;
      case EAST:  return NORTH;
      default:    return NONE;
    } // switch
  } // leftTurn

} // class Direction
```

10 Development of `Cell` and `CellImage`

Next we can consider the class `Cell` and its associated image class `CellImage`. As stated above, we shall develop all the details of `Cell` here, but hide most of `CellImage`. We shall start by identifying the public variable and method interfaces for both of these classes, and then present the code for `Cell` only.

A cell will have a type, and a number of associated attributes. These are as follows.

`NEW` This is for a cell that has just been created, and not yet assigned a type.

`CLEAR` This is for a cell that has nothing in it, i.e. an empty part of the field.

`SNAKE_HEAD` This is for a cell that has the head of the snake in it. Such a cell will also have three attributes, a boolean and 2 directions. The `snakeBloody` boolean is true if and only if the snake is 'dead', usually because the head has crashed into something. The `snakeOutDirection` indicates which way the head is facing. The `snakeInDirection` indicates which way the head has just come from. For example, if a snake is travelling north for a while and then turns to face east, at that moment the snake out direction will be east, and the snake in direction will be south. Figure 3 shows a sneak preview of all 16 combinations of the in and out directions.

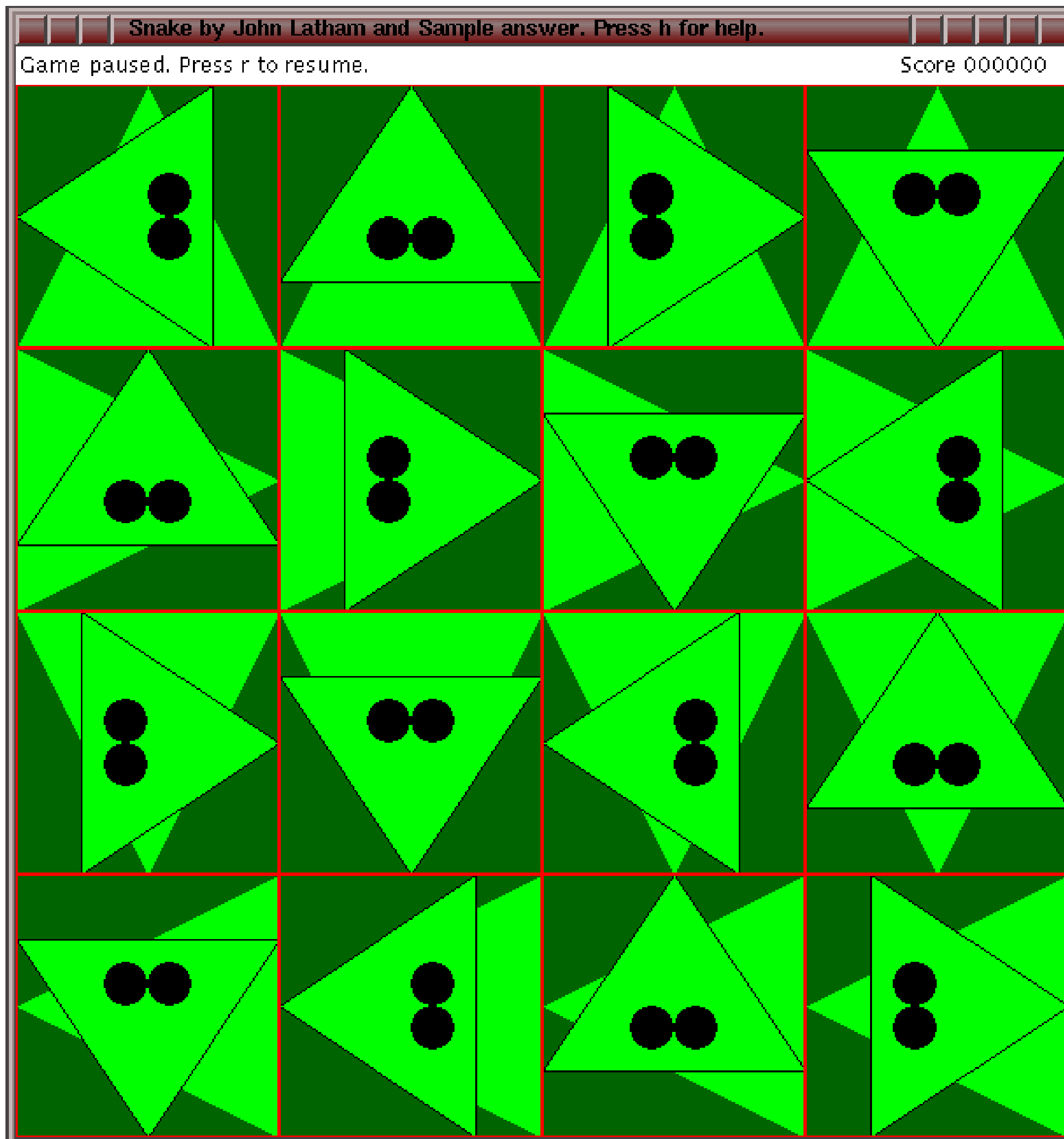

Figure 3: A sneak preview of all 16 combinations of in and out directions of snake heads.

SNAKE_BODY This is for a cell that has part of the body of the snake in it. This also has the same 3 attributes as a `SNAKE_HEAD` cell. The `snakeBloody` boolean is true if and only if the snake is 'dead' at this point, usually because the head of the snake has crashed into this part of the body. The `snakeOutDirection` states in which direction the next piece of snake, nearer to the head, can be found; the `snakeInDirection` states in which direction the previous piece of snake, nearer to the tail, can be found. You can think of a piece of snake as being like a joint of 2 bones. The 2 bones can swivel at the joint, and each bone is fixed to a bone in the next part of the snake. The two directions are stating which way the 2 bones are facing.

SNAKE_TAIL This is for a cell that has the tail of the snake in it. This also has the same 3 attributes. The `snakeBloody` boolean is true if and only if the snake is 'dead' at the tail, usually because the head of the snake has crashed into it. The `snakeOutDirection` states in which direction the next piece of snake, nearer to the head, can be found; the

snakeInDirection states in which direction the tail of the snake has just come from.

FOOD This is for a cell that has the food in it.

TREE This is for a cell that has a tree in it.

OTHER This is for a cell that can be used for optional extra features. It has 1 attribute the otherLevel which is an integer between -255 and 255 inclusive. The image of such a cell will always be a shade of the background colour, with the brightness proportional to the level: a negative number is darker than the background, and 0 is the same colour as the background. In practice there may not be 511 different shades of the background colour available, so some values will look the same. For example, if the background colour is pure green at a value of 100, that is the red, green, blue triple is (0, 100, 0), then there will be 255 different shades brighter than this, but only 100 shades darker. So -100 will be shown as the colour (0, 0, 0) which is black, but so will all the values less than -100. 255 will be shown as the colour (255, 255, 255) which is white – no longer a green really.

10.1 Variable and method interface designs

Cell will have the following public variables.

Variable interfaces for class Cell.

Variable	Type	Description
int	NEW	A static final integer value representing the type of cell which is a newly created cell.
int	CLEAR	A static final integer value representing the type of cell which has nothing in it.
int	SNAKE_HEAD	A static final integer value representing the type of cell which contains a snake head.
int	SNAKE_BODY	A static final integer value representing the type of cell which contains a part of snake body.
int	SNAKE_TAIL	A static final integer value representing the type of cell which contains a snake tail.
int	FOOD	A static final integer value representing the type of cell which contains a piece of food.
int	TREE	A static final integer value representing the type of cell which contains a tree.
int	OTHER	A static final integer value representing the type of cell for optional extra cell types.

The class will also have the following public methods.

Method interfaces for class `Cell`.			
Method	Return	Arguments	Description
Constructor			Constructs a new cell, with type `NEW`.
`getType`	`int`		Returns the type of the cell.
`isSnakeType`	`boolean`		Returns true if and only if the cell is a snake head, body or tail.
`setClear`	`void`		Sets the cell to be a clear cell.
`setSnakeHead`	`void`		Sets the cell to be a snake head, not bloody, but without changing the direction attributes.
`setSnakeHead`	`void`	`int, int`	Sets the cell to be a snake head, not bloody, and with the 2 given directions for snake in and snake out respectively.
`setSnakeBody`	`void`		Sets the cell to be a snake body, not bloody, but without changing the direction attributes.
`setSnakeBody`	`void`	`int, int`	Sets the cell to be a snake body, not bloody, and with the 2 given directions for snake in and snake out respectively.
`setSnakeTail`	`void`		Sets the cell to be a snake tail, not bloody, but without changing the direction attributes.
`setSnakeTail`	`void`	`int, int`	Sets the cell to be a snake tail, not bloody, and with the 2 given directions for snake in and snake out respectively.
`setFood`	`void`		Sets the cell to be a food cell.
`setTree`	`void`		Sets the cell to be a tree cell.
`setOther`	`void`	`int`	Sets the cell to be an other cell, with the given level.
`getOtherLevel`	`int`		Returns the value of the other level attribute.
`isSnakeBloody`	`boolean`		Returns true if and only if the snake is bloody at this cell.
`setSnakeBloody`	`void`	`boolean`	Makes the snake bloody at this cell if the given boolean is true, or not bloody otherwise.
`setSnakeInDirection`	`void`	`int`	Sets the snake in direction to the given direction.
`getSnakeInDirection`	`int`		Returns the snake in direction.
`setSnakeOutDirection`	`void`	`int`	Sets the snake out direction to the given direction.
`getSnakeOutDirection`	`int`		Returns the snake out direction.

Method interfaces for class `Cell`.			
Method	Return	Arguments	Description
`clone`	`Object`		Makes a new cell which has the same type and attributes as this one, and returns a reference to it.
`equals`	`boolean`	`Cell`	Compares 2 cells for an exact match of type and all attributes, returning true if and only if they match.

The class `CellImage` will have the following public methods.

Method interfaces for class `CellImage`.			
Method	Return	Arguments	Description
Constructor		`Cell, int`	Builds a cell image object, to provide the image of the given `Cell`, this image being a square of pixels of the given integer size.
`paint`	`void`	`Graphics`	Paints the actual image of the cell associated with the cell image object.
`getPreferredSize`	`Dimension`		Returns the desired size of the image, as an instance of the class `Dimension`.
`update`	`void`		The cell is compared with the value it had last time the image was painted, and causes a repaint if it has changed.

Let us make some observations about this list of methods.

- `paint()` and `getPreferredSize()` are used by the Java GUI mechanisms, and we are not going to study the details of them here.
- `update()` will be called from `GameImage` as part of its `update()` method.
- `update()` will use the `clone()` and `equals()` methods of the `Cell` class, to keep a copy of the previous value of a cell, and compare the current one to it, respectively.

10.2 Code for `Cell`

We can now develop the code for the `Cell` class. The class has 8 static (class) variables, one each for the different types of cell. Each instance will have 5 private instance variables, to store the type and the associated attributes of the cell.

```
public class Cell
{
  // Values for cell type
  public static final int NEW = 0;
  public static final int CLEAR = 1;
  public static final int SNAKE_HEAD = 2;
  public static final int SNAKE_BODY = 3;
  public static final int SNAKE_TAIL = 4;
  public static final int FOOD = 5;
  public static final int TREE = 6;
  public static final int OTHER = 7;

  // Current cell type
  private int cellType;

  // Attributes used if cellType is a snake type:

    // Is the snake bloody (dead) at this cell?
    private boolean snakeBloody;

    // What are the incoming and outgoing directions of the snake?
    private int snakeInDirection;
    private int snakeOutDirection;

  // Attribute used if cellType is OTHER:

    // A level from -255 to 255
    private int otherLevel;
```

The constructor takes no arguments, and builds a new cell with the type `NEW`.

```
  public Cell()
  {
    cellType = NEW;
  } // Cell
```

Next we write the **accessor method** for the cell type. The purpose of **accessor method**s is to permit public read access to private instance variables.

```
public int getType()
{
  return cellType;
} // getType
```

For convenience, we have the boolean function indicating whether the cell is part of a snake.

```
public boolean isSnakeType()
{
  return    cellType == SNAKE_HEAD
         || cellType == SNAKE_BODY
         || cellType == SNAKE_TAIL;
} // isSnakeType
```

Now we have various **mutator method**s to set the cell type. The purpose of **mutator method**s is to permit public write access to private instance variables.

```
public void setClear()
{
  cellType = CLEAR;
} // setClear
```

For convenience, we have two methods to set the type to be a snake head, one which changes the directions and one which does not.

```
public void setSnakeHead()
{
  cellType = SNAKE_HEAD;
  snakeBloody = false;
} // setSnakeHead
```

```
public void setSnakeHead(int requiredSnakeInDirection,
                         int requiredSnakeOutDirection)
{
  cellType = SNAKE_HEAD;
  snakeBloody = false;
  snakeInDirection = requiredSnakeInDirection;
  snakeOutDirection = requiredSnakeOutDirection;
} // setSnakeHead
```

Similarly, we have two methods to set the type as a snake body.

```
public void setSnakeBody()
{
  cellType = SNAKE_BODY;
  snakeBloody = false;
} // setSnakeBody


public void setSnakeBody(int requiredSnakeInDirection,
                         int requiredSnakeOutDirection)
{
  cellType = SNAKE_BODY;
  snakeBloody = false;
  snakeInDirection = requiredSnakeInDirection;
  snakeOutDirection = requiredSnakeOutDirection;
} // setSnakeBody
```

Also, we have two methods to set the type as a snake tail.

```
public void setSnakeTail()
{
  cellType = SNAKE_TAIL;
  snakeBloody = false;
} // setSnakeTail


public void setSnakeTail(int requiredSnakeInDirection,
                         int requiredSnakeOutDirection)
{
  cellType = SNAKE_TAIL;
  snakeBloody = false;
  snakeInDirection = requiredSnakeInDirection;
  snakeOutDirection = requiredSnakeOutDirection;
} // setSnakeTail
```

We have a method for setting the type as food, and another as a tree.

```
public void setFood()
{
  cellType = FOOD;
} // setFood
```

```
public void setTree()
{
  cellType = TREE;
} // setTree
```

The mutator to set the type as an ‘other’, requires the other level as an argument. This number is truncated into the allowed range if necessary.

```
public void setOther(int requiredOtherLevel)
{
  cellType = OTHER;
  otherLevel = requiredOtherLevel;
  if      (otherLevel < -255) otherLevel = - 255;
  else if (otherLevel > 255)  otherLevel = 255;
} // setOther
```

We also have an accessor method for the other level. Note that this does not bother to check whether the type is an ‘other’.

```
public int getOtherLevel()
{
  return otherLevel;
} // getOtherLevel
```

The method to determine whether the cell is bloody similarly does not bother to check if the cell type is a snake type.

```
public boolean isSnakeBloody()
{
  return snakeBloody;
} // isSnakeBloody
```

The mutator method to set the cell bloody also does not bother to check whether the type is a snake type.

```
public void setSnakeBloody(boolean requiredSnakeBloody)
{
  snakeBloody = requiredSnakeBloody;
} // setSnakeBloody
```

The two mutator methods to set the two snake directions, and the two accessor methods to return them do not bother to check the type either.

```
public void setSnakeInDirection(int requiredSnakeInDirection)
{
  snakeInDirection = requiredSnakeInDirection;
} // setSnakeInDirection


public int getSnakeInDirection()
{
  return snakeInDirection;
} // getSnakeInDirection


public void setSnakeOutDirection(int requiredSnakeOutDirection)
{
  snakeOutDirection = requiredSnakeOutDirection;
} // setSnakeOutDirection


public int getSnakeOutDirection()
{
  return snakeOutDirection;
} // getSnakeOutDirection
```

The instance of `CellImage` associated with a cell will need to determine when the cell has changed. It will do this by keeping a copy of the cell when its image is produced. The `clone()` method makes a copy of the cell and returns a reference to it.

```
public Object clone()
{
  Cell result = new Cell();
  result.cellType = cellType;
  result.snakeBloody = snakeBloody;
  result.snakeInDirection = snakeInDirection;
  result.snakeOutDirection = snakeOutDirection;
  result.otherLevel = otherLevel;
  return result;
} // clone
```

Finally, the `equals()` method compares this cell with another by looking at each of the instance variables. It returns true if they all match, false otherwise.

```
  public boolean equals(Cell other)
  {
    return other.cellType == cellType
           && other.snakeBloody == snakeBloody
           && other.snakeInDirection == snakeInDirection
           && other.snakeOutDirection == snakeOutDirection
           && other.otherLevel == otherLevel;
  } // equals

} // class Cell
```

11 Development of `Game` and `GameImage`

We have already stated that it will be you who develops the `Game` class, and I will develop `GameImage`. Here we shall figure out the public methods these classes will need to have.

11.1 Method interface designs

`Game` will have the following public methods.

Method interfaces for class `Game`.

Method	Return	Arguments	Description
Constructor		`int`	Constructs a game with the given integer as the length of each side of the (square) field of cells.
`getScoreMessage`	`String`		Returns the message which will be placed in the score message bar. This message will typically be set from within the `Game` class, for example when the snake crashes into itself.

Method interfaces for class `Game`.			
Method	Return	Arguments	Description
`setScoreMessage`	`void`	`String`	This enables another object to actually set the score message. For example, this will be used by the instance of `GameGUI` when the game is paused.
`getAuthor`	`String`		Returns the name of the author of the `Game` class code. You will make this be your name, of course!
`getGridSize`	`int`		Returns the length of the sides of the field of cells.
`getGridCell`	`Cell`	`int, int`	This returns the cell in the grid, indexed by the two given integers, x and y. These integers must be legal grid indices.
`setInitialGameState`	`void`	`int, int, int, int`	This (re)initialises the game. The 4 integers are, in order, the x and y positions of the *tail* of the snake, the length of the snake, and the direction it is facing.
`setSnakeDirection`	`void`	`int`	This sets the direction of the snake to the one given.

Method interfaces for class `Game`.

Method	Return	Arguments	Description
`move`	`void`	`int`	This is the main behaviour of the game: a single move. This will move the snake, eat the food, check for crashes, etc.. The given integer is the basic value of the move (it is already proportional to the speed of the game interface). This value is used to calculate the real score of the move, which also depends on the number of trees, and the length of the snake.
`getScore`	`int`		Returns the total score so far this game.
`cheat`	`void`		This turns all bloody snake parts into non-bloody ones, so the game can continue. It also halves the score of the player.
`toggleTrees`	`void`		This switches on or off the trees feature of the game.
`optionalExtras`	`String`		This returns a string describing any optional extras existing in the game, so that it can be displayed in the help box of the game interface. The string should be split into lines, and describe the functionality of the key presses used.

Method interfaces for class Game.			
Method	Return	Arguments	Description
optionalExtraInterface	void	char	This method is passed all characters relating to keys pressed by the player which do not have a function in the game interface. This is so they can be used as optional extra features.

The public methods of the class GameImage include the following.

Method interfaces for class GameImage.			
Method	Return	Arguments	Description
Constructor		Game	Constructs a game image object, to provide an image for the given game.
update	void		Causes the image of the game to be updated.

12 Development of SpeedController and SpeedControllerImage

The next class in our development plan is SpeedController. The primary job of an object from this class is to offer a delay method which can be invoked after every move. The delay will be expressed as an integer, such that the bigger the number the longer will be the delay. An instance of a SpeedController will also have a current speed – the higher the speed, the shorter will be the delays it makes. This enables the game to have delays of different relative periods at different points, should that be required, but all of them will be shorter or longer depending on the current speed of the controller. We shall also want methods to increase and decrease the speed of a controller. In addition, we shall have methods to pause and resume the game, such that if a delay is asked for while the controller is in a paused state, the delay will not end until the controller is resumed.

Our intention is to permit the user to access the methods of the speed controller associated with a GameGUI via keys pressed by the player. However, as an extra convenience, we shall have a SpeedControllerImage class, and each SpeedController object shall have an associated

instance of `SpeedControllerImage`. This will display the speed of the controller and permit the user to invoke the methods via buttons.

We shall not look at the details of the speed controller and its image class here, you can just assume that we have developed them. Apart from the `delay()` method, the only other detail we need to know now is that a `SpeedController` must be given an initial speed when it is created. The class will offer three handy static final integer variables called `MIN_SPEED`, `HALF_SPEED` and `MAX_SPEED` which help the user initialise the speed of a controller without having to use actual numbers. This gives a more abstract facility, and guards against falling foul one day of a controller being replaced by one that has a different number of speeds. `HALF_SPEED` is intended to be a good choice for most cases. Other speeds can also be calculated. For example `HALF_SPEED - 1` is a little slower than `HALF_SPEED`, and a quarter speed is `(MIN_SPEED + HALF_SPEED) / 2)`.

12.1 Variable and method interface designs

We are not going to show all the details of the speed controller class at this stage, but here are its interesting public variables and methods.

Variable interfaces for class `SpeedController`.

Variable	Type	Description
`MIN_SPEED`	`int`	A static (class) variable indicating the number of the slowest speed.
`HALF_SPEED`	`int`	A static (class) variable indicating the speed which is half way between `MIN_SPEED` and `MAX_SPEED`.
`MAX_SPEED`	`int`	A static (class) variable indicating the number of the fastest speed.

Method interfaces for class `SpeedController`.

Method	Return	Arguments	Description
Constructor		`int`	Constructs a speed controller with the given integer as its initial speed.
`delay`	`void`	`int`	Causes a delay of time proportional to the given integer.

13 Development of `AboutBox`

The next class on our list is `AboutBox`. This will simply pop up a window containing a small piece of text, describing the program and how to use it. The details of the class are not interesting at this stage. For now it is sufficient to say that we will simply pass the text we wish the instance of `AboutBox` to include to its constructor, and from then on we can make the box appear and disappear at will. It will also have its own dismiss button to make it vanish.

14 Development of `GameGUI`

The final class before we reach the top level is `GameGUI`. Instances of this will create a game, a game image, a speed controller and an about box, and join them together with the control from the user, to play and show the game.

Most of the details of this class are best left out of this discussion, so we shall just list the interesting public methods here.

14.1 Method interface designs

The public methods of `GameGUI` will include the following.

Method interfaces for class `GameGUI`.			
Method	Return	Arguments	Description
Constructor		`int`	Constructs a game GUI, containing a game having a field with sides as long as the the given integer.
`playGame`	`int`		This repeatedly calls the `move()` method of the instance of `Game`, until the player presses the 'q' key.

15 Development of the top level class `Snake`

The top level class will contain just a main method. Its job will be to create an instance of `GameGUI` and invoke its `playGame()`.

```
public class Snake
```

```
{



  public static void main(String [] args)
  {
    GameGUI gameGUI = new GameGUI(18);
    int score = gameGUI.playGame();

    System.out.println("Score was " + score);
    System.exit(0);
  } // main

} // class Snake
```

16 Conclusion

Apart from the class you will write, we have now completed the development of our program. It consists of a total of 10 classes plus the one you will write. Ignoring comments and documentation in the code, those 10 classes consists of around 1300 lines of spaced out code. My sample answer for the part you will write, including the optional extras, consists of approximately 500 lines, making a total of approximately 1800 lines.

Before we finish, we ought to say something about the order of development we saw here. The choice we made, of considering the classes in a bottom up order, was more motivated by the wish for you to be able to follow it, than a realistic reconstruction of a real development experience. In reality, when we develop programs, we usually have to work on several classes at once. So, for example, if we try to follow a bottom up order after having had a top down high level analysis, we typically need to go back to classes we have already developed in order to add new features or modify existing ones, as we traverse up the dependencies towards the top. This is less necessary in the top down development approach, but instead we initially need to make stubs of our lower level classes.

Finally, we should address an obvious question, which you may well have been already asking yourself. Why did we not divide the `Game` class into separate classes such as, `Field`, `Food`, `Tree`, `GameSnake`, and so on? The short answer is this: *you* are the one that will be looking at that part of the program, in your laboratory exercise! However, in that exercise you are requested to develop the `Game` class without dividing it up, and *then* ask yourself whether it would be a good idea to, and what are the pros and cons.

Index

AboutBox class, 6–8, 30
  description, 5
accessor method concept, 7, 20

bottom-up concept, 6

Cell class, 6–10, 15, 17, 19, 20
  description, 5
  method interface, 18, 19
  variable interface, 17
  class implementation, 20
  constructor
    implementation, 20
  CLEAR variable, 15
  clone(), 19, 24
    implementation, 24
  equals(), 19, 24
    implementation, 24
  FOOD variable, 17
  getOtherLevel()
    implementation, 23
  getSnakeInDirection()
    implementation, 24
  getSnakeOutDirection()
    implementation, 24
  getType()
    implementation, 20
  isSnakeBloody()
    implementation, 23
  isSnakeType()
    implementation, 21
  NEW variable, 15, 18, 20
  OTHER variable, 17
  otherLevel variable, 17
  setClear()
    implementation, 21
  setFood()
    implementation, 22
  setOther()
    implementation, 23
  setSnakeBloody()
    implementation, 23
  setSnakeBody()
    implementation, 22
  setSnakeHead()
    implementation, 21
  setSnakeInDirection()
    implementation, 24
  setSnakeOutDirection()
    implementation, 24
  setSnakeTail()
    implementation, 22
  setTree()
    implementation, 23
  SNAKE_BODY variable, 16
  SNAKE_HEAD variable, 15, 16
  SNAKE_TAIL variable, 16
  snakeBloody variable, 15, 16
  snakeInDirection variable, 15–17
  snakeOutDirection variable, 15, 16
  TREE variable, 17
CellImage class, 9, 10, 15, 19, 24
  description, 9
  method interface, 19
  getPreferredSize(), 19
  paint(), 19
  update(), 10, 19
cheat() (Game), 7
  interface, 27
Class AboutBox, 5–8, 30
Class Cell, 5–10, 15, 17, 19, 20
Class CellImage, 9, 10, 15, 19, 24
Class Dimension, 19
Class Direction, 5–8, 10–13
Class Field, 31
Class Food, 31
Class Game, 5–10, 25, 26, 30, 31
Class GameGUI, 5–10, 26, 28, 30
Class GameImage, 9, 10, 19, 25, 28
Class GameSnake, 31
Class Snake, 5–8, 30
Class SpeedController, 5–8, 28, 29
Class SpeedControllerImage, 10, 28, 29
Class Tree, 31
CLEAR variable (Cell), 15
clone() (Cell), 19, 24
  interface, 19
  implementation, 24
Concept **accessor method**, 7, 20

Concept **bottom-up**, 6
Concept **control**, 8, 9
Concept **M.V.C.**, 8
Concept **model**, 8, 9
Concept **mutator method**, 7, 21
Concept **top-down**, 6
Concept **view**, 8, 9
control concept, 8, 9

delay() (SpeedController), 7, 29
  interface, 29
Dimension class, 19
Direction class, 6–8, 10–13
  description, 5
  method interface, 12, 13
  variable interface, 12
  class implementation, 13
  EAST variable, 14
  leftTurn()
    implementation, 15
  NORTH variable, 14
  opposite()
    implementation, 13
  rightTurn()
    implementation, 14
  SOUTH variable, 14
  WEST variable, 14
  xDelta(), 7
    implementation, 14
  yDelta(), 7
    implementation, 14

EAST variable (Direction), 14
  interface, 12
equals() (Cell), 19, 24
  interface, 19
  implementation, 24

Field class, 31
FOOD variable (Cell), 17
Food class, 31

Game class, 5–10, 25, 26, 30, 31
  description, 5
  method interface, 25–28
  cheat(), 7
  getScore(), 7
  move(), 7, 30
  optionalExtraInterface(), 6
  setSnakeDirection(), 7
  toggleTrees(), 7
GameGUI class, 5–10, 26, 28, 30
  description, 5
  method interface, 30
  playGame(), 6, 30
GameImage class, 9, 10, 19, 25, 28
  description, 9
  method interface, 28
  update(), 10, 19
GameSnake class, 31
getAuthor() (Game)
  interface, 26
getGridCell() (Game)
  interface, 26
getGridSize() (Game)
  interface, 26
getOtherLevel() (Cell)
  interface, 18
  implementation, 23
getPreferredSize() (CellImage), 19
  interface, 19
getScore() (Game), 7
  interface, 27
getScoreMessage() (Game)
  interface, 25
getSnakeInDirection() (Cell)
  interface, 18
  implementation, 24
getSnakeOutDirection() (Cell)
  interface, 18
  implementation, 24
getType() (Cell)
  interface, 18
  implementation, 20

HALF_SPEED variable (SpeedController), 29
  interface, 29

int variable (Cell)
  interface, 17
isSnakeBloody() (Cell)
  interface, 18
  implementation, 23
isSnakeType() (Cell)
  interface, 18
  implementation, 21

leftTurn() (Direction)
  interface, 13
  implementation, 15

M.V.C. concept, 8
main() (Snake)
  implementation, 31
MAX_SPEED variable (SpeedController), 29
  interface, 29
MIN_SPEED variable (SpeedController), 29
  interface, 29
model concept, 8, 9
move() (Game), 7, 30
  interface, 27
mutator method concept, 7, 21

NEW variable (Cell), 15, 18, 20
NONE variable (Direction)
  interface, 12
NORTH variable (Direction), 14
  interface, 12

opposite() (Direction)
  interface, 12
  implementation, 13
optionalExtraInterface() (Game), 6
  interface, 28
optionalExtras() (Game)
  interface, 27
OTHER variable (Cell), 17
otherLevel variable (Cell), 17

paint() (CellImage), 19
  interface, 19
pause() (SpeedController), 7
playGame() (GameGUI), 6, 30
  interface, 30

resume() (SpeedController), 7
rightTurn() (Direction)
  interface, 13
  implementation, 14

setClear() (Cell)
  interface, 18
  implementation, 21
setFood() (Cell)
  interface, 18
  implementation, 22
setInitialGameState() (Game)
  interface, 26
setOther() (Cell)
  interface, 18
  implementation, 23
setScoreMessage() (Game)
  interface, 26
setSnakeBloody() (Cell)
  interface, 18
  implementation, 23
setSnakeBody() (Cell)
  interface, 18
  implementation, 22
setSnakeDirection() (Game), 7
  interface, 26
setSnakeHead() (Cell)
  interface, 18
  implementation, 21
setSnakeInDirection() (Cell)
  interface, 18
  implementation, 24
setSnakeOutDirection() (Cell)
  interface, 18
  implementation, 24
setSnakeTail() (Cell)
  interface, 18
  implementation, 22
setTree() (Cell)
  interface, 18
  implementation, 23
slowDown() (SpeedController), 7
Snake class, 6–8, 30
  description, 5
  class implementation, 30
  main()
    implementation, 31
SNAKE_BODY variable (Cell), 16
SNAKE_HEAD variable (Cell), 15, 16
SNAKE_TAIL variable (Cell), 16
snakeBloody variable (Cell), 15, 16
snakeInDirection variable (Cell), 15–17
snakeOutDirection variable (Cell), 15, 16
SOUTH variable (Direction), 14
  interface, 12
SpeedController class, 6–8, 28, 29
  description, 5
  method interface, 29

variable interface, 29
delay(), 7, 29
HALF_SPEED variable, 29
MAX_SPEED variable, 29
MIN_SPEED variable, 29
pause(), 7
resume(), 7
slowDown(), 7
speedUp(), 7
SpeedControllerImage class, 28, 29
description, 10
speedUp() (SpeedController), 7

toggleTrees() (Game), 7
interface, 27
top-down concept, 6
TREE variable (Cell), 17
Tree class, 31

update() (CellImage), 10, 19
interface, 19
update() (GameImage), 10, 19
interface, 28

view concept, 8, 9

WEST variable (Direction), 14
interface, 12

xDelta() (Direction), 7
interface, 12
implementation, 14

yDelta() (Direction), 7
interface, 13
implementation, 14