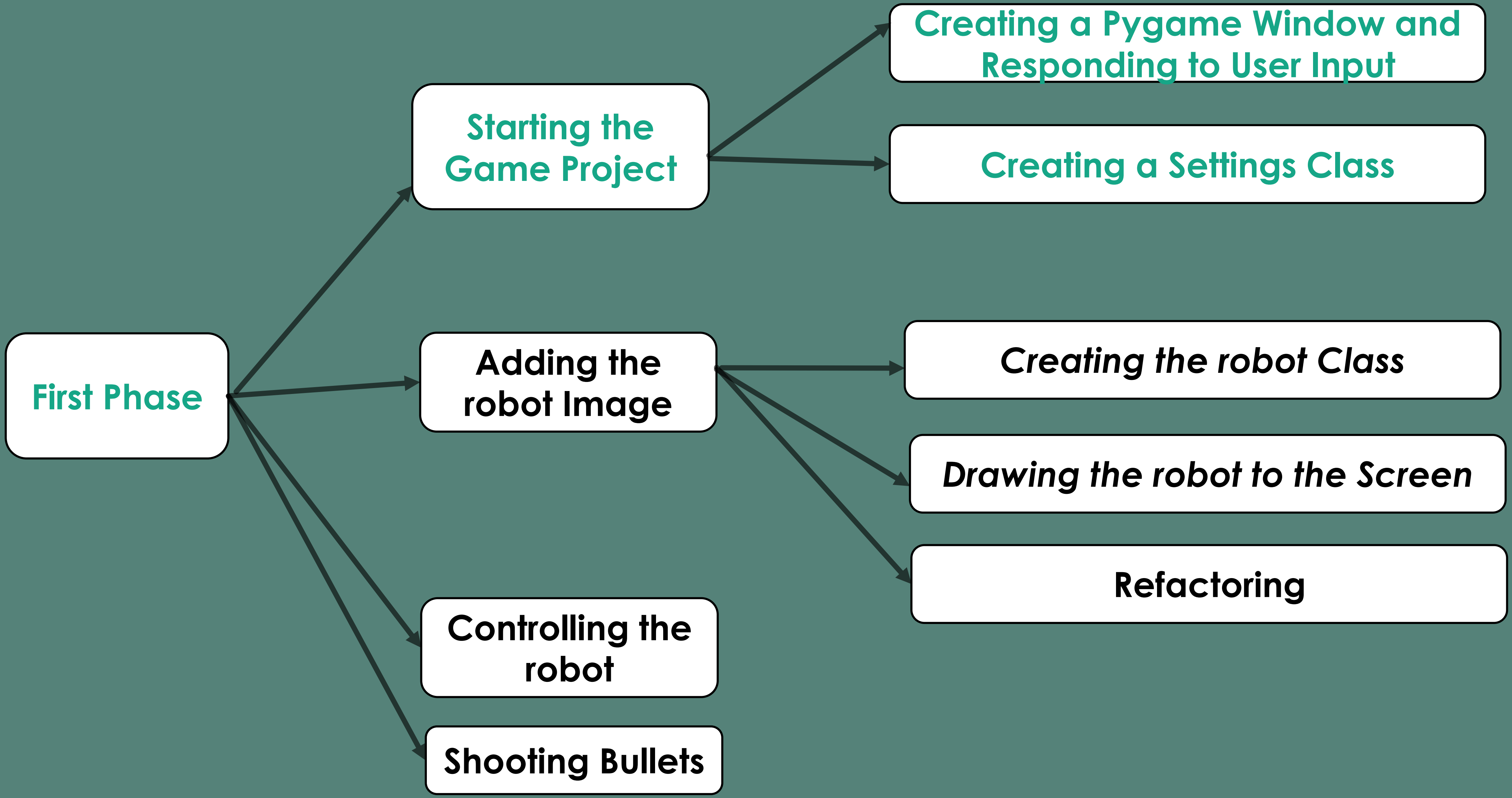
OBJECT ORIENTED PROGRAMMING WITH PYTHON

PROJECT BASED

2nd semester (Lect5)

Dr. Ann Al-Kazzaz

FIRST PHASE


First Phase
Starting the Game Project
Adding the robot Image
Controlling the robot
Shooting Bullets
Creating a Pygame Window and Responding to User Input
Creating a Settings Class
Creating the robot Class
Drawing the robot to the Screen
Refactoring

ADD THE ROBOT IMAGE TO OUR GAME

To draw the player’s robot on the screen, we’ll load an image and then use the Pygame **blit()** method to draw the image.

Notes:
1- When you’re choosing artwork for your games, be sure to pay attention to licensing. The safest and cheapest way to start is to use **freely licensed graphics** that you can use and modify, from a website like *https://pixabay.com/*.

2- Pay particular attention to the **background color** in your chosen image. Try to find a file with a transparent or solid background that you can replace with any background color using an image editor. Your games will look best if the image’s background color matches your game’s background color. Alternatively, you can match your game’s background to the image’s background.

Now, Make a folder called **images** inside your main **Virus_ Invasion** project folder. Save the file **robot.bmp** in the images folder

CREATING ROBOT CLASS

Pygame is efficient because it lets you treat all game elements like rectangles (*rects*), even if they're not exactly shaped like rectangles. Treating an element as a rectangle is efficient because rectangles are simple geometric shapes. When Pygame needs to figure out whether two game elements have collided.

This approach usually works well enough that no one playing the game will notice that we're not working with the exact shape of each game element. We'll treat the robot and the screen as rectangles in this class.

CREATING ROBOT CLASS

Now we need to display the robot image on the screen. To use our robot, we'll create a new robot module (**robot.py**) that will contain the class robot. This class will manage most of the behavior of the player's robot

```
import pygame
class Robot:
"""A class to manage the robot."""
def __init__(self, vi_game):
    """Initialize the robot and set its starting position."""
1   self.screen = vi_game.screen
2   self.screen_rect = vi_game.screen.get_rect()
     # Load the robot image and get its rect.
3   self.image = pygame.image.load('images/robot.bmp')
    self.rect = self.image.get_rect()

    # Start each new robot at the bottom center of the screen.
4   self.rect.midbottom = self.screen_rect.midbottom
5def blitme(self):
    """Draw the robot at its current location."""
    self.screen.blit(self.image, self.rect)
```

CREATING ROBOT CLASS

Attributes

The __init__() method of robot takes **two parameters**: the self reference and a reference to the current instance of the VirusInvasion class. This will give robot access to all the game resources defined in VisrusInvasion. when creating a **Robot** object in the **VirusInvasion** class, such as **self.robot = Robot(self)**, the **self** parameter represents the instance of **VirusInvasion**. This instance is passed as an argument to the **Robot** constructor, and it is assigned to the parameter **vi_game**

1 we assign the **screen** to an attribute of **robot**, so we can access it easily in all the methods in this class.

2 we access the **screen's rect attribute** using the **get_rect()** method and assign it to self.screen_rect. Doing so allows us to place the robot in the correct location on the screen.

3 To load the image, we call **pygame.image.load()**, and give it the location of our robot image. This function returns a surface representing the robot, which we assign to **self.image**. When the image is loaded, we call **get_rect()** to access the robot surface's rect attribute so we can later use it to place the robot.

4 We'll position the robot at the bottom center of the screen. To do so, make the value of **self.rect.midbottom** match the **midbottom** attribute of the screen's rect.

Method

5 we define the **blitme()** method, which draws the image to the screen at the position specified by **self.rect**.

CREATING ROBOT CLASS

Notes about the rect object

When you’re working with a rect object,
you can use the **x- and y-coordinates** of the top, bottom, left, and right edges of the rectangle, as well as the center, to place the object. You can set any of these values to establish the current position of the rect.

When you’re **centering a game element**, work with the center, **centerx**, or **centery** attributes of a rect.

When you’re **working at an edge** of the screen, work with the **top, bottom, left**, or **right** attributes.

There are also attributes that combine these properties, such as **midbottom, midtop, midleft**, and **midright**.

When you’re **adjusting the horizontal or vertical placement of the rect**, you can just use the **x** and **y** attributes, which are **the x- and y-coordinates of its top-left corner.**

*In Pygame, **the origin (0, 0) is at the top-left corner of the screen**, and coordinates increase as you go down and to the right. On a 1200 by 800 screen, the origin is at the top-left corner, and the bottom-right corner has the coordinates (1200, 800). **These coordinates refer to the game window, not the physical screen**.*

DRAWING THE ROBOT TO THE SCREEN

Now let's update ***Virus_ Invasion.py*** so it creates a robot and calls the robot's **blitme()** method:

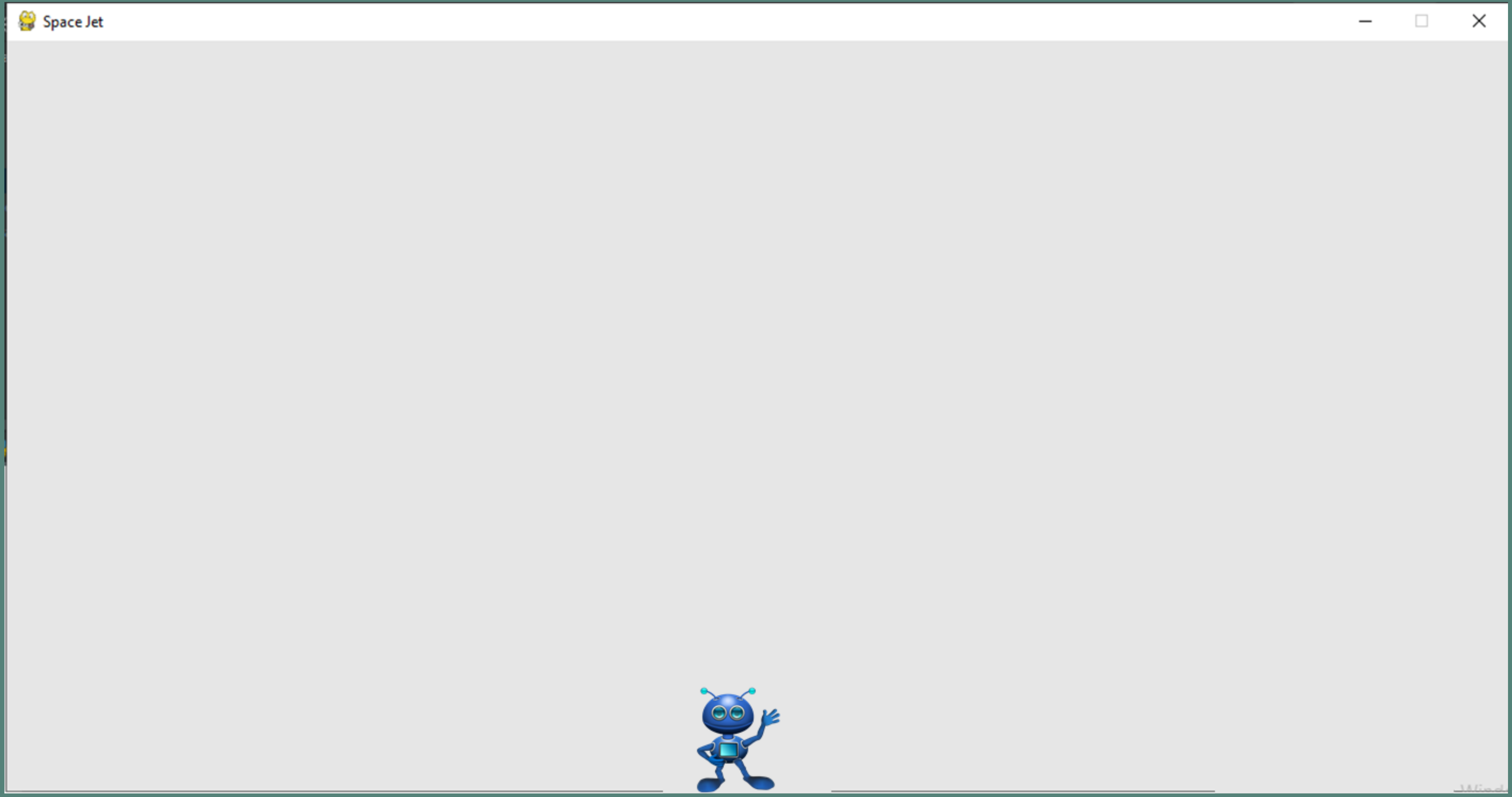
```
--snip--
from settings import Settings
from robot import Robot
class VirusInvasion:
"""Overall class to manage game assets and behavior."""
    def __init__(self):
    --snip--
    pygame.display.set_caption("VirusInvasion")
1   self.robot = Robot(self)
    def run_game(self):
    --snip--
        # Redraw the screen during each pass through the loop.
        self.screen.fill(self.settings.bg_color)
2       self.robot.blitme()
        # Make the most recently drawn screen visible.
        pygame.display.flip()
--snip--
```

1 The call to **robot()** requires one argument, an instance of **VirusInvasion**. . The self argument here refers to the current instance of **VirusInvasion**. This is the parameter that gives robot access to the game's resources, such as the screen object. We assign this robot instance to **self.robot**.

2 After filling the background, we draw the robot on the screen by calling **robot.blitme()**, so the robot appears on top of the background

DRAWING THE ROBOT TO THE SCREEN

When you run *Virus_ Invasion.py* now, you should see an empty game screen with the rocket robot sitting at the bottom center



REFACTORING:

The _check_events() and _update_screen() Methods

- In large projects, you'll often refactor code you've written before adding more code.
- Refactoring simplifies the structure of the code you've already written, making it easier to build on.
- In this Lecture, we'll break the **run_game() method**, which is getting lengthy, into two helper methods.
- A ***helper method*** does work inside a class but isn't meant to be called through an instance. In Python, a single leading underscore indicates a helper method.

The _check_events() Method

Isolating the event loop allows you to manage events separately from other aspects of the game, such as updating the screen.

REFACTORING

_check_events()

Virus_ Invasion.py

```
def run_game(self):
"""Start the main loop for the game."""
      while True:
2             self._check_events()
      # Redraw the screen during each pass through the loop.
       --snip—
1 def _check_events(self):
"""Respond to keypresses and mouse events."""
     for event in pygame.event.get():
          if event.type == pygame.QUIT:
              sys.exit()
```

1 We make a **new _check_events()** method and move the lines that check whether the player has clicked to close the window into this new method.

2 To call a method from within a class, use dot notation with the variable self and the name of the method . We call the method from inside the while loop in run_game().

REFACTORING

To further simplify run_game(), we'll move the code for updating the screen to a separate method called ***_update_screen()***

Virus_ Invasion.py

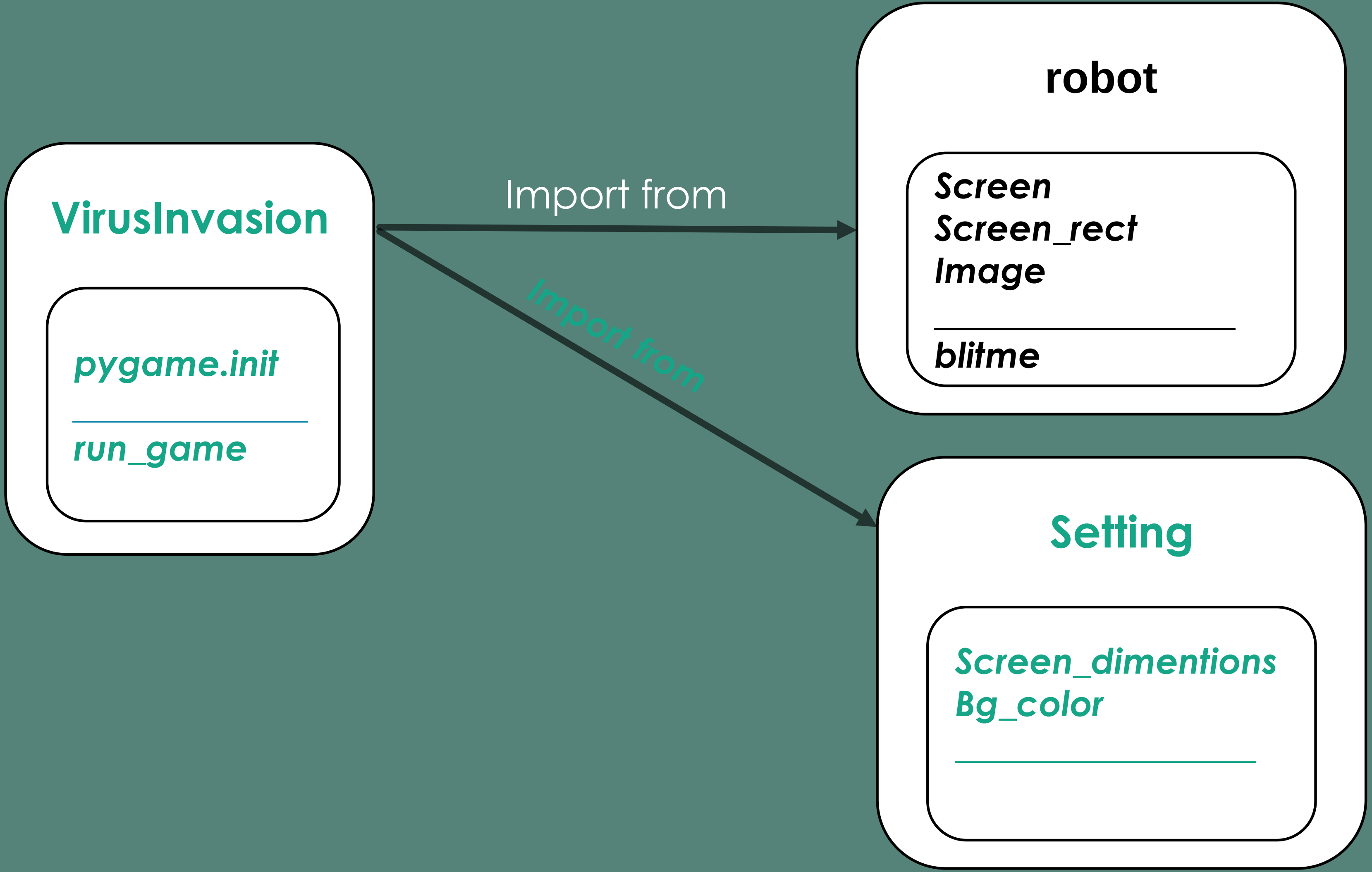
```
def run_game(self):
"""Start the main loop for the game."""
    while True:
        self._check_events()
        self._update_screen()
    # Redraw the screen during each pass through the loop.
     --snip—
def _check_events(self):
--snip—
def _update_screen(self):
"""Update images on the screen, and flip to the new screen."""
    self.screen.fill(self.settings.bg_color)
    self.robot.blitme()

    pygame.display.flip()
```

We moved the code that draws the background and the robot and flips the screen to _update_screen(). Now the body of the main loop in run_game() is much simpler.

Now that we've restructured the code to make it easier to add to, we can work on the dynamic aspects of the game!

CLASSES DIAGRAM


VirusInvasion
pygame.init
run_game
Import from
Import from
robot
Screen
Screen_rect
Image
blitme
Setting
Screen_dimentions
Bg_color