OBJECT ORIENTED PROGRAMMING WITH PYTHON

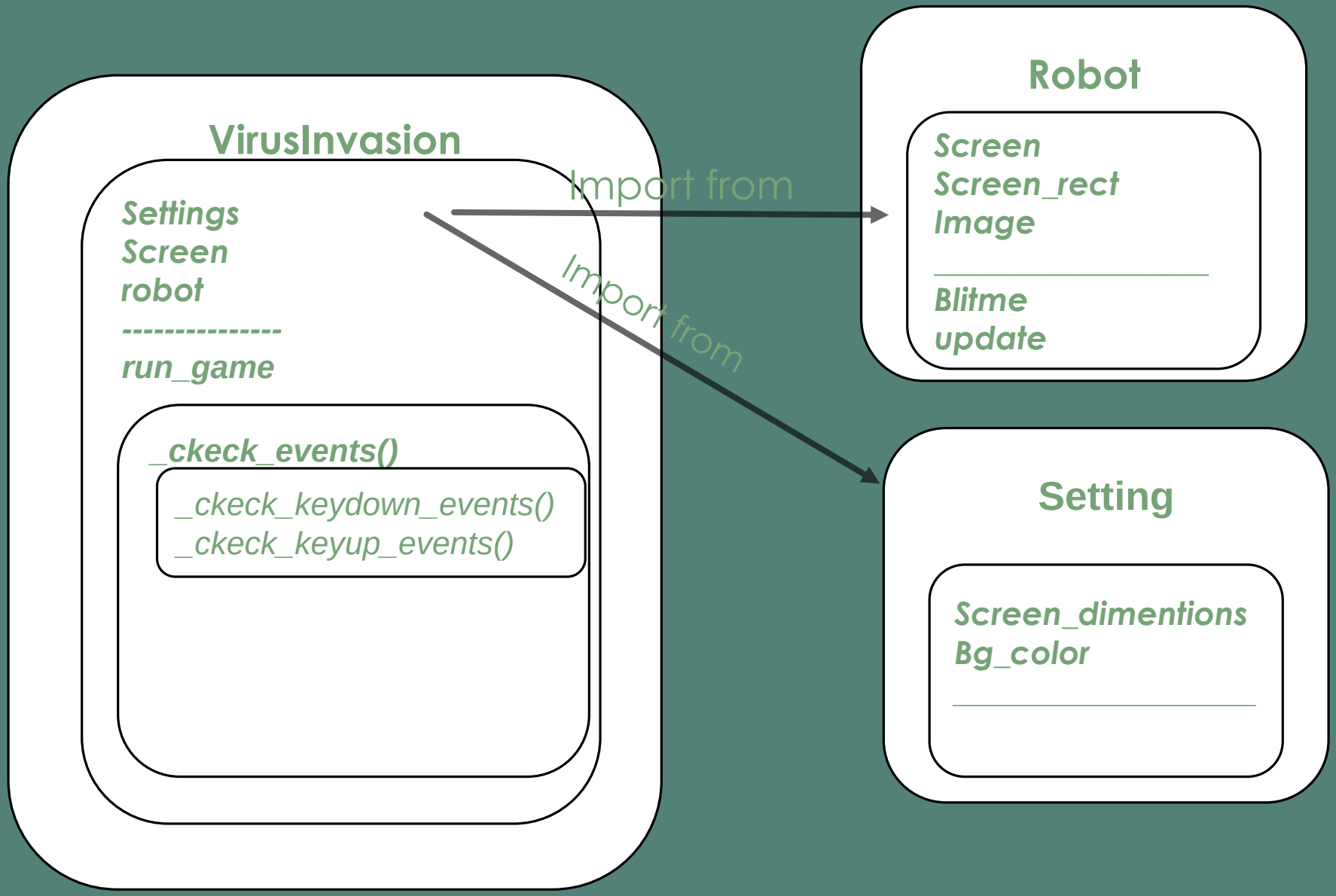
PROJECT BASED

2nd semester (Lect6)

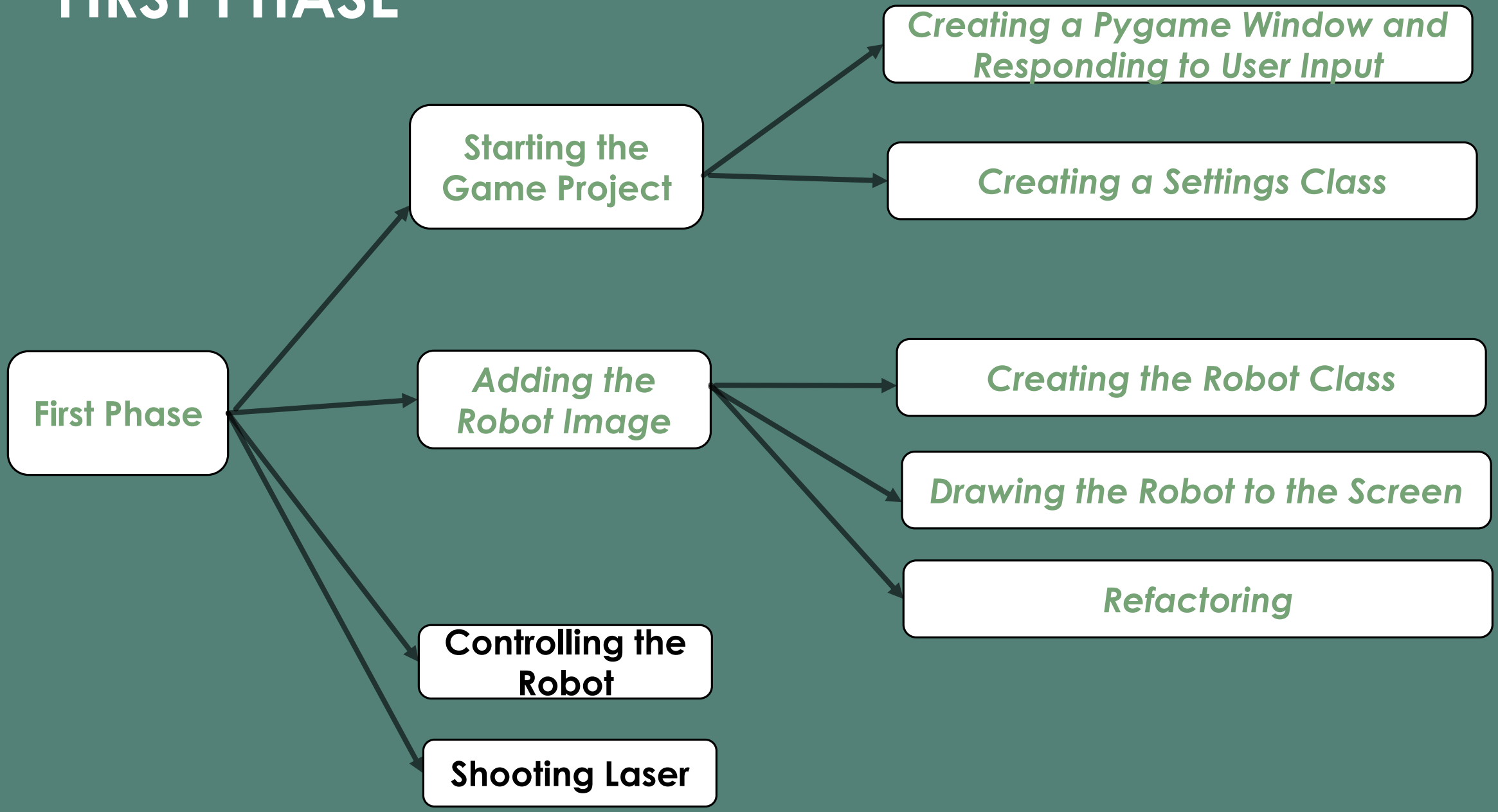
Dr. Ann Al-Kazzaz

- *Controlling the robot*
- Allowing Continuous Movement
- Moving Both Left and Right
- Adjusting the Robot's Speed
- *Limiting the Robot's Range*
- *Refactoring*
- *Pressing Q to Quit*

CLASSES DIAGRAM



FIRST PHASE


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

CONTROLLING THE ROBOT

- Next, we'll give the player the ability to move the robot right and left.
- We'll write code that responds when the player presses the right or left arrow key.
- We'll focus on movement to the right first, and then we'll apply the same principles to control movement to the left.
- As we add this code, you'll learn **how to control the movement of images on the screen and respond to user input**.

Responding to a Keypress

- Whenever the player presses a key, that keypress is registered in Pygame as an event.
- Each event is picked up by the **pygame.event.get()** method

We need to specify in our **_check_events()** method what kind of events we want the game to check for.

- Each **keypress** is registered as a **KEYDOWN** event.
- When Pygame detects a **KEYDOWN** event, we need to check whether the key that was **pressed** is one that triggers a certain action.
- if the player presses the right arrow key, we want to increase the **robot's rect.x** value to move the robot to the right

CONTROLLING THE ROBOT

_check_events()

virus_invasion.py

```
def _check_events(self):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                # Move the robot to the right.
                self.robot.rect.x += 1
```

1. Inside **_check_events()** we add an **elif** block to the event loop to respond when Pygame detects a **KEYDOWN** event. We check whether the key pressed, **event.key**, is the right arrow key

2. The right arrow key is represented by pygame.K_RIGHT. If the right arrow key was pressed,

3. we move the robot to the right by increasing the value of self.robot.rect.x by 1

When you run **virus_invasion.py** now, the robot should move to the right one pixel every time you press the right arrow key. That's a start, but it's not an efficient way to control the robot. Let's improve this control by allowing continuous movement.

ALLOWING CONTINUOUS MOVEMENT

- When the player holds down the right arrow key, we want the robot to continue moving right until the player releases the key.
- We'll have the game detect a **pygame.KEYUP** event so we'll know when the right arrow key is Released.
- Then we'll use the **KEYDOWN** and **KEYUP** events together with a **flag** called **moving_right** to implement continuous motion.
- When the **moving_right** flag is **False**, the robot will be **motionless**. When the player **presses the right arrow key**, we'll set the flag to **True**, and when the player **releases the key**, we'll set the flag to **False again**.
- The **Rrobot class** controls all attributes of the robot, so we'll give it an **attribute** called **moving_right** and an **update() method** to check the status of the **moving_right flag**.
- The **update()** method will change the position of the robot if the flag is set to True.
- We'll call this method once on each pass through the while loop to update the position of the robot.

ALLOWING CONTINUOUS MOVEMENT

robot.py

```
class Robot:
    """A class to manage the Robot."""
    def __init__(self, vi_game):
        --snip--
        #Start each new Robot at the bottom center of the screen.
        self.rect.midbottom = self.screen_rect.midbottom
        # Movement flag
1       self.moving_right = False
2   def update(self):
        #Update the robot's position based on the movement flag.
3       if self.moving_right:
            self.rect.x += 1
    def blitme(self):
    --snip--
```

1. We add a self.moving_right attribute in the __init__() method and set it to False initially.
2. Then we add update(), which moves the robot right if the flag is True

The update() method will be called through an instance of Robot, so it's not considered a helper method.

ALLOWING CONTINUOUS MOVEMENT

Now we need to modify **_check_events()** so that **moving_right** is set to **True** when the right arrow key is pressed and False when the key is released:

virus_invasion.py

```
def _check_events(self):
        """Respond to keypresses and mouse events."""
        for event in pygame.event.get():
            --snip--
            elif event.type == pygame.KEYDOWN:
                if event.key == pygame.K_RIGHT:
1                   self.robot.moving_right = True
2           elif event.type == pygame.KEYUP:
                if event.key == pygame.K_RIGHT:
                    self.robot.moving_right = False
```

1. we modify how the game responds when the player presses the right arrow key: instead of changing the robot's position directly, we merely set moving_right to True.

2. we add a new elif block, which responds to KEYUP events. When the player releases the right arrow key (K_RIGHT), we set moving_right to False.

ALLOWING CONTINUOUS MOVEMENT

Next, we modify the while loop in **run_game()** so it calls the **robot's update() method** on each pass through the loop:

virus_invasion.py

```
def run_game(self):
    """Start the main loop for the game."""
    while True:
        self._check_events()
        self.robot.update()
        self._update_screen()
```

The robot's position will be updated after we've checked for keyboard events and before we update the screen. This allows the robot's position to be updated in response to player input and ensures the updated position will be used when drawing the robot to the screen.

When you run *virus_invasion.py* and hold down the right arrow key, the robot should move continuously to the right until you release the key.

MOVING BOTH LEFT AND RIGHT

Now that the robot can move continuously to the right, adding movement to the left is straightforward. Again, we'll modify the robot class and the _check _events() method. Here are the relevant changes to __init__() and update() in robot:

robot.py

```
def __init__(self, vi_game):
    --snip--
    # Movement flags
    self.moving_right = False
    self.moving_left = False
def update(self):
    """Update the robot's position based on movement flags."""
    if self.moving_right:
        self.rect.x += 1
    if self.moving_left:
        self.rect.x -= 1
```

- In __init__(), we add a self.moving_left flag.
- In update(), we use two separate if blocks rather than an elif to **allow the robot's rect.x value to be increased and then decreased when both arrow keys are held down**. This results in the robot standing still
- If we used elif for motion to the left, the right arrow key would always have priority.

MOVING BOTH LEFT AND RIGHT

Doing it this way makes the movements more accurate when switching from right to left when the player might momentarily hold down both keys.
We have to make two adjustments to _check_events():

virus_invasion.py

```
def _check_events(self):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        --snip--
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_RIGHT:
                self.robot.moving_right = True
            elif event.key == pygame.K_LEFT:
                self.robot.moving_left = True
        elif event.type == pygame.KEYUP:
            if event.key == pygame.K_RIGHT:
                self.robot.moving_right = False
            elif event.key == pygame.K_LEFT:
                self.robot.moving_left = False
```

If a **KEYDOWN** event occurs for the **K_LEFT** key, we set **moving_left** to True. If a **KEYUP** event occurs for the **K_LEFT** key, we set **moving_left** to False. We can use elif blocks here because each event is connected to only one key. If the player presses both keys at once, two separate events will be detected.

ADJUSTING THE ROBOT'S SPEED

Currently, the robot moves one pixel per cycle through the while loop, but we can take finer control of the robot's speed by adding a robot_speed attribute to the Settings class. We'll use this attribute to determine how far to move the robot on each pass through the loop

setting.py

```
class Settings:
    """A class to store all settings for Alien Invasion."""
    def __init__(self):
    --snip--
     # robot settings
     self.robot_speed = 1.5
```

- We set the initial value of robot_speed to 1.5. When the robot moves now, its position is adjusted by 1.5 pixels rather than 1 pixel on each pass through the loop.

- We're using decimal values for the speed setting to give us finer control of the robot's speed when we increase the tempo of the game later on.
- However, rect attributes such as x store only integer values, so we need to make some modifications to robot:

ADJUSTING THE ROBOT'S SPEED

robot.py

```
class Robot:
    """A class to manage the Robot."""
1   def __init__(self, vi_game):
        """Initialize the Robot and set its starting position."""
        self.screen = vi_game.screen
        self.settings = vi_game.settings
        --snip--
        # Start each new Robot at the bottom center of the screen.
        --snip--
        # Store a decimal value for the Robot's horizontal position.
2       self.x = float(self.rect.x)
        # Movement flags
        self.moving_right = False
        self.moving_left = False
    def update(self):
        """Update the Robot's position based on movement flags."""
            # Update the Robot's x value, not the rect.
            if self.moving_right:
3               self.x += self.settings.robot_speed
            if self.moving_left:
                self.x -= self.settings.robot_speed
            # Update rect object from self.x.
4           self.rect.x = self.x
     def blitme(self):
      --snip--
```

ADJUSTING THE ROBOT'S SPEED

robot.py

1. We create a settings attribute for robot, so we can use it in update()

2. Because we're adjusting the position of the robot by fractions of a pixel, we need to assign the position to a variable that can store a decimal value. You can use a decimal value to set an attribute of rect, but the rect will only keep the integer portion of that value. To keep track of the robot's position accurately, we define a new **self.x** attribute that can hold decimal values.
   We use the float() function to convert the value of self.rect.x to a decimal and assign this value to self.x.

3. Now when we change the robot's position in update(), the value of self.x is adjusted by the amount stored in settings.robot_speed w. After self.x has been updated, we use the new value to update self.rect.x, which controls the position of the robot.

4. Only the integer portion of self.x will be stored in self.rect.x, but that's fine for displaying the robot.

Now we can change the value of robot_speed, and any value greater than one will make the robot move faster. This will help make the robot respond quickly enough to shoot down aliens, and it will let us change the tempo of
the game as the player progresses in gameplay.

LIMITING THE ROBOT'S RANGE

Now, the robot will disappear off either edge of the creen if you hold down an arrow key long enough. We do this by modifying the update() method in robot:

robot.py

```
def update(self):
    """Update the robot's position based on movement flags."""
    # Update the robot's x value, not the rect.
1   if self.moving_right and self.rect.right < self.screen_rect.right:
        self.x += self.settings.robot_speed
2   if self.moving_left and self.rect.left > 0:
        self.x -= self.settings.robot_speed
    # Update rect object from self.x.
    self.rect.x = self.x
```

- This code checks the position of the robot before hanging the value of self.x.

1 The code **self.rect.right** returns the x-coordinate of the right edge of the robot's rect.

- This code checks the position of the robot before changing the value of self.x. The code self.rect.right returns the x-coordinate of the right edge of the robot's rect.

2 The same goes for the left edge: if the value of the left side of the rect is greater than zero, the robot hasn't reached the left edge of the screen

REFACTORING _CHECK_EVENTS()

virus_invasion.py

```
def _check_events(self):
    """Respond to keypresses and mouse events."""
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            self._check_keydown_events(event)
        elif event.type == pygame.KEYUP:
            self._check_keyup_events(event)
def _check_keydown_events(self, event):
    """Respond to keypresses."""
    if event.key == pygame.K_RIGHT:
        self.robot.moving_right = True
    elif event.key == pygame.K_LEFT:
        self.robot.moving_left = True
def _check_keyup_events(self, event):
    """Respond to key releases."""
    if event.key == pygame.K_RIGHT:
        self.robot.moving_right = False
    elif event.key == pygame.K_LEFT:
        self.robot.moving_left = False
```

The _check_events() method will increase in length as we continue to develop the game, so let's break _check_events() into two more methods: one that handles **KEYDOWN** events and another that handles **KEYUP** events:

REFACTORING _CHECK_EVENTS()

- We make two new helper methods:
  - **_check_keydown_events()**
  - **_check _keyup_events().**
- Each needs a self parameter and an event parameter.
- The bodies of these two methods are copied from _check_events(), and we've replaced the old code with calls to the new methods.

- The _check_events() method is simpler now with this cleaner code structure, which will make it easier to develop further responses to player input.

PRESSING Q TO QUIT

Now that we're responding to keypresses efficiently, we can add another way to quit the game. so we'll add a keyboard shortcut to end the game when the player presses **Q**:

virus_invasion.py

```
def _check_keydown_events(self, event):
    --snip--
    elif event.key == pygame.K_LEFT:
        self.robot.moving_left = True
    elif event.key == pygame.K_q:
        sys.exit()
```

In _check_keydown_events(), we add a new block that ends the game when the player presses Q

RUNNING THE GAME IN FULLSCREEN MODE

Pygame has a fullscreen mode that you might like better than running the game in a regular window To run the game in fullscreen mode, make the following changes in __init__():

virus_invasion.py

```
def __init__(self):
    """Initialize the game, and create game resources."""
    pygame.init()
    self.settings = Settings()
1   self.screen = pygame.display.set_mode((0, 0),       pygame.FULLSCREEN)
2   self.settings.screen_width = self.screen.get_rect().width
    self.settings.screen_height = self.screen.get_rect().height
    pygame.display.set_caption("Alien Invasion")
```

1. When creating the screen surface, we pass a size of (0, 0) and the parameter **pygame.FULLSCREEN** . This tells Pygame to figure out a window size that will fill the screen. Because we don't know the width and height of the screen ahead of time,

2. we update these settings after the screen is created . We use the width and height attributes of the screen's rect to update the settings object.