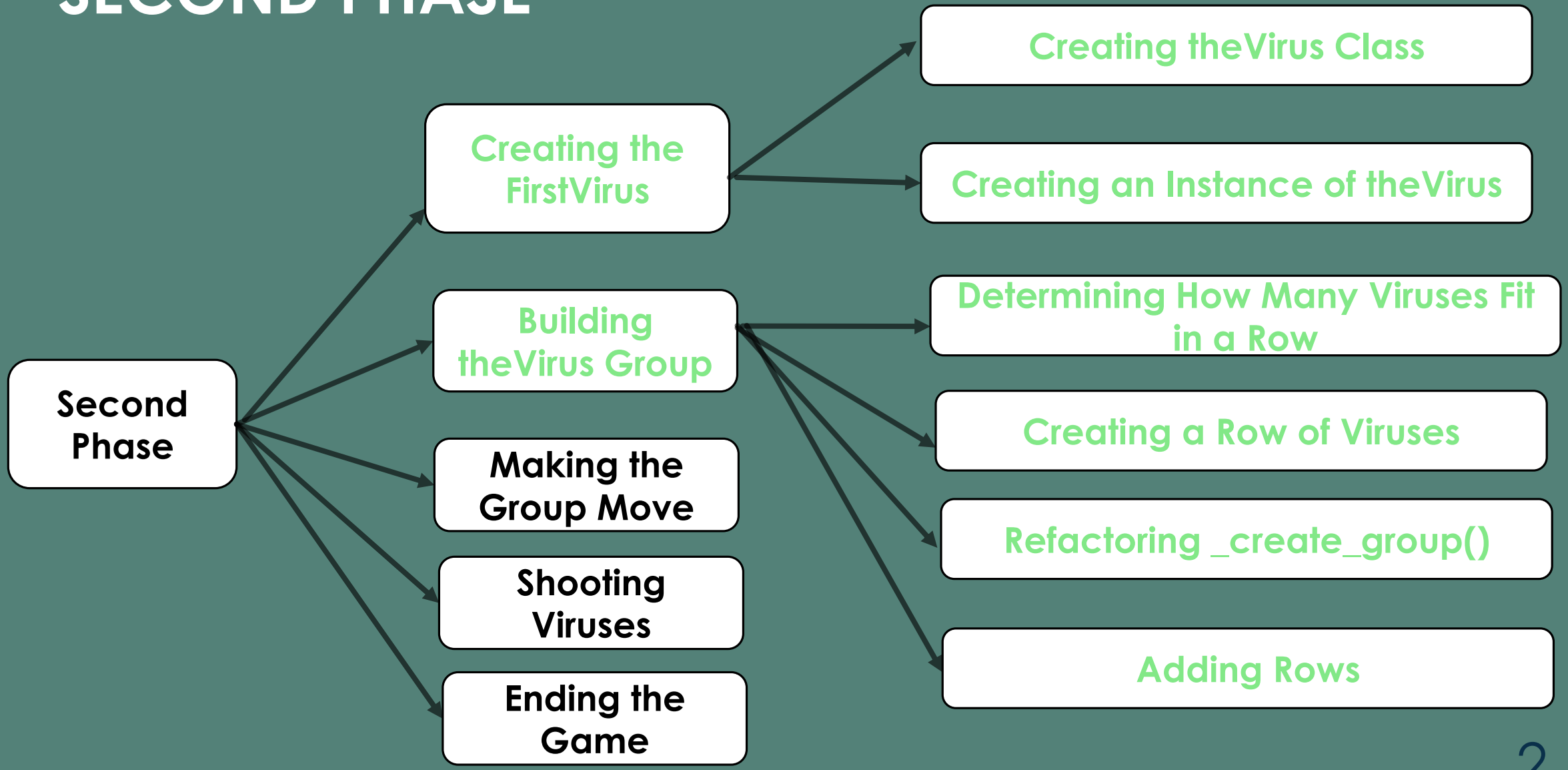
OBJECT ORIENTED PROGRAMMING WITH PYTHON

PROJECT BASED

2nd semester (Lect9)

Dr. Ann Al-Kazzaz

SECOND PHASE



MAKING THE GROUP MOVE

- Now let's make the group of viruses move to the right across the screen until it hits the edge, and then make it drop a set amount and move in the other direction. We'll continue this movement until all viruses have been shot down, one collides with the robot, or one reaches the bottom of the screen.

Moving the Viruses Right

- To move the viruses, we'll use an **update()** method in ***virus.py***, which we'll call for each virus in the group of viruses.
- **First**, add a setting to control the speed of each virus:

settings.py

```
def __init__(self):
    --snip--
    #Virus settings
    self.virus_speed = 1.0
```

MOVING THE VIRUSES RIGHT

- Then use this setting to implement **update()**:

Virus.py

```
def __init__(self, vi_game):
    """Initialize the virus and set its starting position."""
    super().__init__()
    self.screen = vi_game.screen
    self.settings = vi_game.settings
    --snip--
def update(self):
    """Move the virus to the right."""
1   self.x += self.settings.virus_speed
2   self.rect.x = self.x
```

1. Each time we update a virus's position, we move it to the right by the amount stored in **virus_speed**. We track the virus's exact position with the **self.x** attribute, which can hold decimal values

2. We then use the value of **self.x** to update the position of the **virus's rect.**

We create a settings parameter in **__init__()** so we can access the virus's speed in **update().**

MOVING THE VIRUSES RIGHT

- In the main while loop, we already have calls to update the robot and laser_beem positions. Now we'll add a call to update the position of each virus as well:

virusinvasion.py

```
while True:
    self._check_events()
    self.robot.update()
    self._update_laser_beems()
    self._update_viruses()
    self._update_screen()
```

- We're about to write some code to manage the movement of the group, so we create a new method called **_update_viruses().** We set the viruses' positions to update after the laser_beems have been updated, because we'll soon be checking to see whether any laser_beems hit any viruses.
- Where you place this method in the module is not critical

virusinvasion.py

```
def _update_viruses(self):
        """Update the positions of all viruses in the group."""
        self.viruses.update()
```

CREATING SETTINGS FOR GROUP DIRECTION

Now we'll create the settings that will make the group move down the screen and to the left when it hits the right edge of the screen. Here's how to implement this behavior:

settings.py

```
#Virus settings
self.virus_speed = 1.0
self.group_drop_speed = 10
# group_direction of 1 represents right; -1 represents left.
self.group_direction = 1
```

The setting **group_drop_speed** controls how quickly the group drops down the screen each time a virus reaches either edge. It's helpful to separate this speed from the viruses' horizontal speed so you can adjust the two speeds independently.

To implement the setting **group_direction**, let's use the values **1** and **−1**, and switch between them each time the group changes direction.

CHECKING WHETHER A VIRUS HAS HIT THE EDGE

We need a method to check whether a virus is at either edge, and we need to modify update() to allow each virus to move in the appropriate direction. This code is part of theVirus class:

Virus.py

```
def check_edges(self):
    """Return True if virus is at edge of screen."""
    screen_rect = self.screen.get_rect()
1   if self.rect.right >= screen_rect.right or self.rect.left <= 0:
        return True
def update(self):
    """Move the virus right or left."""
2    self.x += (self.settings.virus_speed * self.settings.group_direction)
     self.rect.x = self.x
```

1. The virus is at the right edge if the right attribute of its rect is greater than or equal to the right attribute of the screen's rect. It's at the left edge if its left value is less than or equal to 0

2. We modify the method update() to allow motion to the left or right by multiplying the virus's speed by the value of **group_direction**

We can call the new method **check_edges()** on any virus to see whether it's at the left or right edge.

DROPPING THE GROUP AND CHANGING DIRECTION

When a virus reaches the edge, the entire group needs to drop down and change direction. Therefore, we need to add some code to **VirusInvasion** because that's where we'll check whether any viruses are at the left or right edge. We'll make this happen by writing the methods **_check_group_edges()** and **_change_group_direction()**, and then modifying **_update_viruses()**.

virusinvasion.py

```
def _check_group_edges(self):
    """Respond appropriately if any viruses Laser beam have
reached an edge."""
1   for virus in self.viruses.sprites():
        if virus.check_edges():
2           self._change_group_direction()
            break
def _change_group_direction(self):
    """Drop the entire group and change the group's direction."""
    for virus in self.viruses.sprites():
3       virus.rect.y += self.settings.group_drop_speed
    self.settings.group_direction *= -1
```

1. In **_check_group_edges()**, we loop through the group and call **check_edges()** on each virus.

2. If **check_edges()** returns True, we know a virus is at an edge and the whole group needs to change direction; so we call **_change_group _direction**() and break out of the loop

3. In **_change_group_direction()**, we loop through all the viruses and drop each one using the setting **group_drop _speed**
   - then we change the value of **group_direction** by multiplying its current value by −1.

DROPPING THE GROUP AND CHANGING DIRECTION

Here are the changes to **_update_viruses():**

virusinvasion.py

```
def _update_viruses(self):
#Check if the group is at an edge, then update the positions of all viruses in the group.
    self._check_group_edges()
    self.viruses.update()
```

We've modified the method by calling **_check_group_edges()** before updating each virus's position.

When you run the game now, the group should move back and forth between the edges of the screen and drop down every time it hits an edge. Now we can start shooting down viruses and watch for any viruses that hit the robot or reach the bottom of the screen.

SHOOTING VIRUSES

when the laser_beams reach the viruses, they simply pass through because we aren't checking for collisions. In game programming, ***collisions*** happen when game elements overlap. To make the laser_beams shoot down viruses, we'll use the method **sprite.groupcollide()** to look for collisions between members of two groups.

Detecting Laser_beam Collisions

- We want to know right away when a laser_beam hits a virus so we can make a virus disappear as soon as it's hit. To do this, we'll look for collisions immediately after updating the position of all the laser_beams.

- The **sprite.groupcollide()** function compares the **rects** of each element in one group with the **rects** of each element in another group. In this case, it compares each **laser_beam's rect** with each **virus's rect** and **returns a dictionary containing the laser_beams and viruses that have collided.**
- Each key in the dictionary will be a laser_beam, and the corresponding value will be the virus that was hit(We'll also use this dictionary when we implement a scoring system).

- Add the following code to the end of **_update_laser_beams()** to check for collisions between laser_beams and viruses:

DETECTING LASER_BEAM COLLISIONS

virusinvasion.py

```
def _update_laser_beams(self):
    """Update position of laser_beams and get rid of old laser_beams."""
     --snip--
    # Check for any laser_beams that have hit viruses.
    # If so, get rid of the laser_beam and the virus.
    collisions = pygame.sprite.groupcollide( self.laser_beams, self.viruses, True, True)
```

- The new code we added compares the positions of all the laser_beams in **self.laser_beams** and all the viruses in **self.viruses**, and identifies any that overlap.
- Whenever the **rects** of a laser_beam and virus overlap, **groupcollide()** adds a key-value pair to the dictionary it returns. The two True arguments tell **Pygame** to delete the laser_beams and viruses that have collided.
- When you runVirus *Invasion* now, viruses you hit should disappear.

REPOPULATING THE GROUP

One key feature ofVirus *Invasion* is that the viruses are relentless: every time the group is destroyed, a new group should appear.
To make a new group of viruses appear after a group has been destroyed, we first check whether the viruses group is empty. If it is, we make a call to **_create_group().** We'll perform this check at the end of **_update_laser_beams(),** because that's where individual viruses are destroyed.

virusinvasion.py

```
def _update_laser_beams(self):
    --snip--
1   if not self.viruses:
  # Destroy existing laser_beams and create new group.
2       self.laser_beams.empty()
        self._create_group()
```

1. we check whether the viruses group is empty. An empty group evaluates to False, so this is a simple way to check whether the group is empty.

2. If it is, we get rid of any existing laser_beams by using the **empty()** method, which removes all the remaining sprites from a group. We also call **_create_group()**, which fills the screen with viruses again.

REFACTORING _UPDATE_LASER_BEEMS()

virusinvasion.py

```
def _update_laser_beems(self):
    --snip--
    # Get rid of laser_beems that have disappeared.
    for laser_beem in self.laser_beems.copy():
        if laser_beem.rect.bottom <= 0:
            self.laser_beems.remove(laser_beem)
    self._check_laser_beem_virus_collisions()

def _check_laser_beem_virus_collisions(self):
    """Respond to laser_beem-virus collisions."""
    # Remove any laser_beems and viruses that have collided.
    collisions = pygame.sprite.groupcollide( self.laser_beems, self.viruses, True, True)
    if not self.viruses:
        # Destroy existing laser_beems and create new group.
        self.laser_beems.empty()
        self._create_group()
```

ENDING THE GAME

DetectingVirus and Robot Collisions

We'll start by checking for collisions between viruses and the robot so we can respond appropriately when a virus hits it.
We'll check for virus and robot collisions immediately after updating the position of each virus in **VirusInvasion** :

virusinvasion.py

```
def _update_viruses(self):
    --snip--
    self.viruses.update()
    # Look for virus-robot collisions.
1   if pygame.sprite.spritecollideany(self.robot, self.viruses):
2       print("Robot hit!!!")
```

1. If no collisions occur, **spritecollideany()** returns None

2. If it finds a virus that has collided with the robot, it returns that virus and the if block executes: it prints *Robot hit!!!*

- The **spritecollideany()** function takes two arguments: a sprite and a group. The function looks for any member of the group that has collided with the sprite and stops looping through the group as soon as it finds one member that has collided with the sprite. Here, it loops through the group viruses and returns the first virus it finds that has collided with robot.

RESPONDING TO VIRUS AND ROBOT COLLISIONS

- Now we need to figure out exactly what will happen when a virus collides with the robot. Instead of destroying the robot instance and creating a new one, we'll count how many times the robot has been hit by tracking statistics for the game. Tracking statistics will also be useful for scoring. Let's write a new class, **GameStats**, to track game statistics, and save it as
- *game_stats.py*:

game_stats.py

```
class GameStats:
    """Track statistics forVirus Invasion."""
    def __init__(self, vi_game):
        """Initialize statistics."""
        self.settings = vi_game.settings
        self.reset_stats()
    def reset_stats(self):
    """Initialize statistics that can change during the game."""
    self.robots_left = self.settings.robot_limit
```

- We'll make one **GameStats** instance for the entire timeVirus *Invasion* is running. But we'll need to reset some statistics each time the player starts a new game.
- To do this, we'll initialize most of the statistics in the reset _stats() method instead of directly in __init__().
- We'll call this method from __init__() so the statistics are set properly when the GameStats instance is first created . But we'll also be able to call reset_stats() any time the player starts a new game.

SCORING

Let's implement a scoring system to track the game's score in real time and display the high score, level, and number of robots remaining. The score is a game statistic, so we'll add a score attribute to **GameStats**:

game_stats.py

```
class GameStats:
        --snip--
        def reset_stats(self):
         """Initialize statistics that can change during the game."""
         self.robots_left = self.ai_settings.robot_limit
self.score = 0
```