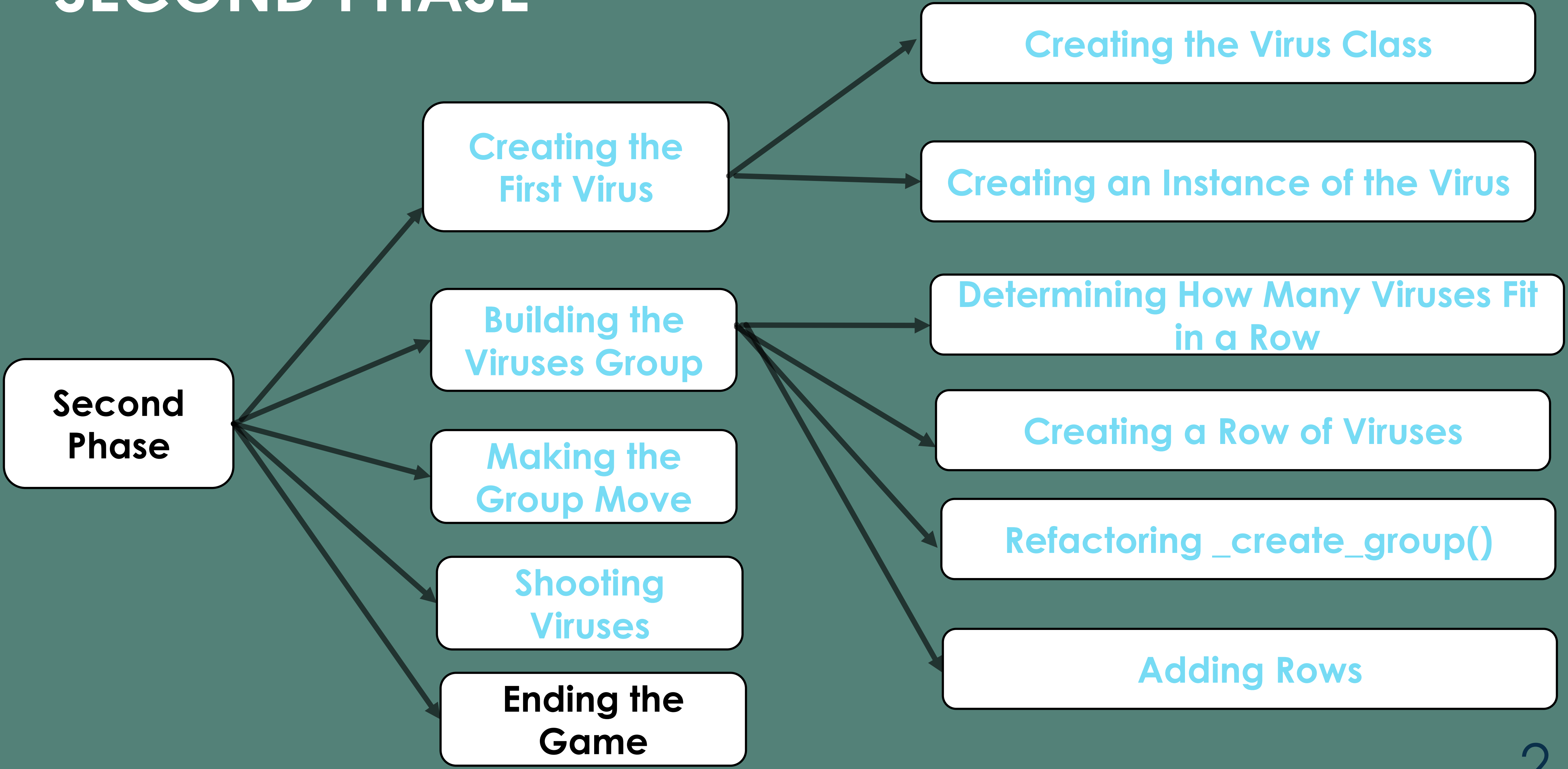
OBJECT ORIENTED PROGRAMMING WITH PYTHON

PROJECT BASED

2nd semester (Lect10)

Dr. Ann Al-Kazzaz

SECOND PHASE


Second Phase
Creating the First Virus
Building the Viruses Group
Making the Group Move
Shooting Viruses
Ending the Game
Creating the Virus Class
Creating an Instance of the Virus
Determining How Many Viruses Fit in a Row
Creating a Row of Viruses
Refactoring _create_group()
Adding Rows

RESPONDING TO VIRUS AND ROBOT COLLISIONS

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

```
class GameStats:
    """Track statistics for Virus Invasion."""

    def __init__(self, vi_game):
        """Initialize statistics."""
        self.settings = vi_game.settings
1       self.reset_stats()

    def reset_stats(self):
        """Initialize statistics that can change during the game."""
        self.robots_left = self.settings.robot_limit
```

1 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.

We'll make one **GameStats** instance for the entire time *virus invasion*is running. But we'll need to reset some statistics each time the player starts a new game.

RESPONDING TO VIRUS AND ROBOT COLLISIONS

Right now we have only one statistic, **robots_left**, the value of which will change throughout the game. The number of robots the player starts with should be stored **in *settings.py*** as **robot_limit**:

settings.py

```
# robot settings
self.robot_speed=1.5
self.robot_limit = 3
```

We also need to make a few changes in ***virus_invasion.py*** to create an instance of **GameStats**. **First**, we'll update the import statements at the top of the file:

virus_invasion.py

```
import sys
from time import sleep

import pygame
from settings import Settings
from game_stats import GameStats

from robot import Robot
--snip--
```

We import the **sleep()** function from the time module in the Python standard library so we can pause the game for a moment when the robot is hit. We also import **GameStats**.

RESPONDING TO VIRUS AND ROBOT COLLISIONS

We'll create an instance of GameStats in **__init__()**:

virus_invasion.py

```
def __init__(self):
    --snip--
    self.screen = pygame.display.set_mode((self.settings.screen_width, self.settings.screen_height))
    pygame.display.set_caption(“Virus Invasion")

    # Create an instance to store game statistics.
    self.stats = GameStats(self)
    self.robot = Robot(self)
    --snip--
```

We make the instance after creating the game window but before defining other game elements, such as the robot.

When a virus hits the robot, we'll

1. subtract one from the number of **robots left**,
2. destroy all existing viruses and Laser beams,
3. create a new group,
4. and reposition the robot in the middle of the screen.
5. We'll also pause the game for a moment so the player can notice the collision and regroup before a new group appears.

RESPONDING TO VIRUS AND ROBOT COLLISIONS

Let’s put most of this code in a new method called **_robot_hit()**. We’ll call this method from _update_viruses() when a virus hits the robot:

virus_invasion.py

```
def _robot_hit(self):
    """Respond to the robot being hit by a virus."""

    # Decrement robots_left.
1   self.stats.robots_left -= 1
    # Get rid of any remaining viruses and laser beams.
2   self.viruses.empty()
    self.laser beams.empty()
3   # Create a new group and center the robot.
    self._create_group()
    self.robot.center_robot()
    # Pause.
4   sleep(0.5)
```

RESPONDING TO VIRUS AND ROBOT COLLISIONS

1. The new method **_robot_hit()** coordinates the response when a virus hits a robot. Inside **_robot_hit()**, the number of robots left is reduced by 1.

2. After which we empty the groups viruses and laser beams.

3. Next, we create a new group and center the robot . (We'll add the method **center_robot()** to Robot in a moment.)

4. Then we add a pause after the updates have been made to all the game elements but before any changes have been drawn to the screen, so the player can see that their robot has been hit.

- The **sleep()** call pauses program execution for half a second, long enough for the player to see that the virus has hit the robot. When the **sleep()** function ends, code execution moves on to the **_update_screen()** method, which draws the new group to the screen.

RESPONDING TO VIRUS AND ROBOT COLLISIONS

In **_update_viruses()**, we replace the print() call with a call to **_robot_hit()** when a virus hits the robot:

virus_invasion.py

```
def _update_viruses(self):
    --snip--
    if pygame.sprite.spritecollideany(self.robot, self.viruses):
        self._robot_hit()
```

Here's the new method **center_robot()**; add it to the end of ***robot.py***

robot.py

```
def center_robot(self):
    """Center the robot on the screen."""
    self.rect.midbottom = self.screen_rect.midbottom
    self.x = float(self.rect.x)
```

We center the robot the same way we did in **__init__().** After centering it, we reset the **self.x** attribute, which allows us to track the robot's exact position.

VIRUSES THAT REACH THE BOTTOM OF THE SCREEN

If a virus reaches the bottom of the screen, we'll have the game respond the same way it does when a virus hits the robot. To check when this happens, add a new method in **virus_invasion.py**

virus_invasion.py

```
def _check_viruses_bottom(self):
    """Check if any viruses have reached the bottom of the screen."""
    screen_rect = self.screen.get_rect()
    for virus in self.viruses.sprites():
        if virus.rect.bottom >= screen_rect.bottom:
            # Treat this the same as if the robot got hit.
            self._robot_hit()
            break
```

- The method **_check_viruses_bottom(**) checks whether any viruses have reached the bottom of the screen. a virus reaches the bottom when its **rect.bottom** value is greater than or equal to the screen's **rect.bottom** attribute.
- If a virus reaches the bottom, we call **_robot_hit()**. If one virus hits the bottom, there's no need to check the rest, so we break out of the loop after calling **_robot_hit()**.

VIRUSES THAT REACH THE BOTTOM OF THE SCREEN

We'll call this method from **_update_viruses()**:

virus_invasion.py

```
def _update_viruses(self):
    --snip--
    # Look for virus-robot collisions.
    if pygame.sprite.spritecollideany(self.robot, self.viruses):
        self._robot_hit()

    # Look for viruses hitting the bottom of the screen.
    self._check_viruses_bottom()
```

We call **_check_viruses_bottom()** after updating the positions of all the viruses and after looking for virus and robot collisions. Now a new group will appear every time the robot is hit by a virus or a virus reaches the bottom of the screen.

GAME OVER!

virus invasion game more complete now, but the game never ends. The value of **robots_left** just grows increasingly negative. Let's add a **game_active** flag as an attribute to **GameStats** to end the game when the player runs out of robots. We'll set this flag at the end of the **__init__()** method in **GameStats.**

Game_stats.py

```
def __init__(self, vi_game):
        --snip--
        # Start virus invasionin an active state.
        self.game_active = True
```

Now we add code to **_robot_hit()** that sets **game_active** to False when the player has used up all their robots:

virus_invasion.py

```
def _robot_hit(self):
      """Respond to robot being hit by virus."""
      if self.stats.robots_left > 0:
          # Decrement robots_left.
          self.stats.robots_left -= 1
          --snip--
          # Pause.
          sleep(0.5)
      else:
            self.stats.game_active = False
```

- Most of **_robot_hit()** is unchanged. We've moved all the existing code into an if block, which tests to make sure the player has at least one robot remaining.
- If so, we create a new group, pause, and move on.
- If the player has no robots left, we set **game_active** to False.

IDENTIFYING WHEN PARTS OF THE GAME SHOULD RUN

We need to identify the parts of the game that should always run and the parts that should run only when the game is active:

virus_invasion.py

```
def run_game(self):
    """Start the main loop for the game."""
    while True:
        self._check_events()
        if self.stats.game_active:
            self.robot.update()
            self._update_laser beams()
            self._update_viruses()
        self._update_screen()
```

- In the main loop, we always need to call **_check_events()**, even if the game is inactive. For example, we still need to know if the user presses **Q** to quit the game or clicks the button to close the window.
- We also continue updating the screen so we can make changes to the screen while waiting to see whether the player chooses to start a new game. The rest of the function calls only need to happen when the game is active, because when the game is inactive, we don't need to update the positions of game elements.
- Now when you play *VirusInvasion*, the game should freeze when you've used up all your robots.