Object-Oriented Programming

Object-Oriented Programming

Dr. Seán Russell
sean.russell@ucd.ie

School of Computer Science,
University College Dublin

September 12, 2018

Table of Contents

Learning outcomes

After this lecture and the related practical students should...

- understand why we use OOP
- be able create multiple objects from the same class
- understand the relationship between classes and objects

Object-Oriented-Programming

Why do we do it?

Code Reuse

- The idea is that we can write our code in a way that we can use it many times without changing it
- This can save us a large amount of time when completing projects

Easier to Understand

- Another reason for using object oriented programming is that it makes code easier to understand, maintain or change

Example Arcade Game

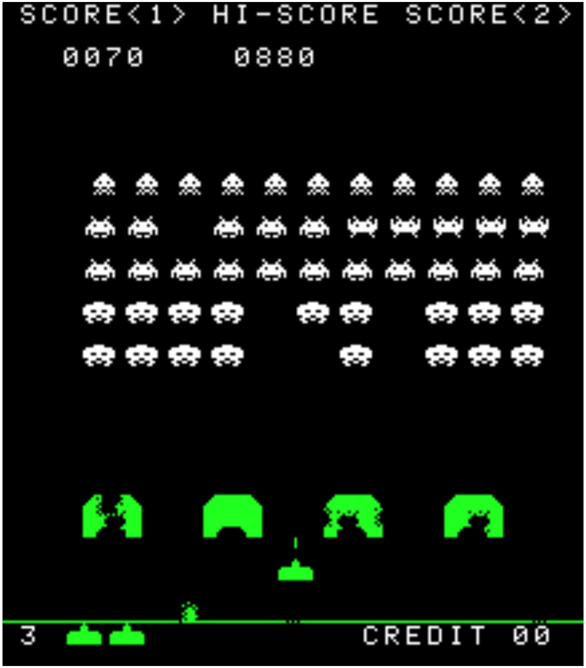
- The game will have a very basic functionality
- The game should allow the player to
  - Move our ship
  - Fire bullets
  - Destroy enemy ship

Just the Beginning

- As we complete more of the course, there will be more examples
- At the end of some chapters, there may be tasks to complete
- Some additional information is provided in the book that is not discussed in the course

Space Invaders

- Space Invaders was the first really successful arcade game
- Before we can complete a full game, there are a lot of topics we must learn


SCORE<1> HI-SCORE SCORE<2>
0070 0880
3
CREDIT 00

Game Design Problem

The initial parts of the Game are designed in Java, but do not use any object-oriented techniques

- We will start by representing the players location and health
- First we create a class named `Game`, to hold our game

Variables for name and health

```java
class Game {
  int playerX = 50;
  int playerY = 400;
  int playerHealth = 20;
}
```

Storing enemy ship information

- We to store the info about the alien ships, but how do we do it?
- We could create variables for each ship, with names like enemy1X,....

```
int enemy1X = 70;
int enemy1Y = 50;
int enemy2X = 90;
int enemy2Y = 50;
```

Storing enemy character information

- We could use arrays
- We must remember which index is for which ship

```
class Game {
  int playerX = 50;
  int playerY = 400;
  int playerHealth = 20;

  int numberOfEnemies = 3;
  int[] enemyX = { 70, 90, 110};
  int[] enemyY = { 50, 50, 50 };
}
```

- Adding more ships does not require new variables

Adding Functionality

- The code represents the player and the alien ships
- Enemy ships must be able to be destroyed
- When destroyed, we need to remove the information about that ship
- This functionality is defined as a method that can remove ship `i` from the arrays
- This method can be used every time a ship must be removed

Removing a Character

```
void removeShip(int m){
  // code to remove the location of enemy m from the arrays
  if(m < numberOfEnemies && m >= 0){
    for(int i = m; i < numberOfEnemies -1; i++){
      enemyX[i] = enemyX[i+1];
      enemyY[i] = enemyY[i+1];
    }
    enemyX[numberOfEnemies-1] = 0;
    enemyY[numberOfEnemies-1] = 0;
    numberOfEnemies--;
  }
}
```

Testing Remove Functionality

- Once written, we should always test code
- Not testing $\rightarrow$ bad grades
- To test we will do the following steps:
  - Print info in the arrays
  - Call the `removeShip` method
  - Print info in the arrays
- If the process is completed correctly, there should be only two characters printed after the remove method is called

Testing Remove Character method

```
void test() {
  System.out.println("Player (" + playerX +
   ", " + playerY + ") has health " +
   playerHealth);
  for (int i = 0; i < numberOfEnemies; i++) {
    System.out.println("Enemy "+i+" located
   at (" + enemyX[i] + ", " + enemyY[i] ")");
  }
  removeShip(1);
  for (int i = 0; i < numberOfEnemies; i++) {
    System.out.println("Enemy "+i+" located
   at (" + enemyX[i] + ", " + enemyY[i] ")");
  }
}
```

Changing Requirements

- Changing requirements is a very common problem
- After some time working on the game, it is decided that ships should have a cooldown for their weapons
- We must change our code
- We add new instance variables, then update our test code

Game Problem

variables representing strength

```java
int playerCooldown = 10;
int[] enemyCooldown = {15, 20, 0};
```

- After we have made this change, we need to test our code again and there is a problem

Comparing the result before and after

Before	After
enemyX: {70, 90, 110}	enemyX: {90, 110}
enemyY: {50, 50, 50}	enemyY: {50, 50}
enemyCD: {15, 20, 0}	enemyCD: {15, 20, 0}

Small Mistake

- When we added the new variable, we forgot to change the remove character method to also remove the cooldown
- We might have done the same in all of the other methods that deal with enemy ships
- This makes changing the program very difficult
- We decide that instead that we are going to use OOP to solve our problem

Using OOP

- Instead of having a different array for each piece of information we are going to write a single class for ships
    - Every ship has the same info (location, health, cooldown)
    - We can write a class that contains an instance variable for each of these
- This means that every character can be represented by a single array

Ship Class

```
class Ship {
  int x;
  int y;
  int cooldown;
  int health;
  Ship(int x1, int y1, int c, int h){
    x = x1;
    y = y1;
    cooldown = c;
    health = h;
  }
}
```

Ship Class

- The Ship class groups together values for x, y, health and cooldown
- It also provides a constructor
- To represent the player we only need a single variable
- To represent the other ships, we only need one array

Character Ships

Representing Ships

```java
Ship player = new Ship(50, 400, 20, 20);

int numberOfEnemies;
Ship[] enemyShips;
```

Constructing Ships

- We need a constructor for the Game class that creates our enemy ships for us

Constructing Ships

```
Game() {
  enemyShips = new Ship[numberOfEnemies];
  enemyShips[0] = new Ship(70, 50, 15, 5);
  enemyShips[1] = new Ship(90, 50, 20, 5);
  enemyShips[2] = new Ship(110, 50, 0, 5);
}
```

Removing a Ship Updated

- Instead of changing multiple arrays, only a single array needs to be changed

```java
void removeShip(int m) {
  if (m < numberOfEnemies && m >= 0) {
    for (int i = m; i < numberOfEnemies - 1; i++) {
      enemyShips[i] = enemyShips[i + 1];
    }
    enemyShips[numberOfEnemies - 1] = null;
    numberOfEnemies--;
  }
}
```

Updating Tests

- Previously we just printed the values of each variable, now the variables are contained inside classes
- That means that instead of printing the variable `playerHealth`, we print `player.health`
- Instead of printing out the array element `enemyX[i]`, we print `enemyShips[i].x`.

Updated Tests

```java
class Game {
  // variables declared and given values
  void test() {
    System.out.println("Player (" + player.x + ", "
   + player.y + ") has health " + player.health);

    for (int i = 0; i < numberOfEnemies; i++) {
      System.out.println("Enemy "+i+" located at ("
   + enemyShips[i].x + ", " + enemyShips[i].y ") has
   health "+ enemyShips[i].health );
    }
  }
...
}
```

Bullets

- Players and enemy ships are able to shoot bullets at each other
- We must represent
  - the location of the bullet
  - the direction it is moving in
  - how much damage it can cause

Designing the Bullet class

```
class Bullet{
  int x;
  int y;
  boolean up; // true for up, false for down
  int damage;
  Bullet(int x1, int y1, boolean u, int d){
    x = x1;
    y = y1;
    up = u;
    damage = d;
  }
}
```

Creating bullets in the Ship

```
class Ship {
   int x, y, cooldown, health;
   boolean player;
   Ship(int x1, int y1, int c, int h, boolean p){
      x = x1;
      y = y1;
      cooldown = c;
   health = h;
      player = p;
   }
   Bullet fire() {
      Bullet b = null;
      if(cooldown == 0) {
         b = new Bullet(x,y, player, this);
      }
      return b;
   }
}
```

Did it hit?

```java
boolean isHit(Bullet b) {
  if(b.x >= x && b.x <= x + 10 &&
      b.y >= y && b.y <= y + 10) {
    return true;
  }
  return false;
}
```

- Makes assumptions about size of ship
- Checks if the location of the bullet is inside a square of size 10 at the coordinates of the ship

Cohesion

- Cohesion in programming is about how much the elements of a component belong together
- Usually cohesion is in terms of our classes
- Cohesion measures the strength of relationship between methods in a class
- Classes are usually described as “high cohesion” or “low cohesion”
- In a class with high cohesion, all of the functionality is closely related

High Cohesion is Better

- Classes with high cohesion tend to be better
- This is because high cohesion is associated with desirable traits robustness, reliability, reusability, and understandability
- In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand

Are our classes Cohesive?

- The Ship class has functionality related to ships
- The bullet class has functionality related to bullets
- These are related, making them highly cohesive
- The only exception to this is the Game class
  - This class is intended to manage the overall game
  - However, it also contains methods for testing the the functionality is working correctly
  - This gives the class functionalities that are quite different

Is it easier?

- Cohesion makes it easier to understand what each class does and how they relate to each other
- The ship class is responsible for maintaining information about ships, if we are modifying code related to the levels in the game, we know the ship class will probably not be broken by the changes that we have made
- A class that is too large is a sign that we are being too broad in our definition of related functionality