**Урок №15. Использование ООП при генерации игры**

**Вводный блок (10 минут)**

**Цели и задачи урока:**

1. Рассмотрим принципы построения объектов с использованием ООП.
2. Создадим основной класс и наследуем от него необходимые параметры.
3. Разработаем несколько дочерних классов.

**Вспомним прошлые занятия**

1. Что такое ООП?
2. Как создавать классы и зачем они нужны?
3. Как располагаются данные внутри класса? Что такие инкапсуляция?

**Познание нового (10 минут)**

Для реализации игры нам потребуется такое понятие как наследование.

Наследование - это механизм, который позволяет создавать новые классы на основе существующих. Дочерние классы (подклассы) наследуют свойства и методы из родительских классов (суперклассов). Наследование позволяет повторно использовать код и создавать иерархии классов.

Вот основные аспекты наследования в ООП:

1. Базовый класс и производный класс:

- Базовый класс (родительский, суперкласс) - это исходный класс, который определяет общие свойства и поведение.

- Производный класс (дочерний, подкласс) - это новый класс, который наследует от базового класса.

2. Иерархия классов:

- Наследование позволяет создавать иерархии классов, где производные классы могут наследовать от других производных классов.

- Это создает древовидную структуру, в которой каждый подкласс наследует от своего родительского класса.

3. Повторное использование кода:

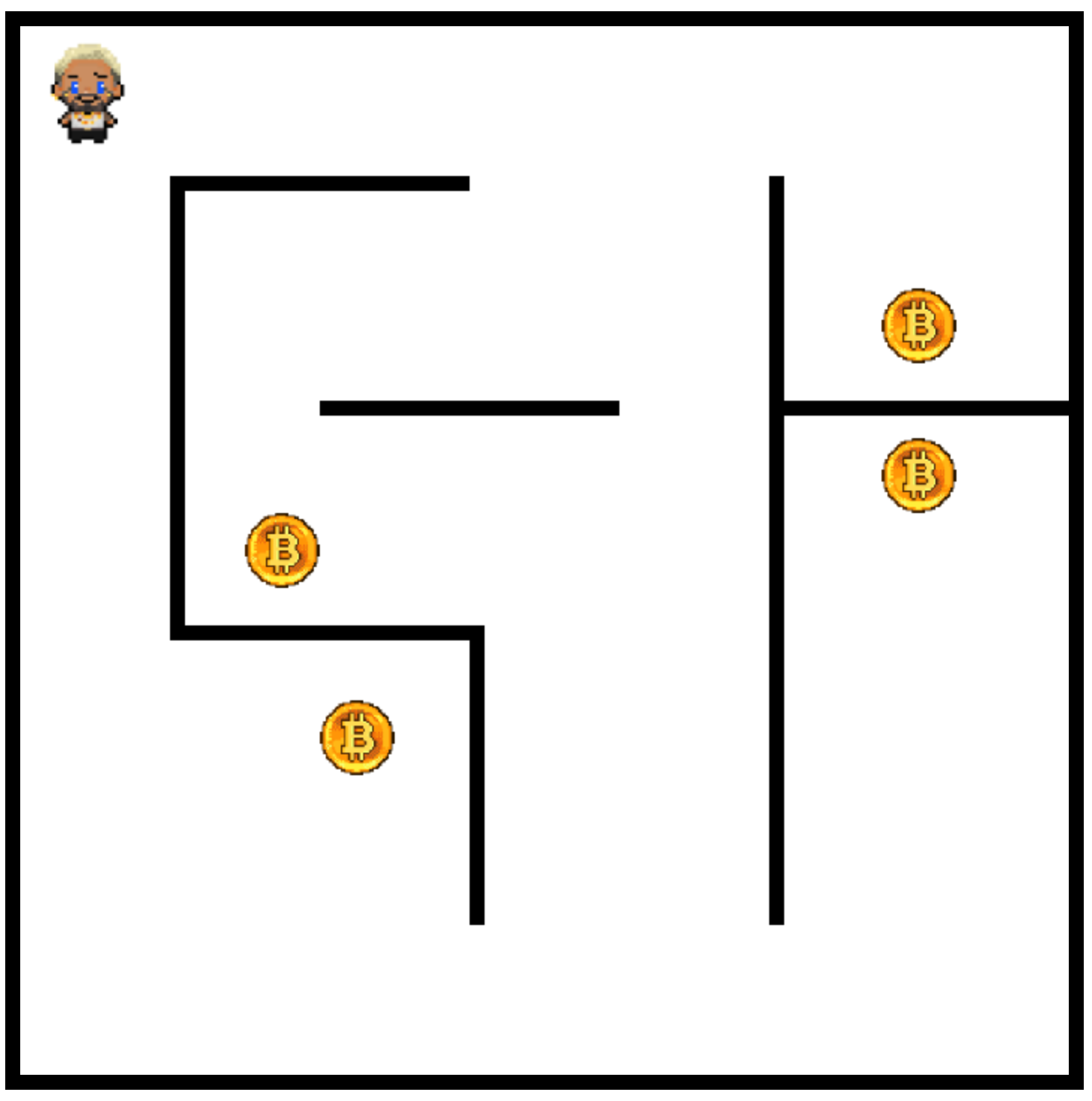
- Наследование позволяет повторно использовать код, избегая дублирования.

- Вместо того, чтобы писать одни и те же свойства и методы в каждом классе, можно определить их в базовом классе и наследовать в производных классах.

**Работа над проектом (10 минут)**

Тема: Создание нового проекта и написание программы, которая создает игровой уровень с использованием парадигмы ООП.

Игровой уровень, который будет создан на занятии



На данном занятии мы создадим уровень с использованием ООП.

**Вопросы к ученикам:**

1. Из чего состоит уровень?
2. Сколько здесь классов (различных объектов)?

В больших проектах, где много объектов, всегда принято сначала находить общее. Если присмотреться, то у всех объектов есть 4 общих элемента: координаты и размеры.

Их можно вынести в отдельный класс, а дальше все свойства, записанные в этом классе, будут наследоваться в других классах.

**Некомпьютерная активность (15 минут)**

Наследование: Семейное древо

1. Попросите участников создать семейное древо, используя карточки или листы бумаги.
2. Пусть они изобразят отношения между родителями, детьми, дедушками/бабушками и другими родственниками.
3. Объясните, что это представляет собой иерархию классов, где родители - это "суперклассы", а дети - "подклассы", наследующие свойства от родителей.
4. Обсудите, как принцип наследования позволяет повторно использовать код и создавать иерархии.

**Работа над проектом (40 минут)**

На первом этапе создадим класс gameObject, который будет объединять все объекты на уровне и в классе пропишем ранее перечисленные 4 параметра:

class gameObject {

constructor(x, y, w, h) {

this.x = x;

this.y = y;

this.w = w;

this.h = h;

}

}

Если посмотреть далее, то можно увидеть, что стены на уровне не обладают какими-либо спрайтами а отображены в виде прямоугольников.

Таким образом, мы можем в этом классе написать метод draw, который будет отображать прямоугольные фигуры:

class gameObject {

constructor(x, y, w, h) {

this.x = x;

this.y = y;

this.w = w;

this.h = h;

}

draw(context) {

context.fillRect(this.x, this.y, this.w, this.h)

}

}

И это будет самым базовым классом, который используется во всем проекте. На основе этого класса мы можем создать все стены, которые есть на уровне.

Для этого будем использовать массив и выводить все стены через цикл for:

class gameObject {

constructor(x, y, w, h) {

this.x = x;

this.y = y;

this.w = w;

this.h = h;

}

draw(context) {

context.fillRect(this.x, this.y, this.w, this.h)

}

}

const walls = [

new gameObject(100, 100, 10, 300),

new gameObject(100, 100, 200, 10),

new gameObject(100, 400, 200, 10),

new gameObject(300, 400, 10, 200),

new gameObject(500, 100, 10, 500),

new gameObject(200, 250, 200, 10),

new gameObject(500, 250, 200, 10),

]

function move() {

context.clearRect(0, 0, canvas.width, canvas.height);

for (wall of walls) {

wall.draw(context);

}

requestAnimationFrame(move);

}

У следующих же объектов будут изображения, поэтому можно создать другой класс, к которому можно наследовать все написанные методы и конструктор от gameObject:

class imageObject extends gameObject {

constructor(x, y, w, h, image) {

super(x, y, w, h);

this.image = new Image();

this.image.src = image;

}

draw(context) {

context.drawImage(this.image, this.x, this.y, this.w, this.h)

}

}

Для наследования класса используется ключевое слово extends, после которого записывается наименование наследуемого класса.

Как можно заметить, в этом классе так же есть метод draw и он перезаписывает тот метод draw, который был в классе gameObject.

Помимо этого, используется функция super, которая вызывает конструктор базового класса (gameObject).

Теперь можно создать монетки, которые располагаются на уровне:

**const coins = [**

**new imageObject(575, 175, 50, 50, 'coin.png'),**

**new imageObject(575, 275, 50, 50, 'coin.png'),**

**new imageObject(150, 325, 50, 50, 'coin.png'),**

**new imageObject(200, 450, 50, 50, 'coin.png'),**

**]**

function move() {

context.clearRect(0, 0, canvas.width, canvas.height);

for (wall of walls) {

wall.draw(context);

}

**for (coin of coins) {**

**coin.draw(context);**

**}**

requestAnimationFrame(move);

}

На последнем этапе создадим игрового персонажа, Что мы о нем знаем? Он должен передвигаться и отображаться на уровне. Отображение есть в классе imageObject, а вот перемещения не реализовано. Поэтому, мы можем наследоваться от класса imageObject и добавить метод move:

class heroObject extends imageObject {

constructor(x, y, w, h, image) {

super(x, y, w, h, image);

}

move(dx, dy) {

this.x += dx;

this.y += dy;

}

}

В главной функции move добавим отображение и перемещение для созданного персонажа и не забудем добавить события нажатия на клавиши для всего окна:

**const hero = new heroObject(10, 10, 70, 70, "player.png");**

**let move\_x = 0;**

**let move\_y = 0;**

function move() {

context.clearRect(0, 0, canvas.width, canvas.height);

for (wall of walls) {

wall.draw(context);

}

for (coin of coins) {

coin.draw(context);

}

**hero.draw(context);**

**hero.move(move\_x, move\_y);**

requestAnimationFrame(move);

}

move();

**window.addEventListener("keydown", (event) => {**

**if (event.key.toLowerCase() == "w") { move\_y = -2; }**

**if (event.key.toLowerCase() == "s") { move\_y = 2; }**

**if (event.key.toLowerCase() == "a") { move\_x = -2; }**

**if (event.key.toLowerCase() == "d") { move\_x = 2; }**

**})**

**window.addEventListener("keyup", (event) => {**

**if (event.key.toLowerCase() == "w") { move\_y = 0; }**

**if (event.key.toLowerCase() == "s") { move\_y = 0; }**

**if (event.key.toLowerCase() == "a") { move\_x = 0; }**

**if (event.key.toLowerCase() == "d") { move\_x = 0; }**

**})**

Помимо этого, можно добавить коллизию для объектов. На примере непроходимости сквозь стены рассмотрим коллизию.

Создадим новый метод collide в классе gameObject, поскольку коллизия важна для всех объектов на уровне:

class gameObject {

constructor(x, y, w, h) {

this.x = x;

this.y = y;

this.w = w;

this.h = h;

}

draw(context) {

context.fillRect(this.x, this.y, this.w, this.h)

}

**collide(obj) {**

**if (this.x > obj.x**

**&& this.x < obj.x + obj.w**

**&& this.y > obj.y**

**&& this.y < obj.y + obj.h**

**) {**

**return true;**

**}**

**if (this.x + this.w > obj.x**

**&& this.x + this.w < obj.x + obj.w**

**&& this.y > obj.y**

**&& this.y < obj.y + obj.h**

**) {**

**return true;**

**}**

**if (this.x + this.w > obj.x**

**&& this.x + this.w < obj.x + obj.w**

**&& this.y + this.h > obj.y**

**&& this.y + this.h < obj.y + obj.h**

**) {**

**return true;**

**}**

**if (this.x > obj.x**

**&& this.x < obj.x + obj.w**

**&& this.y + this.h > obj.y**

**&& this.y + this.h < obj.y + obj.h**

**) {**

**return true;**

**}**

**}**

}

В функции move добавим проверку на коллизию и смещение назад, если коллизия сработает:

function move() {

context.clearRect(0, 0, canvas.width, canvas.height);

for (wall of walls) {

wall.draw(context);

}

for (coin of coins) {

coin.draw(context);

}

hero.draw(context);

hero.move(move\_x, move\_y);

**for (wall of walls) {**

**if (hero.collide(wall)) {**

**hero.move(-move\_x, -move\_y);**

**}**

**}**

requestAnimationFrame(move);

}

**Рефлексия (5 минут)**

"Что нового вы узнали сегодня?"

"Что такое наследование? Как правильно выстраивать иерархию классов?"