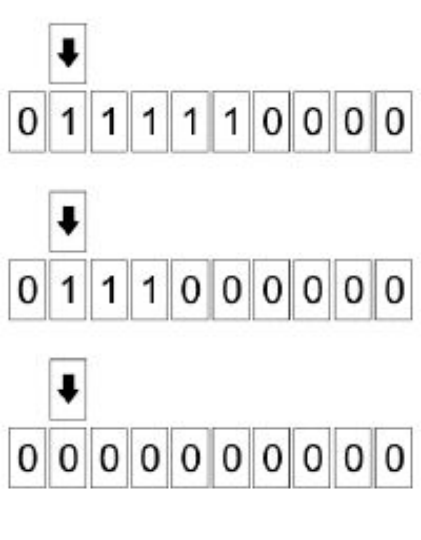
Кирилл Волков @ MERA
github.com/vulko/Cpp_Basics_Lectures

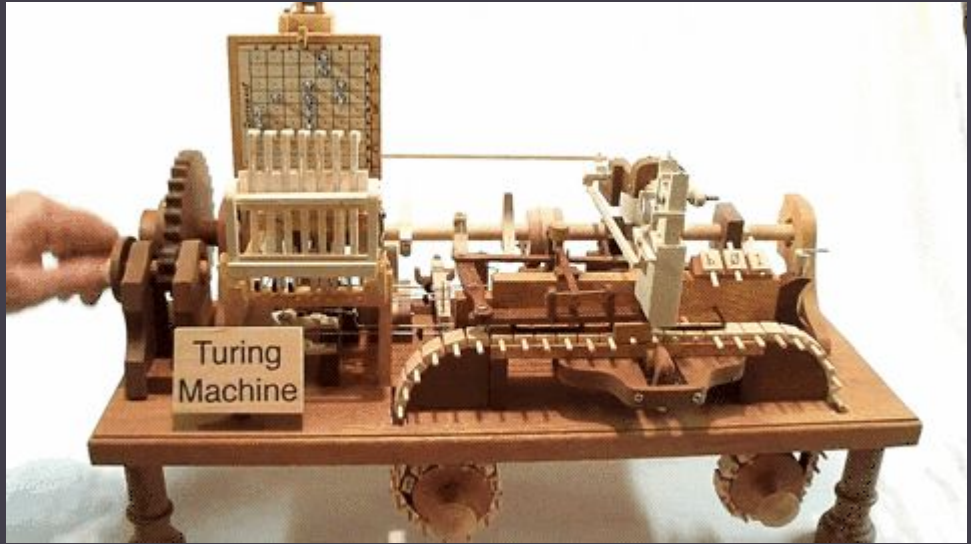
C++

Основы линейного программирования: функции, перегрузка функций, аргументы по умолчанию.

Основы объектно-ориентированного программирования: структуры и классы.
Конструкторы, деструкторы, операторы и операнды.

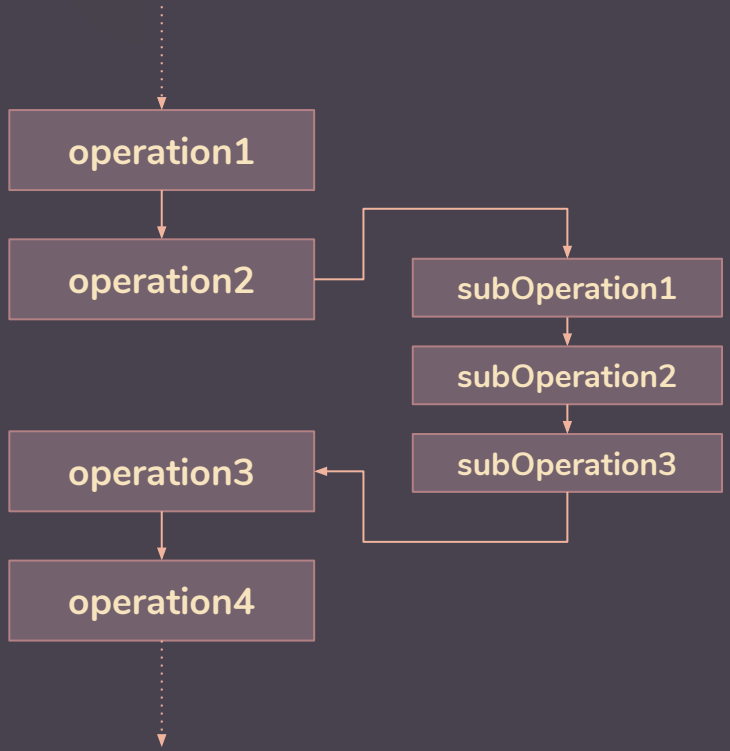
Машина Тьюринга





Линейное программирование

Main operations sequence

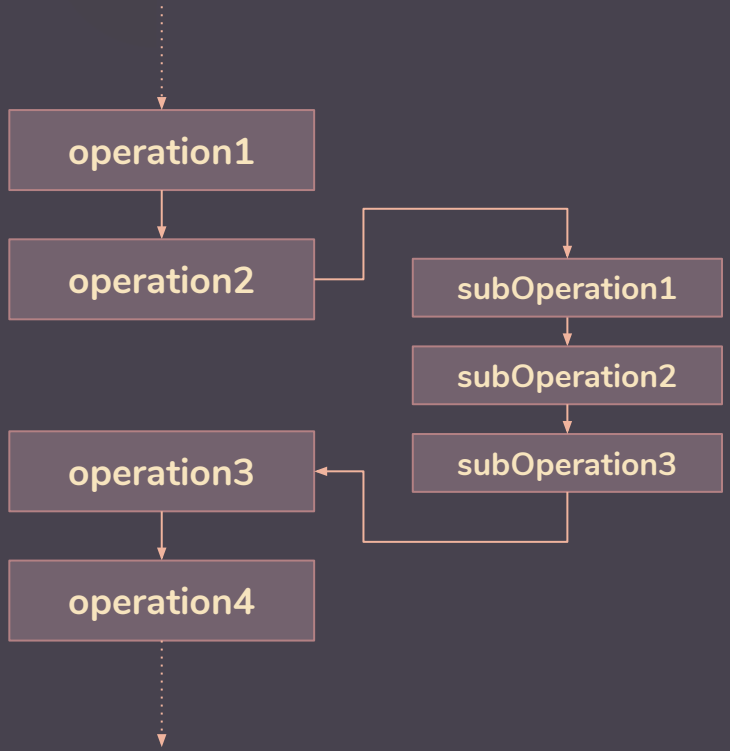


Code

```
int main() {
    ...
    operation1();

    subOperation1();
    subOperation2();
    subOperation3();
    subOperation4();

    operation3();
    operation4();
    ...
}
```

Линейное программирование: эволюция

Main operations sequence



Code

```
int main() {
    ...
    operation1();
    operation2();
    operation3();
    operation4();
    ...
}

void operation2() {
    subOperation1();
    subOperation2();
    subOperation3();
    subOperation4();
}
```

Линейное программирование: эволюция

```
#include “keyboard.h”
#include “display.h”

int main() {
    initKeyboard();
    initDisplay();

    while (userInput != ‘x’) {
        if (userInput == ‘w’) {
            moveForward();
        } else if (userInput == ‘s’) {
            moveLeft();
        } else if (userInput == ‘a’) {
            moveRight();
        } else if (userInput == ‘d’) {
            moveBackward();
        }
    }

    return 0;
}
```

main.cpp

```
void moveForward() {
    // move forward
}

void moveBackward() {
    // move backward
}

void moveLeft() {
    // move left
}

void moveRight() {
    // move right
}
```

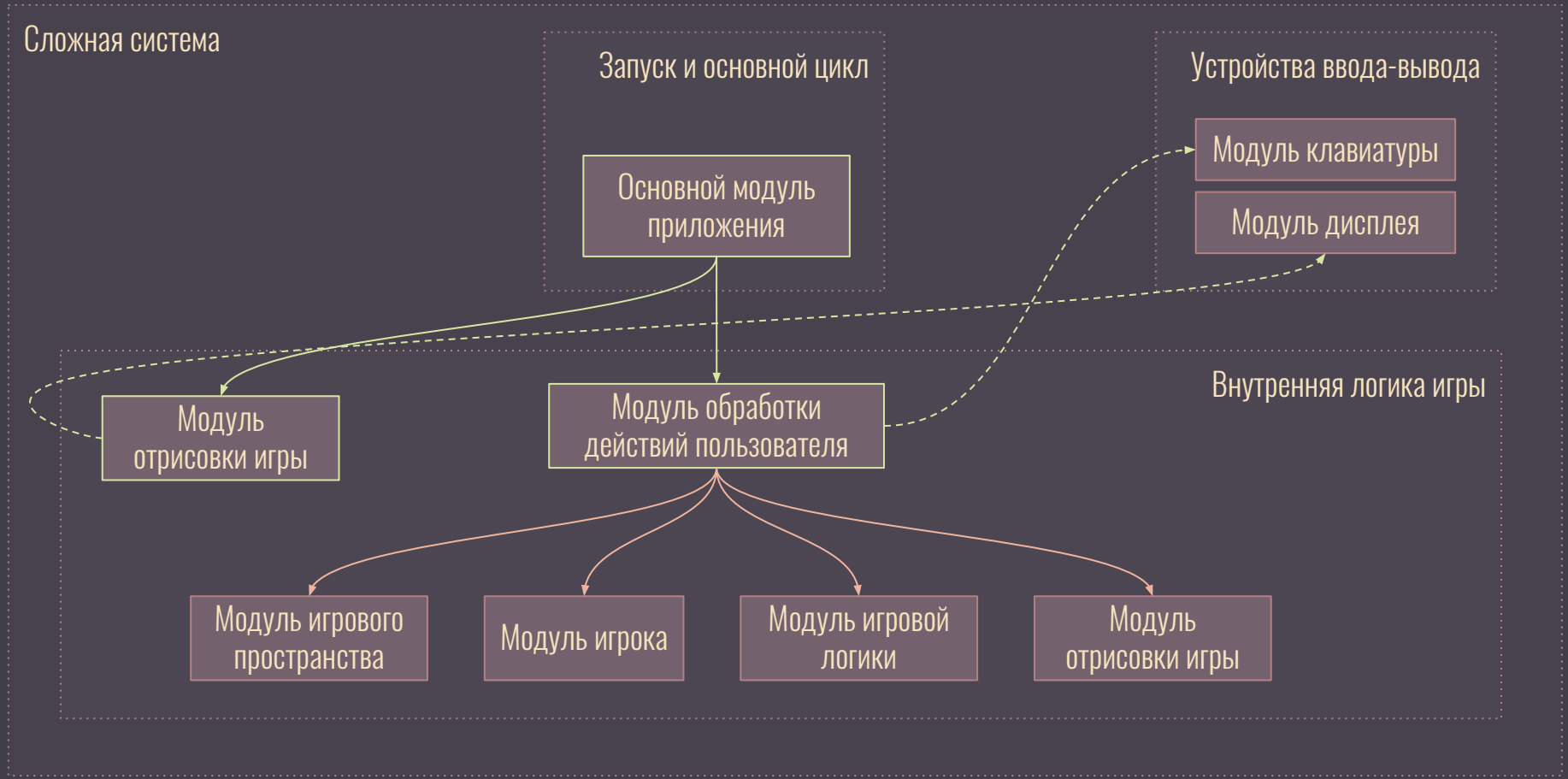
keyboard.h

```
#include “sys_keyboard.h”

void initKeyboard() {
    sys_keyboard keyboard;
    keyboard.start(MODE_READ);
}
```

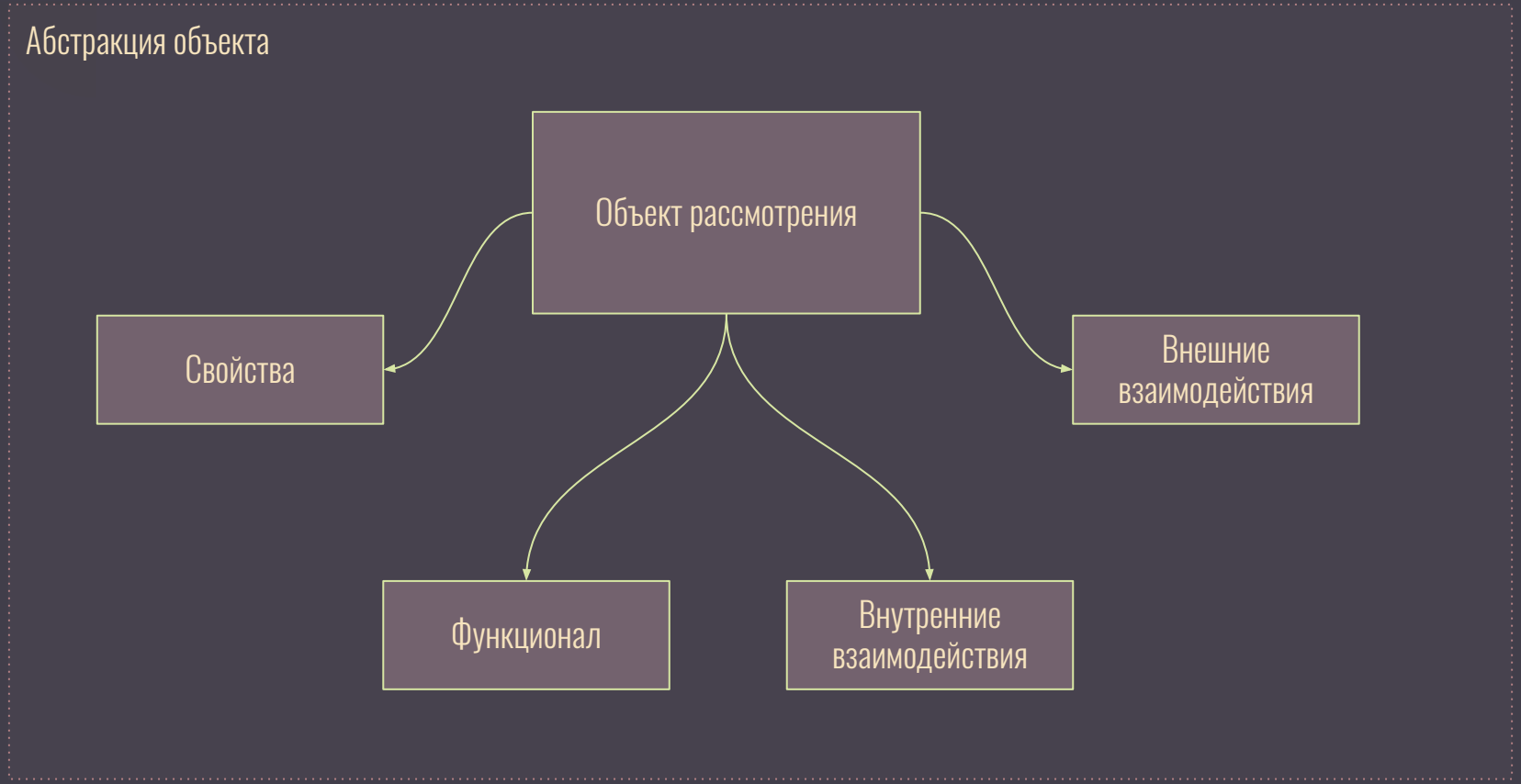
display.h

```
#include “sys_display.h”

void initKeyboard() {
    sys_display display;
    display.start(MODE_DRAW);
}
```

Линейное программирование: эволюция



Линейное программирование: революция


Абстракция объекта
Объект рассмотрения
Свойства
Внешние взаимодействия
Функционал
Внутренние взаимодействия


Абстракция объекта Игра
ИГРА
Свойства
Сведения об игроке
Сведения об игровом поле
Внешние взаимодействия
Клавиатура (действия пользователя)
HDD (сохранение и загрузка состояния игры)
Внутренние взаимодействия
Функционал
Обработка действий пользователя
“Игровая логика”
Взаимодействия с объектами представления составных частей игры
Изменение состояния игры
Отрисовка игрового поля в соответствии с состоянием игры
От действий юзера
От игровой логики

На пути к ООП: объект как абстракция

Идея

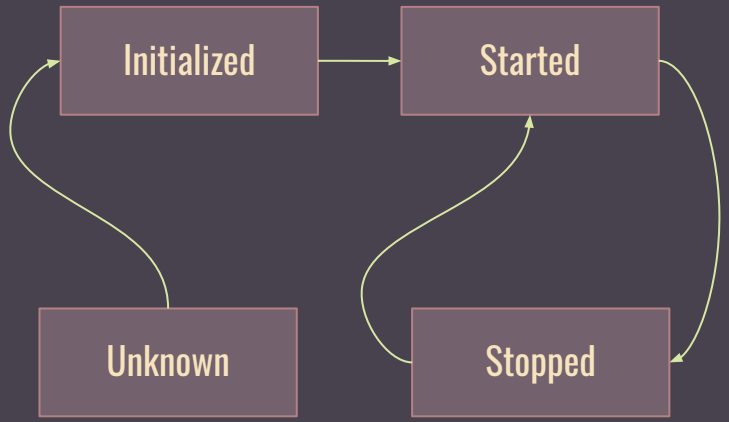
```
class Game {

    // свойства

    // функционал

    // внешние зависимости

}
```

Реализация

```
class Game {

    // свойства
    GameState mGameState;

    // функционал
    void startGame();
    void stopGame();
    void initialize();

    // внешние зависимости
    Player mPlayer;
    GameBoard mGameBoard;

}
```

На пути к ООП: реализация

`GameState`




```
enum GameState
{

   // объект Game не инициализирован и не готов к использованию
   Unknown,

   // объект Game инициализирован и готов к использованию
   Initialized,

   // игра начата и в процессе
   Started,

   // игра остановлена... например закончилась или пользователь
   // завершил игровую сессию
   Stopped

};
```

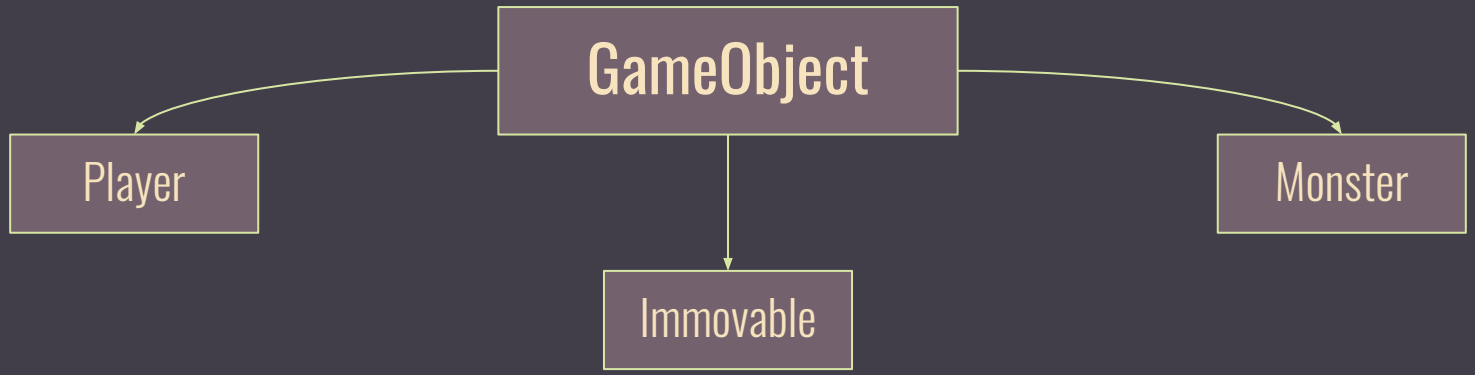
На пути к ООП: реализация

```
void Game::initialize() {
    if (mGameState != GameState.UNKNOWN) {
        // wrong state
        return;
    }

    // initialize
    mGameState = GameState.INITIALIZED;
}

void Game::startGame() {
    if (mGameState == GameState.STARTED) {
        // already started
        return;
    }

    if (mGameState == GameState.UNKNOWN) {
        // wrong state
        return;
    }

    if (mGameState == GameState.INITIALIZED) {
        // FIRST START
    }

    if (mGameState == GameState.STOPPED) {
        // RESTART
    }
}

void Game::stopGame() {
    if (mGameState == GameState.UNKNOWN ||
        mGameState == GameState.INITIALIZED) {
        // wrong state
        return;
    }

    if (mGameState == GameState.STOPPED) {
        // RESTART
    }
}
```

На пути к ООП: итерации

На пути к ООП: формализация

Наследование



Все эти объекты объединяют некоторые свойства... например:

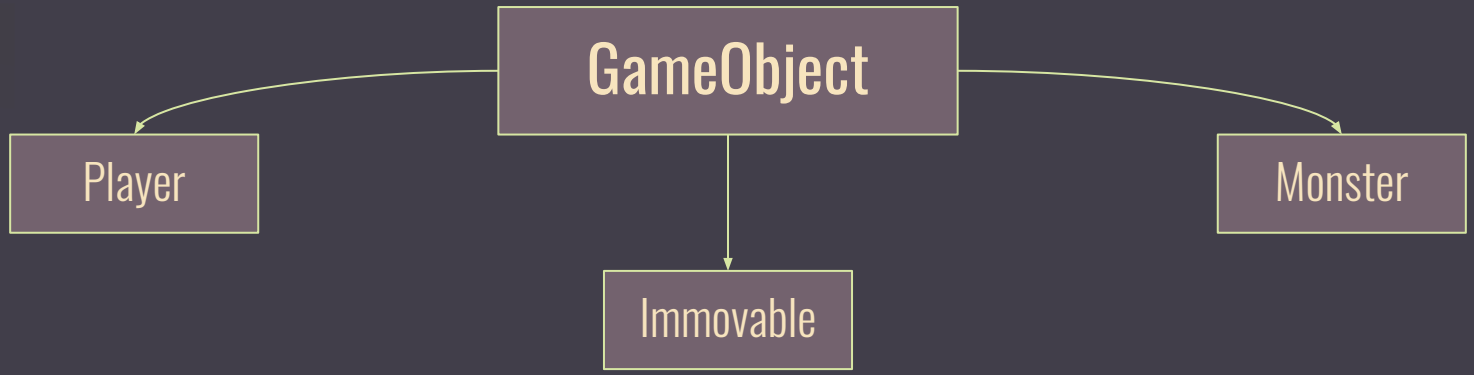
- имя
- позиция в игровом мире
- абстракция отрисовки

Иерархию можно продолжить, если требуется менять внутренние свойства, поведение и т.п...

Например наследовать от Immovable:

- Ground
- Water
- Tree
- Stone

Наследование

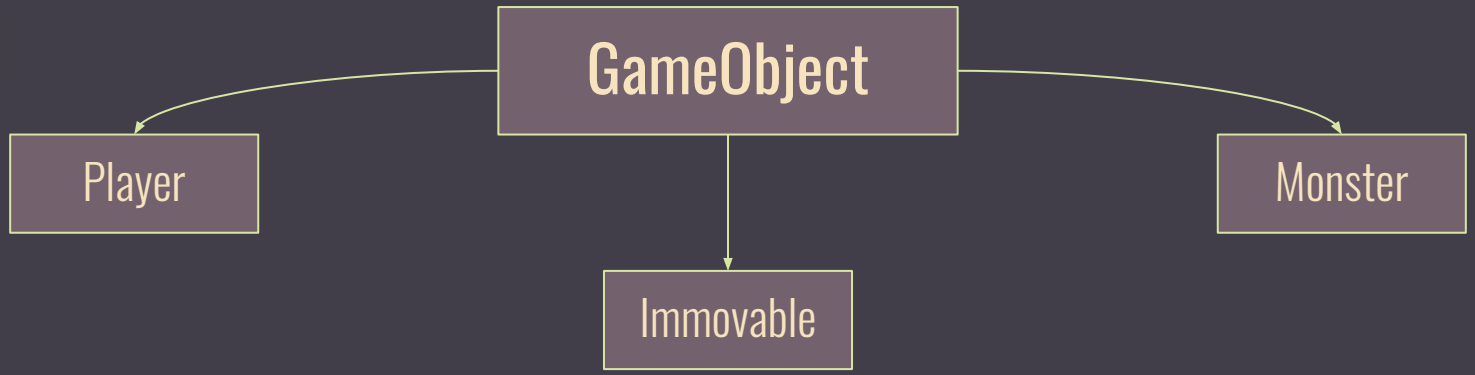



```
class Player : GameObject  {

    void draw();
    void draw(Rect rect);

    void move(int x, int y);
    void move(float x, float y);

}
```

overloading

overloading

Полиморфизм




```
class GameObject {

    int posX;
    int posY;

    virtual void draw() = 0;

}
```

pure virtual

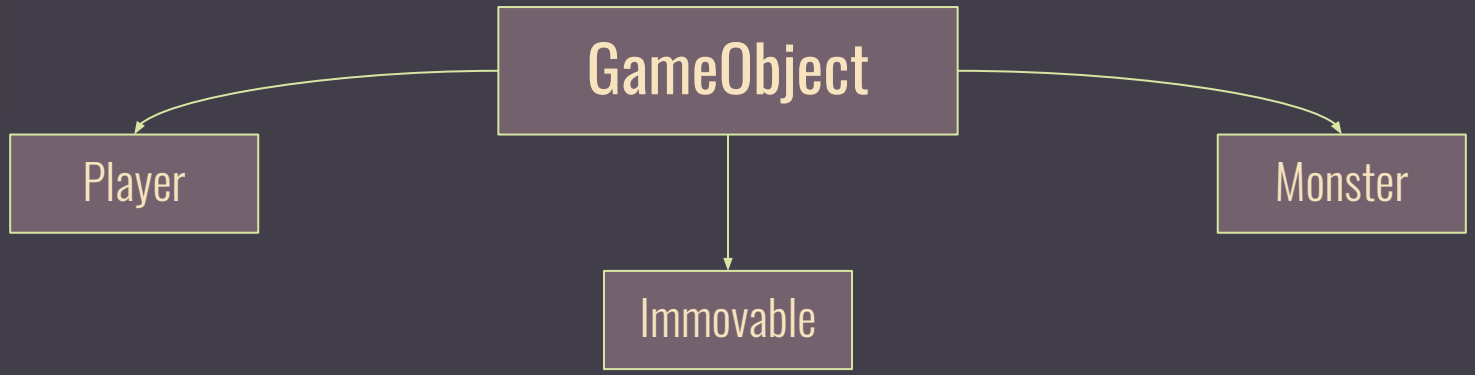
virtual

```
class Player : GameObject {

    int posX;
    int posY;

    virtual void draw() {
        println("I'm a player standing at %d:%d", posX, posY);
    }

}
```

virtual

implementation

Полиморфизм




```
class GameObject {

    int posX;
    int posY;

    virtual void draw() = 0;

}
```

pure virtual

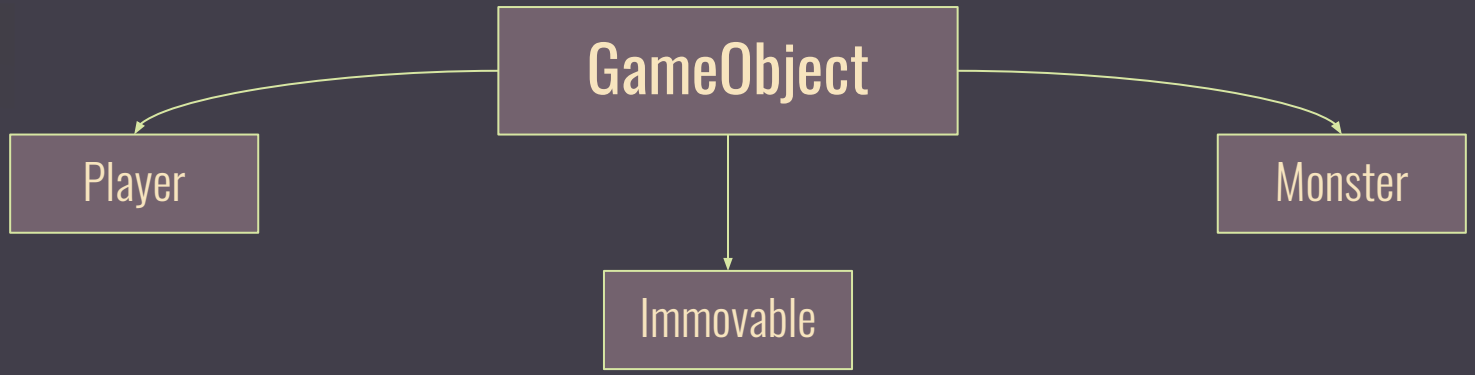
virtual

```
class Player : GameObject {

    int posX;
    int posY;

    virtual void draw() {
        println("I'm a player standing at %d:%d", posX, posY);
    }

}
```

virtual

implementation

Полиморфизм

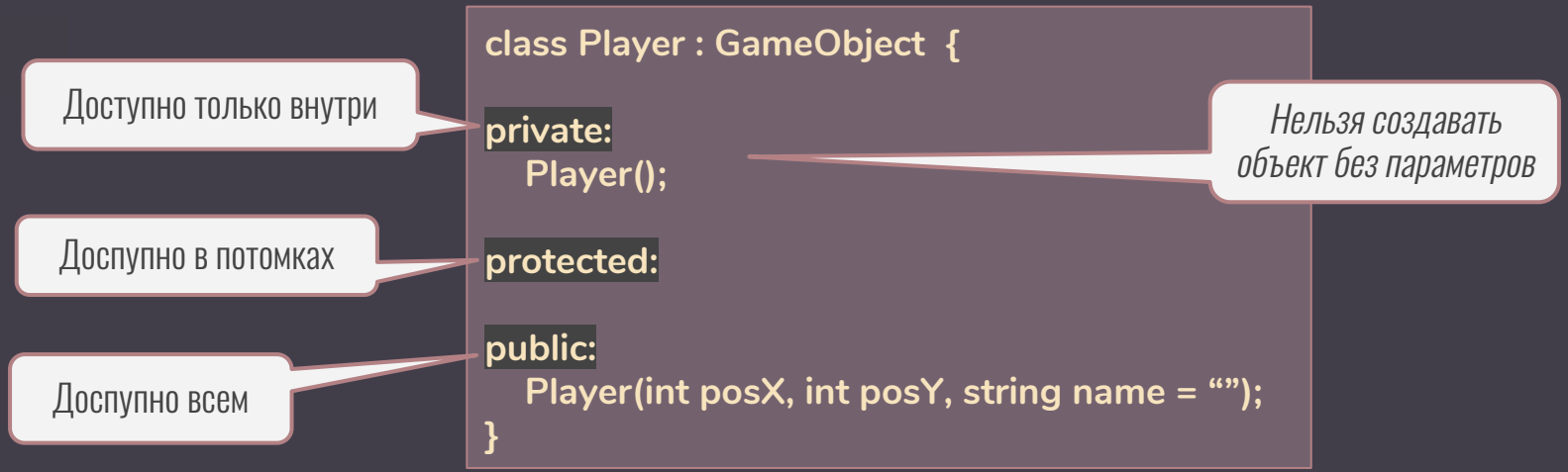



```
class Player : GameObject  {

    Player();
    Player(int posX, int posY, string name = “”);

}
```

Конструктор по умолчанию

Параметрический полиморфизм

Инкапсуляция



Конструктор класса

```
class Player {
public:
    Player() = default;

    Player(int posX, int posY, string name = “Player”)
        : mPosX(0)
        : mPosY(0)
        : mName(name) {
    }

    Player(const Player& a) = delete;

    Player(Array&& a);

    virtual ~Player() { // release memory }

private:
    int mPosX;
    int mPosY;
    string mName;
};
```

Конструктор по-умолчанию

компилятор создаст сам

Инициализация полей

Конструктор копирования

явно удален

Конструктор перемещения

Деструктор

Если есть виртуальные методы, деструктор тоже должен быть виртуальным