

## Лабораторная работа №3: Инкапсуляция. Повторное использование классов.

### Цель работы:

- Знакомство с принципом инкапсуляции.
- Применение ранее разработанных классов.

### Задачи:

- Реализовать программу согласно описанному функционалу.
- Создать класс для хранения числовых данных в формате массива и использовать класс для хранения больших чисел.
- Разработать систему классов для реализации логики программы. Классы должны удовлетворять принципу инкапсуляции.

### Функционал программы:

- Загрузка сохраненных шаблонов из редактора противников реализованного в лабораторной работе №2 в качестве противников в игре при запуске программы.
- По нажатию на иконку противника тому наносится урон. По достижению нуля жизней игроку дается золото, соответствующее этому значению у противника.
- После победы над противником случайно выбирается следующий. Характеристики следующего противника модифицируются в соответствии со значениями его модификаторов и уровнем игрока.
- Игрок может тратить золото на улучшение характеристики урона.

### Теоретическая информация.

**Инкапсуляция** — один из фундаментальных принципов объектно-ориентированного программирования, который предполагает объединение данных и методов, работающих с этими данными, внутри одного класса, а также ограничение доступа к этим данным извне.

С первой частью термина вы уже познакомились в первой лабораторной работе где был рассмотрен процесс формирования класса. Что касается ограничения доступа к данным извне этот механизм реализуется благодаря следующим принципам:

**Скрытие данных** - атрибуты класса, как правило, делаются `private`, чтобы их нельзя было изменить напрямую из внешнего кода. Таким образом мы скрываем внутренние детали реализации объекта и ограничиваем доступ к его состоянию.

**Контроль доступа** - класс предоставляет контролируемые точки доступа к своим данным через публичные методы, что позволяет контролировать и управлять тем, как эти данные изменяются или используются.

В качестве примера инкапсуляции в жизни можно привести человека – у человека есть возраст и имя, которое мы не можем знать просто, посмотрев на него, эти параметры мы можем считать «приватными атрибутами». Чтобы узнать эти атрибуты мы можем воспользоваться «публичными методами» - спросить у человека его возраст или имя.

Зачем нужна инкапсуляция? На этот вопрос есть несколько ответов:

**Защита данных** – самой очевидной причиной является защита. При реализации инкапсуляции в классах объект контролирует, какие изменения можно вносить в его состояние, что предотвращает ошибки и некорректное использование.

**Упрощение отладки и тестирования** – инкапсуляция изолирует изменения в пределах класса, что уменьшает влияние на остальную часть программы. Другими словами, при возникновении ошибки в коде нам намного проще отследить причину ограничивая круг возможных причин до конкретных классов.

**Гибкость и расширяемость** – внутренние детали реализации можно изменить, не затрагивая код, который использует класс. Говоря о расширяемости – если вам нужно добавить новую функцию или расширить существующий класс, это можно сделать, не изменяя текущие методы или поля класса, что снижает вероятность внесения ошибок. Если говорить о гибкости, то внутренние детали класса могут быть изменены без необходимости изменения кода, который использует этот класс. Еще одним аспектом гибкости является возможность использовать ранее реализованные классы в других проектах.

Одним из возможных вариантов реализации инкапсуляции в языке C# вы пользовались в предыдущей лабораторной работе. Данный метод заключается в реализации приватных атрибутов в классе и публичных методов для взаимодействия с ними:

```
class Person
{
    private string name; //приватное поле

    public string getName() //метод для его получения
    {
        return name;
    }

    public void setName(string name) //метод для его установки
    {
        this.name = name;
    }
}
```

Другим вариантом реализации является использование **свойств**. Свойство – это по сути атрибут со встроенными возможностями определения функций чтения, записи и вычисления. Полное определение свойства содержит в себе два блока: **get** и **set**:

```
class Person
{
    private string name; //приватное поле

    public string Name //публичное свойство
    {
        get { return name; } //чтение поля через блок get
        set { name = value; } //запись поля через блок set
    }
    //value – значение которое передается в свойство
}
```

Пример использования свойств в коде:

```
Person person = new Person();  
  
person.Name = "FirstName"; //запись атрибута name через блок set свойства Name  
string n = person.Name; // чтение атрибута через блок get свойства Name
```

Свойства могут иметь доступ только к чтению или только к записи:

```
private string name;  
private int age;  
  
public string Name //свойство только для получения строки  
{  
    get { return name; }  
}  
  
public int Age //свойство только для записи возраста  
{  
    set { age = value; }  
}
```

В качестве функции свойства можно задавать выражения:

```
private string firstName;  
private string secondName;  
  
public string FullName  
{  
    get { return $"{firstName} {secondName}"; } //получение полного имени  
} //состоящего из имени и фамилии
```

К блокам get и set можно добавлять модификаторы доступа:

```
private string name;  
public string Name  
{  
    get { return name; } //имя можно прочитать откуда угодно  
    private set { name = value; } //имя можно записать только внутри класса  
}
```

При использовании модификаторов в свойствах следует учитывать ряд ограничений:

- Модификатор для блока set или get можно установить, если свойство имеет оба блока.
- Только один блок set или get может иметь модификатор доступа, но не оба сразу.
- Модификатор доступа блока set или get должен быть более ограничивающим, чем модификатор доступа свойства. Например, если свойство имеет модификатор public, то блок set/get может иметь только модификаторы private.

Свойства не обязательно использовать вместе с атрибутами, вместо этого можно использовать автоматические свойства, которые заменяют собой атрибуты:

```
public string Name { get; set; }
public int Age { get; set; }

public Person(string name, int age) //конструктор использует сразу свойства
{
    Name = name;
    Age = age;
}
```

### Задание. Игра кликкер.

Заданием на данную лабораторную работу является разработка игры-кликкера. Предполагается что для решения задачи будут использованы ранее реализованные классы для хранения шаблонов противников и иконок из лабораторной работы №2.

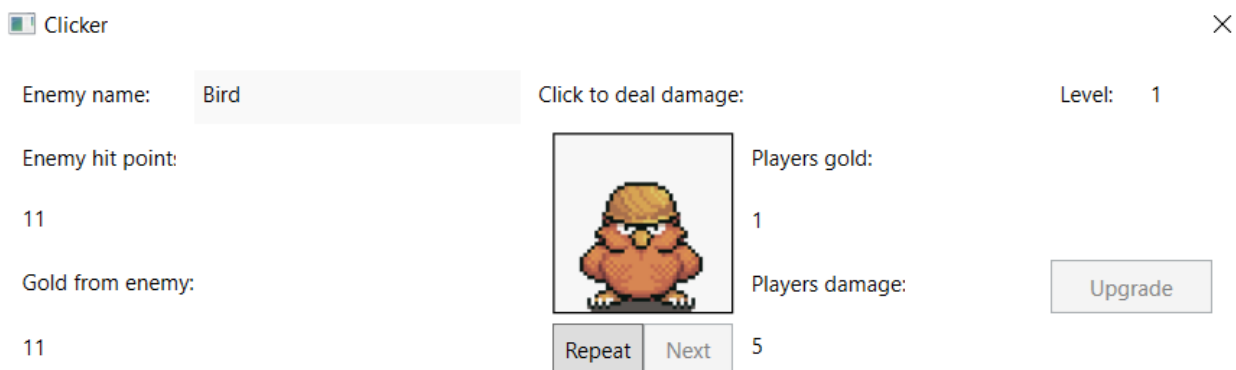


Рисунок 1 – Пример интерфейса программы

Класс для хранения больших чисел нужен для решения проблемы ограниченности встроенных типов данных: так целочисленный тип `int` поддерживает значения до примерно двух миллиардов (2,147,483,647), а тип `long` в свою очередь ограничен девятью квинтиллионами (9,223,372,036,854,775,807) – но даже такие огромные числа могут быть недостаточными для реализуемой программы.

Вместо этого стоит хранить числа по разрядам в виде массива так что каждая ячейка массива может хранить число до 1000:

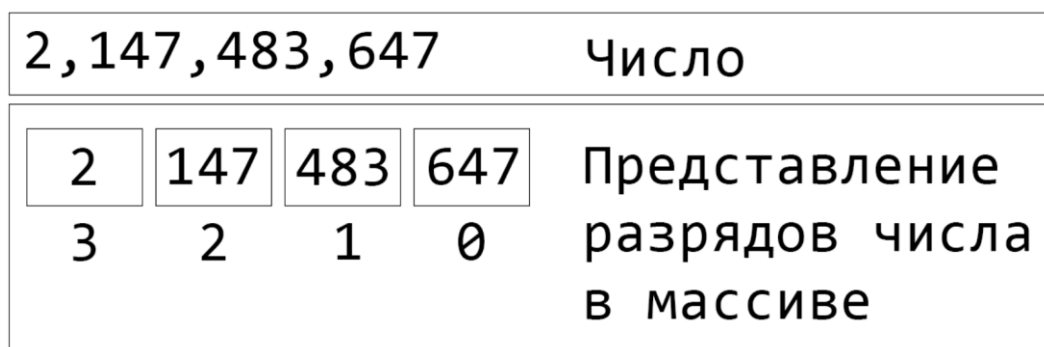


Рисунок 2 – Представление больших чисел в виде массива разрядов числа

**Сложение и вычитание** таких значений происходит так же как сложение или вычитание «столбиком» - если значения из  $n$  разряда для сложения превосходят размер текущего разряда получившаяся разница отправляется в следующий разряд. Для вычитания мы наоборот «занимаем» из разряда  $n+1$  значение если его не хватает для вычитания:

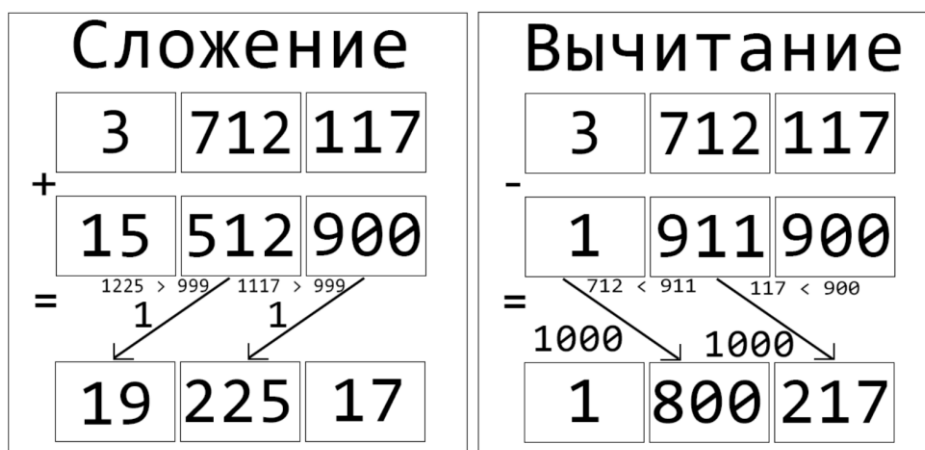


Рисунок 3 – Алгоритм сложения и вычитания для больших чисел

**Деление** на числа размером меньше чем размер разряда требуется в цикле начиная с самого большого разряда числа делить число на делитель, результат будет результатом, который будет содержать ячейка текущего разряда. Остаток, который был получен при таком делении переносится на разряд ниже умноженный на размерность разрядов. Остаток последнего разряда отбрасывается:

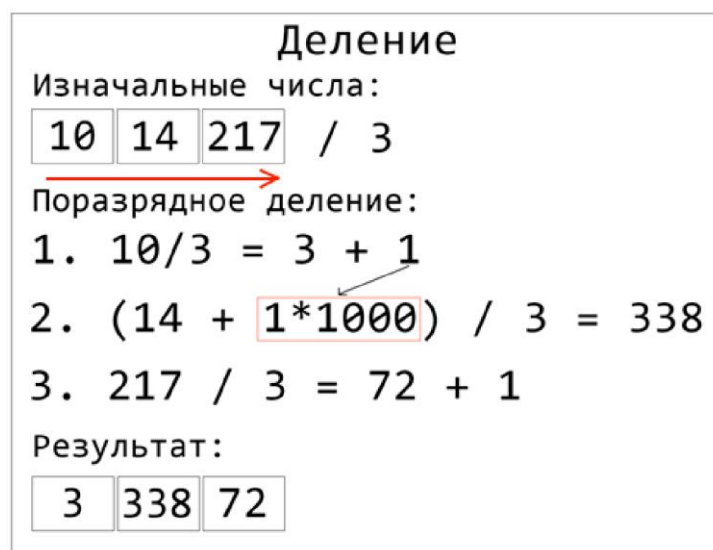


Рисунок 4 – Алгоритм деления для больших чисел

**Умножение** схоже в своем алгоритме с делением, отличие состоит в том, что алгоритм начинается с самого младшего разряда, результат умножения превышающий размер разряда переносится в следующий:

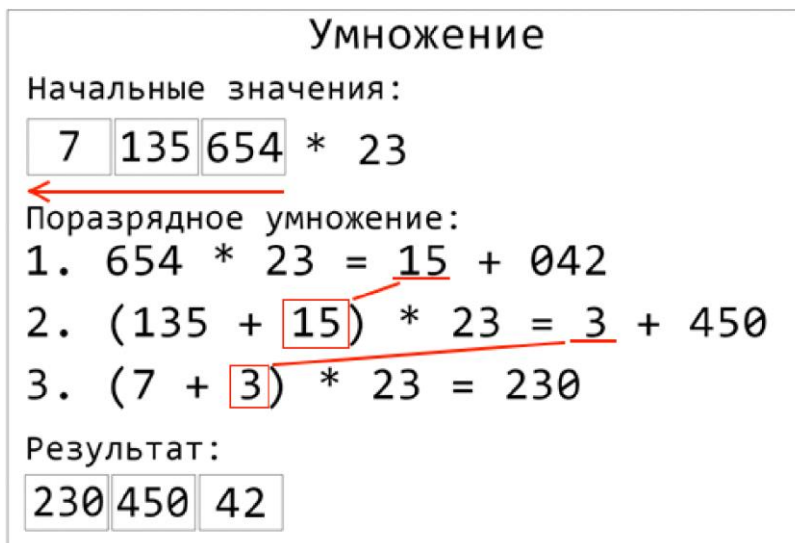


Рисунок 5 – Алгоритм умножения для больших чисел

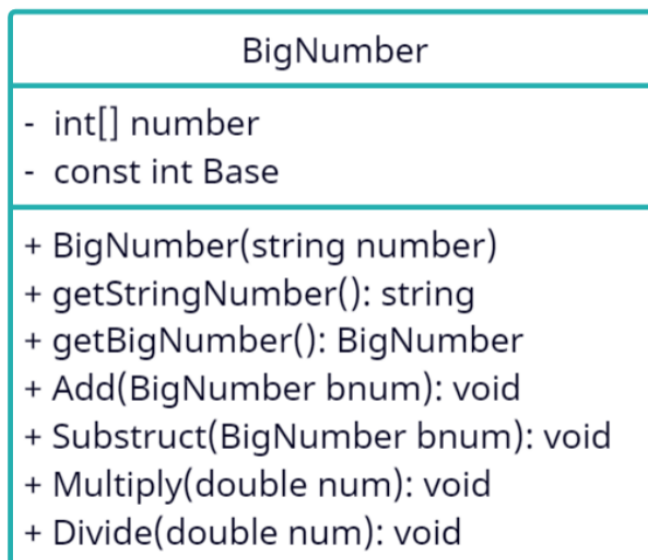


Рисунок 6 – UML-диаграмма класса больших чисел

Заметьте, что в данной диаграмме база числа указана как константа – её можно обозначить сразу в самом классе чтобы все экземпляры класса использовали у себя одинаковую основу для разряда числа. Для класса игрока можно использовать следующую структуру атрибутов. Для получения доступа к этим атрибутам лучше всего сделать их в форме свойств:

```
public class CPlayer    //класс, описывающий игрока
{
    //уровень улучшения
    int lvl;
    //количество золота у игрока
    BigNumber gold;
    //урон и модификатор урона
    BigNumber damage;
    double damageModifier;
    //цена и модификатор цены улучшения
    BigNumber upgradeCost;
    double upgradeModifier;
}
```

В качестве формул увеличения значения урона и цены улучшения можно использовать следующие формулы, но лучше придумать более усложненные уравнения:

```
upgradeCost = upgradeCost.Multiply(upgradeModifier * lvl));
damage = damage.Multiply(damageModifier);
```

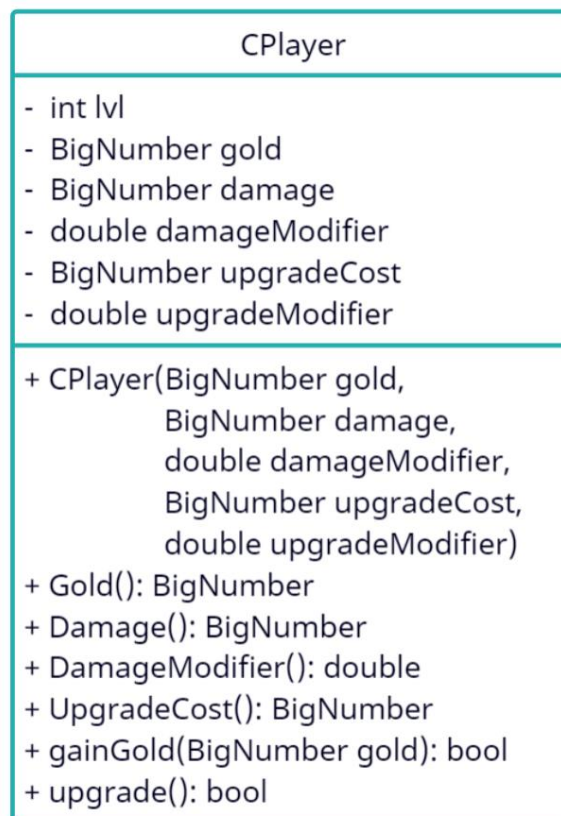


Рисунок 7 – UML-диаграмма класса игрока

Для противника требуется реализовать отдельный класс. В прошлой лабораторной работе были реализованы шаблоны противников, тут же потребуется хранить информацию о противнике (как например текущее здоровье и золото которое получит игрок за победу над противником), который будет «существовать» в сцене. Противник будет создаваться из шаблона и может иметь следующую

структуру атрибутов:

```
public class CEnemy //класс, описывающий конкретного противника
{
    string name;
    BigNumber hitPoints;
    BigNumber gold;
    Rectangle icon;
}
```

Для загрузки шаблонов и хранения готовых противников стоит реализовать отдельный класс. Одним из возможных функций для данного класса может стать функция нормализации шансов появления. Так как сумма шансов появления противников может превышать значения условных ста процентов их требуется нормализовать:

```
void normalizeChances() //нормализация шансов выбора объектов, сумма шансов
//должна быть равна 1
{
    double sum = 0;
    //enemies - List<CEnemyTemplate>
    for (int i = 0; i < enemies.Count; i++)
        sum += enemies[i].SpawnChance;

    for (int i = 0; i < enemies.Count; i++)
        enemies[i].SpawnChance /= sum;
}
```

Затем для выбора случайного шаблона можно использовать следующую функцию:

```
CEnemyTemplate findByChance(double chance) //поиск объекта по выпавшей вероятности
{
    double sum = 0;
    for(int i = 0; i < enemies.Count; i++)
    {
        sum += enemies[i].SpawnChance;

        if (sum >= chance) return enemies[i];
    }

    return null;
}
```

## Отчет:

Результат выполнения работы предоставить в виде:

- Архив с проектом (если размер архива больше 2 Мбайт, то рекомендуется загрузить проект на <https://github.com/> или на другое общедоступное хранилище и предоставить ссылку);
- Отчет по лабораторной работе в формате Microsoft Word, который содержит следующие разделы:
  1. титульный лист;
  2. задание на лабораторную работу;
  3. краткое описание разработанных программ и используемых алгоритмов со скриншотами выполнения;
  4. вывод о проделанной работе.



## Список литературы:

- 1) Герберт Шилдт "С# 4.0: полное руководство"
- 2) Эндрю Троелсен "Язык программирования С# 5.0 и платформа .NET 4.5"
- 3) Полное руководство по языку программирования С# 7.0 и платформе .NET 4.7  
<https://metanit.com/sharp/tutorial/>
- 4) Руководство по WPF <https://metanit.com/sharp/wpf/>
- 5) С# 5.0 и платформа .NET 4.5  
[http://professorweb.ru/my/csharp/charp\\_theory/level1/infocsharp.php](http://professorweb.ru/my/csharp/charp_theory/level1/infocsharp.php)
- 6) <https://github.com/Microsoft/WPF-Samples>