Лабораторная работа №2: Инкапсуляция.

Цель работы:

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

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:

- **Сокрытие 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.

Вариантом реализации инкапсуляции является использование **свойств.** Свойство — это 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 №1. Игра-кликкер.

Задачей на данную лабораторную работу является реализация игры-кликкера.

Функционал программы:

- Загрузка сохраненных шаблонов из редактора противников в качестве противников в игре при запуске программы.
- Нажатие на иконку наносит определенный урон противнику. По достижению нуля жизней игроку дается золото, соответствующее значению противника.
- После победы над противником случайно выбирается следующий. Характеристики следующего противника модифицируются в соответствии со значениями его модификаторов.
- Игрок может тратить золото на улучшение характеристики урона.

Задачи:

– Реализовать программу согласно представленному функционалу.
– Для хранения числовых данных (золото, количество жизней, урон и т.д.) использовать класс для хранения больших чисел.
– Разработать систему классов для программы. Классы должны удовлетворять принципу инкапсуляции.

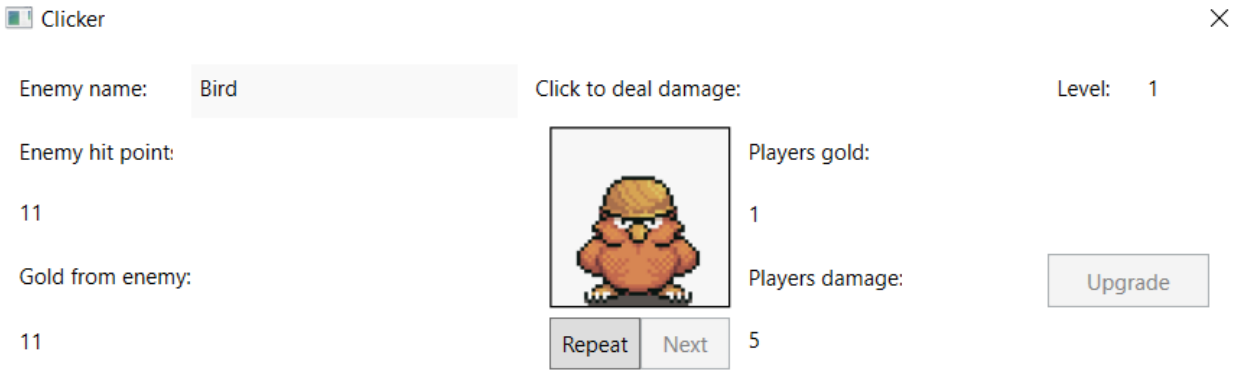


Рисунок 1 – Пример интерфейса программы

Механизм больших чисел.

Предыстория

В данной методичке предложены две формулы для роста урона и стоимости. Давайте рассмотрим их подробнее:

- Формула урона: damage = damage.Multiply(damageModifier);

Это «безобидный» экспоненциальный рост. Если damageModifier равен 1.2, то урон будет расти на 20% с каждым апгрейдом.

- Формула стоимости: upgradeCost = upgradeCost.Multiply(upgradeModifier * lvl));

Это не просто экспонента. Это суперэкспонента. Множитель, на который мы умножаем цену (уровень), сам растёт с каждым уровнем.

Начинаем со скромных 100 золота за апгрейд и upgradeModifier = 1.2.

1. Апгрейд на 2-й уровень: cost = 100 * (1.2 * 1) = 120. Норм.

2. Апгрейд на 3-й уровень: cost = 120 * (1.2 * 2) = 120 * 2.4 = 288. Уже заметно.
3. Апгрейд на 4-й уровень: cost = 288 * (1.2 * 3) = 288 * 3.6 = 1036.
4. Апгрейд на 5-й уровень: cost = 1036 * (1.2 * 4) = 1036 * 4.8 = 4972.

Мы только на пятом уровне, а цена уже выросла в 50 раз. Эта формула растёт с факториальной скоростью. Теперь представьте, что будет на 20-м уровне. Нам придётся умножать на (1.2 * 18), потом на (1.2 * 19), потом на (1.2 * 20).

К 25-му апгрейду стоимость следующего улучшения будет числом примерно с 30-ю нулями. Это в миллиард раз больше, чем максимальное значение long. Наша переменная long переполнится (пройдет через ноль и станет отрицательным числом (как в той самой истории про Ядерного Ганди в Civilization)).

Мы делаем свой BigNumber не потому, что long «может не хватить». Мы делаем его потому, что предложенная система роста гарантированно и зрелищно уничтожит любой стандартный тип данных.

Проблема: Экспоненциальный рост, который является основой «залипательных» игр, может «сжирать» стандартные типы данных за минуты геймплея. Игра становится неиграбельной, потому что её математика просто ломается.

Примечание: для работы с числами произвольной точности в платформе **.NET** существует стандартный и высокооптимизированный класс **System.Numerics.BigInteger**. В условиях промышленной разработки использование именно этой реализации является единственно верным подходом.

Основная задача заключается не в поиске наиболее эффективного решения проблемы, а в получении фундаментального понимания алгоритмов и структур данных, лежащих в основе подобных высокоуровневых абстракций.

В основе собственного класса будет лежать представление большого числа в виде композитной структуры данных — массива целочисленных типов, где каждый элемент хранит отдельный разряд числа в выбранной системе счисления (например, с основанием 1000). Арифметические операции, такие как сложение и вычитание, будут реализованы вручную на основе классических «школьных» алгоритмов, включающих поразрядную обработку и механизм переноса.

Реализация

Мы будем работать не с обычными цифрами (0-9), а с «большими цифрами» (назовем их блоки), которые могут хранить числа от 0 до 999. Наше «основание системы счисления» - 1000.

Число 3 712 117 — это массив блоков [3, 712, 117].

Число 15 512 900 — это массив блоков [15, 512, 900].

Все операции (перенос при сложении, заём при вычитании) происходят с целым числом 1000.

Пример хранения числа в виде разрядов:

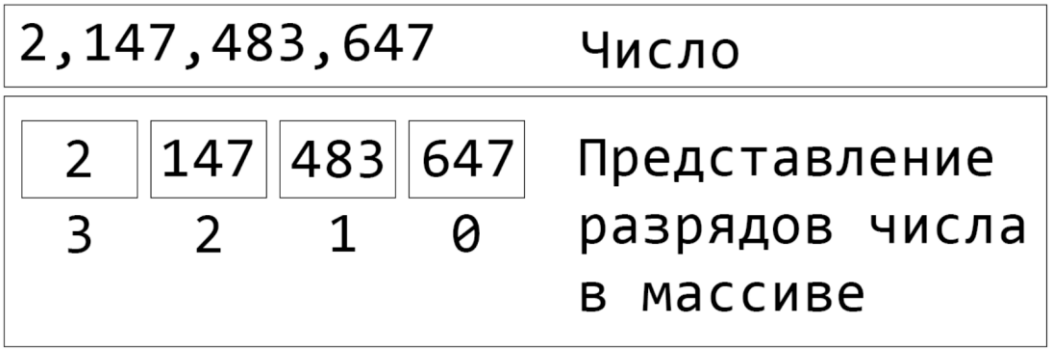


Рисунок 2 – Представление больших чисел в виде массива разрядов числа

1. Сложение

В качестве примера мы будем складывать **3 712 117 и 15 512 900.** Мы идем справа налево, как в обычном сложении в столбик.

Шаг 1: складываем самые младшие блоки (единицы)

Действие: 117 + 900 = 1017.

Результат 1017 больше или равен нашей базе 1000. Это значит, что у нас есть перенос в следующий разряд.

В результат записываем остаток от деления на 1000: 1017 % 1000 = 17.

Целую часть от деления на 1000 мы перенесем на следующий этап: 1017 / 1000 = 1. *Эта единичка в младшем разряде превратится в тысячу в более старшем.*

Промежуточный результат: ... «??? ??? 17» (так как мы используем массив чисел, мы просто не сможем хранить «017», 0 мы добавим при выводе).

Шаг 2: складываем средние блоки (тысячи)

Действие: 712 + 512 и не забываем добавить перенос с прошлого шага: 712 + 512 + 1 = 1225.

Результат 1225 снова больше или равен 1000. Значит, опять есть перенос.

В результат пишем 1225 % 1000 = 225, на следующий этап переводим остаток 1225 / 1000 = 1.

Промежуточный результат: ... «??? 225 017».

Шаг 3: складываем старшие блоки (миллионы)

Действие: 3 + 15 и добавляем перенос с прошлого шага: 3 + 15 + 1 = 19.

Результат 19 меньше 1000. Переноса нет.

В результат запишем просто 19.

Итоговый результат: собираем все вместе: [19] [225] [17] => 19 225 017.

2. Вычитание

Задача: 3 712 117 - 1 911 900

Тоже идем справа налево.

Шаг 1: вычитаем самые младшие блоки

Действие: 117 - 900.

Результат отрицательный (117 < 900). Нам не хватает. Нужно "занять" у старшего разряда (у блока 712).

Мы забираем 1 из соседнего блока 712 (он станет 711). Эта 1 в нашем разряде превращается в 1000.

Новое действие: (117 + 1000) - 900 = 1117 - 900 = 217.

В результат мы записываем **217**.

Важно: держим в уме (ну в какой-нибудь переменной), что блок 712 теперь равен 711.

Промежуточный результат: ... «??? ??? 217».

Шаг 2: вычитаем средние блоки

Действие: помним, что у нас осталось не **712, а 711**. Значит, **711 - 911**.

Нам снова не хватает (711<911). Опять нужен заём у старшего разряда (у блока 3).

Забираем 1 у блока 3 (он станет 2). Эта 1 у нас превращается в 1000.

Новое действие: (711 + 1000) - 911 = 1711 - 911 = 800.

В результат мы записываем **800**.

Важно: держим в уме, что блок 3 теперь равен 2.

Промежуточный результат: ... «??? 800 217».

Шаг 3: вычитаем старшие блоки

У нас осталось не **3, а 2**. Значит, **2 - 1 = 1.**

Всё в порядке, результат положительный, занимать ничего не нужно.

В результат записываем **1**.

Итоговый результат: собираем все вместе: [1] [800] [217] => 1 800 217.

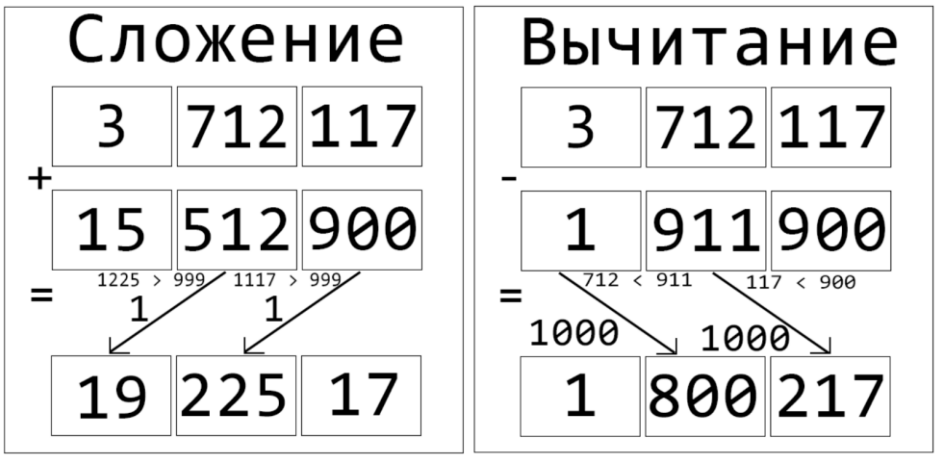


Рисунок 3 – Алгоритм сложения и вычитания для больших чисел

3. Деление

Мы имитируем школьное деление «уголком» или «в столбик», но оперируем целыми блоками.

Задача: 10 014 217 / 3

Мы будем двигаться **слева направо**, передавая остаток от деления на следующий шаг.

Шаг 1: делим старший блок

Первый блок: 10 делим на 3:

- Частное: 10 / 3 = 3. Это **первая часть** нашего итогового ответа.
- Остаток: 10 % 3 = 1. Этот остаток **нужно перенести** на следующий шаг.

Промежуточный результат: Частное = [3], Остаток для переноса = 1.

Шаг 2: делим средний блок

Второй блок: 14. К этому блоку нужно «приписать» остаток с предыдущего шага, умноженный на базу. Это как в школе мы «сносим» следующую цифру.

- Новое число = (Остаток для переноса * 1000) + Текущий блок
- Новое число = (1 * 1000) + 14 = 1014.

Делим это новое число на 3:

- Частное: 1014 / 3 = 338. Это **вторая часть** нашего ответа.
- Остаток: 1014 % 3 = 0. Этот новый остаток **переносим** на следующий шаг.

Промежуточный результат: Частное = [3, 338], Остаток для переноса = 0.

Шаг 3: делим младший блок

Третий блок: 217 Формируем последнее число для деления:

- Новое число = (Остаток для переноса * 1000) + Текущий блок
- Новое число = (0 * 1000) + 217 = 217 *(Этот момент на картинке пропущен, предполагается, что мы просто «0 * 1000»)*.

Делим новое число на 3:

- Частное: 217 / 3 = 72. Это **третья часть** нашего ответа.
- Остаток: 217 % 3 = 1. Это **финальный остаток** от деления всего числа. Мы его отбрасываем, так как он нигде не используется.

Промежуточный результат: Частное = [3, 338, 72], Финальный остаток = 1.

Шаг 4: собираем правильный ответ

Мы получили набор блоков частного: [3, 338, 72].

Чтобы превратить это в настоящее число, нужно помнить правило: **все блоки, кроме первого, должны быть дополнены нулями слева до 3-х знаков.**

- 3 -> 3
- 338 -> 338
- 72 -> 072

Итоговый результат: 3 338 072 и остаток 1.

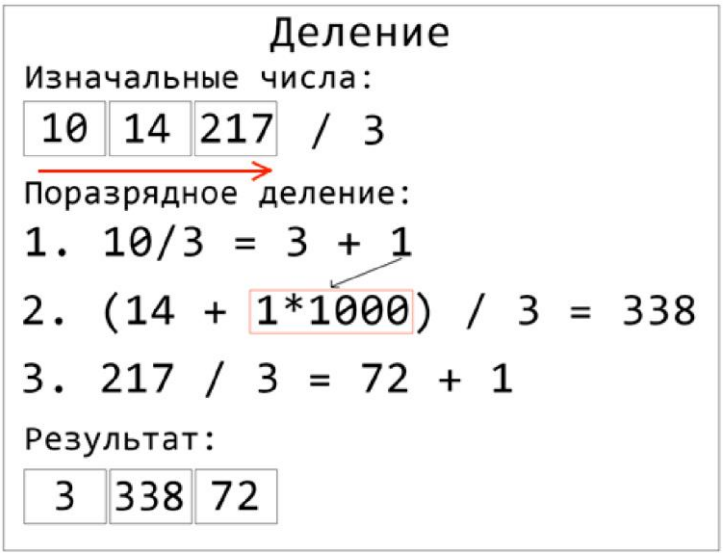


Рисунок 4 – Алгоритм деления для больших чисел

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:

Задача: **7 135 654 * 23**

Блоки: [7, 135, 654]

Множитель: 23

Мы двигаемся справа налево, от младших разрядов к старшим, и держим в уме «перенос» (carry).

Шаг 1: умножаем младший блок

Первый блок (справа): 654, умножаем его на 23:

654 * 23 = 15042.

Разделяем результат:

- **Часть для записи:** остаток от деления на 1000. 15042 % 1000 = 42. Записываем 42 в ответ.
- **Часть для переноса:** целая часть от деления на 1000. 15042 / 1000 = 15. Этот перенос пойдет на следующий шаг.

Промежуточный результат: Частное = [..., ..., 42], Перенос = 15.

Шаг 2: умножаем средний блок

Второй блок: 135, с**начала** умножаем его на 23:

135 * 23 = 3105.

Теперь к результату прибавляем перенос с прошлого шага:

3105 + 15 = 3120.

Разделяем новый результат:

- Часть для записи в ответ: 3120 % 1000 = 120.

- Часть для переноса: 3120 / 1000 = 3.

Промежуточный результат: Частное = [..., 120, 42], Перенос = 3.

Шаг 3: умножаем старший блок

Последний блок: 7, умножаем его на 23:

7 * 23 = 161.

Прибавляем перенос:

161 + 3 = 164.

Разделяем:

- Часть для записи в ответ: 164 % 1000 = 164. Так как это последний шаг, мы записываем все число.
- Часть для переноса: 164 / 1000 = 0. Переноса больше нет.

Промежуточный результат: Частное = [164, 120, 42].

Шаг 4: собираем правильный ответ

Соединяем все полученные блоки: [164] [120] [42].

Итоговый правильный результат для вывода на экран: 164 120 042.

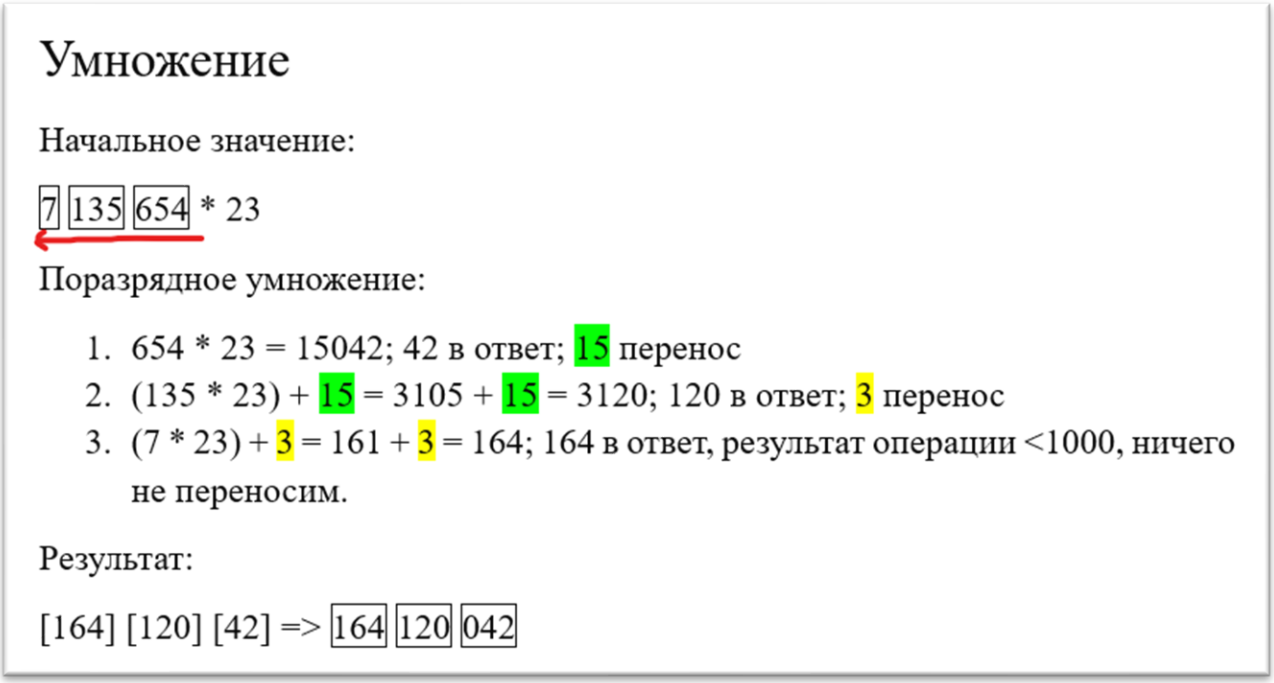


Рисунок 5 – Алгоритм умножения для больших чисел

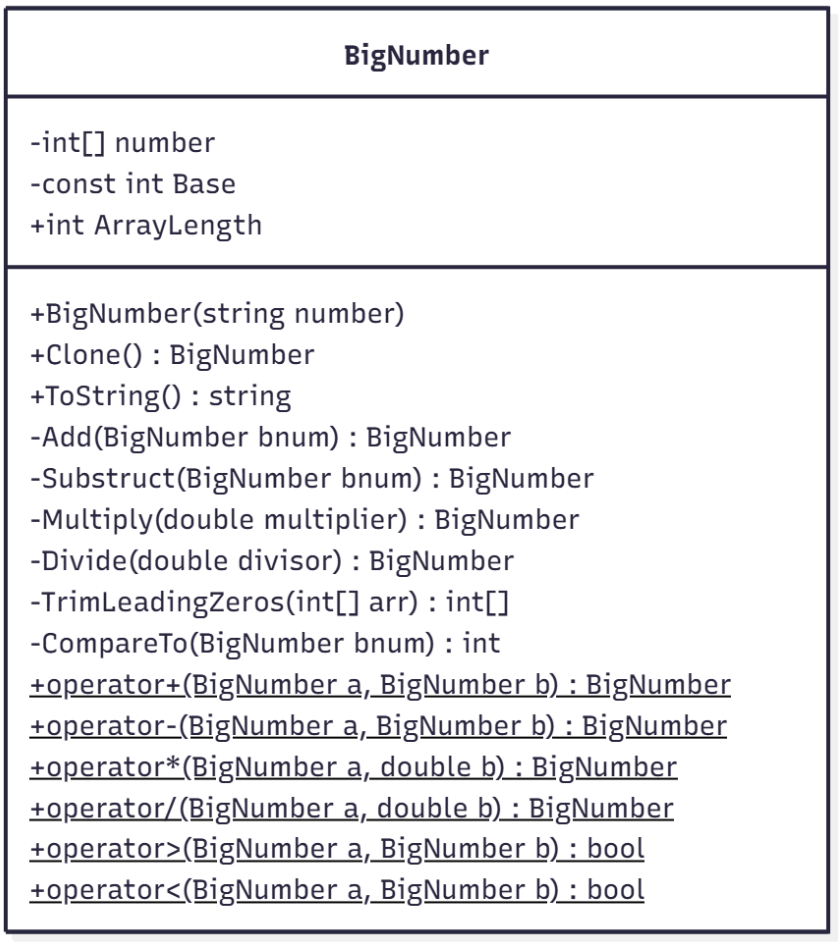


Рисунок 6 – UML-диаграмма класса больших чисел

Концепция: Мы пишем свой собственный тип данных для работы с огромными числами. Пользователь этого класса (другой программист или вы сами в будущем) может даже не задумываться о том, как он устроен внутри. Мы скрываем логику, например, сложения в отдельный метод **(икапсуляция)** и предоставляем ему некие рычаги управления **(абстракция)** нашей моделью **BigNumber.**

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.

Принцип неизменяемости (immutability)

Объект не должен меняться после создание, вместо этого создается новый объект, и работа продолжается с ним.

В нашем классе больших чисел этот принцип реализуется в методах «математических операций». Мы должны не изменять текущий объект, а создавать новый объект (результат операции). Поэтому тип возвращаемого значения для мат операций будет **BigNumber.**

```
// Метод сложения двух больших чисел (немутирующий)
private BigNumber Add(BigNumber bnum)
{
    // тут будет ваша
    // логика сложения двух BigNumber.
    // Результат записывается
    // в новую переменную BigNumber

    return new BigNumber(“Ваш результат”);
}
```

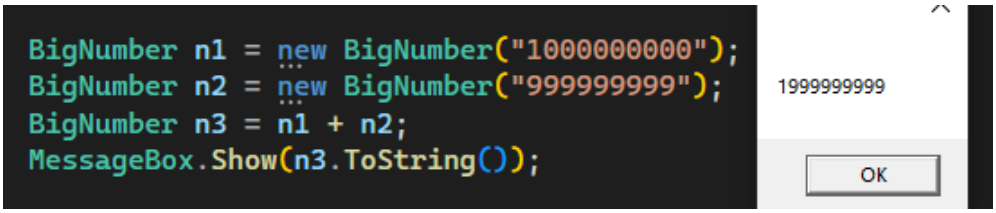
Переопределение операторов (+,-,*,/)

Для класса мы можем переопределить логику поведения работы с операторами.

```
public static BigNumber operator +(BigNumber a, BigNumber b)
{
    return a.Add(b);
}
```

Данный оператор может использовать готовую логику сложения чисел из метода **Add,** сам метод может быть приватным. Тогда для сложения мы сможем использовать следующий код:

```
BigNumber n1 = new BigNumber("1000000000");
BigNumber n2 = new BigNumber("999999999");
BigNumber n3 = n1 + n2;
```



Переопределение ToString()

Каждый объект в .NET унаследован от базового класса **System.Object**. У этого прародителя всех объектов есть виртуальный метод **ToString()**, и по умолчанию он выводит полное имя типа. Ваш класс, не имея собственной версии этого метода, просто использует эту версию, и при попытке сделать **n3.ToString()** мы увидим **«YourProject.BigNumber»** - примерно такое.

Для решения этой проблемы и использования более «привычного» метода для преобразования в строку мы должны **переопределить (override)** стандартное поведение метода **ToString()**.

```
public override string ToString()
{
    StringBuilder sb = new StringBuilder();
    for (int i = number.Length - 1; i >= 0; i--)
    {
        if (i == number.Length - 1)
            sb.Append(number[i].ToString());
        else
            sb.Append(number[i].ToString("D3"));
    }
    sb.ToString().TrimStart('0');
    return sb.ToString();
```

```
}
```

Теперь «`MessageBox.Show(n3.ToString());`» выведет «**1999999999**» из операции предыдущего этапа (где мы переопределили поведение для «+»).

Для класса игрока можно использовать следующую структуру атрибутов. Для получения доступа к этим атрибутам лучше всего сделать их в форме свойств.
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);
```

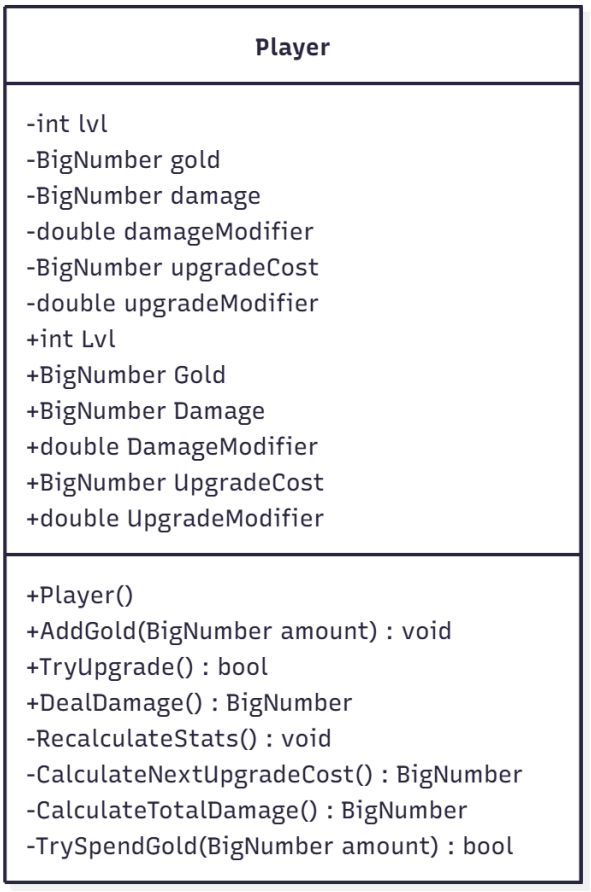


Рисунок 7 – UML-диаграмма класса игрока

Обратите внимание: все свойства имеют **private set**. Это значит, что внешний код может только читать состояние игрока, но изменять его можно только через публичные методы (TryUpgrade, AddGold). Это и есть инкапсуляция.

Класс игрока инкапсулирует логику работы с данными в методы, как это делал **BigNumber.** Сам класс игрока должен отвечать за свое состояние и его изменение. Из главного окна (MainWindow) мы должны лишь давать команду на изменение, а «игрок» сам решает «получилось ли увеличить уровень?».

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. Также этот класс, как и предыдущие классы, должен сам отвечать за свое состояние.

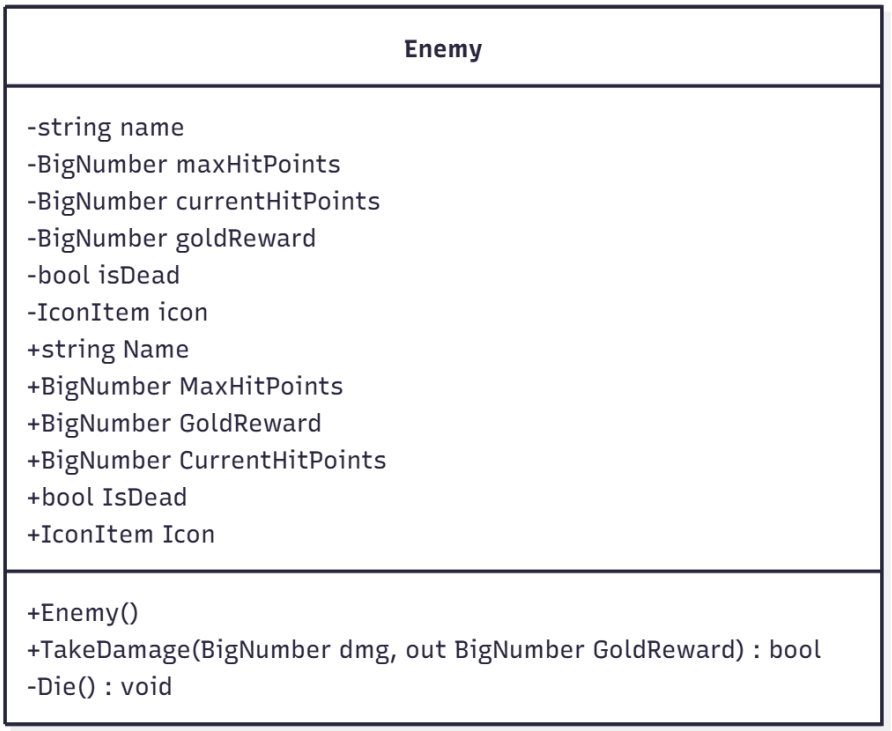


Рисунок 8 – UML-диаграмма класса протвника

Обратите внимание: все свойства имеют **private set**. Это значит, что внешний код может только читать состояние игрока, но изменять его можно только через публичные методы (TakeDamage).

Функция TakeDamage может возвращать true/false, если противник был побежден при атаке (урон>здоровье). В случае true в out аргумент можно записать награду за победу над противником.

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;
        }
```