Панов Михаил Александрович

к.э.н., доцент кафедры информационных технологий и статистики
«УрГЭУ»

panov79@ya.ru

Создадим аналог «Морского боя»: игра «Потопи сайт»

Часть
1.

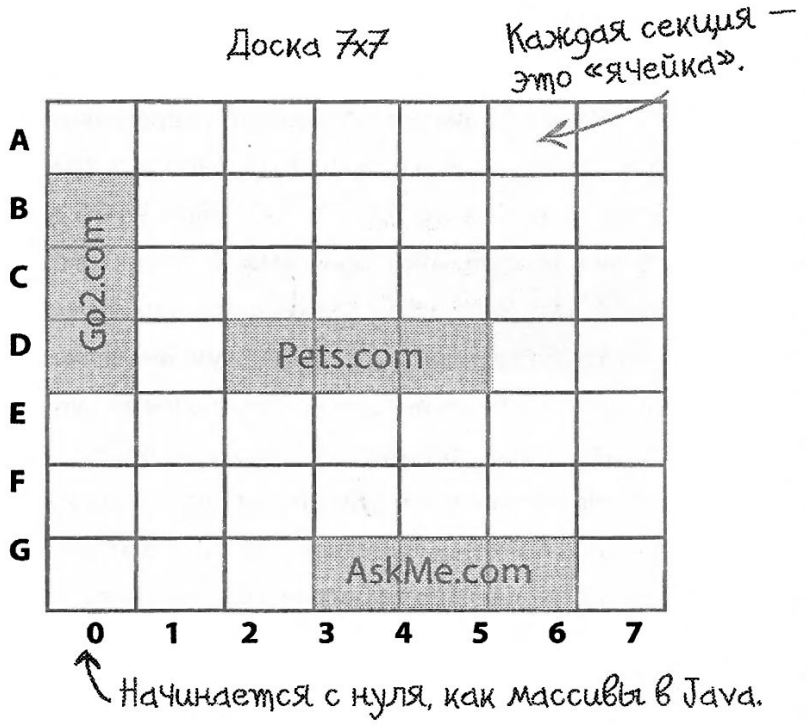
Создадим аналог «Морского боя»: игра «Потопи сайт»

В этой игре вы будете играть против компьютера. Основное отличие от оригинального «Морского боя» состоит в том, что тут не нужно размещать свои корабли. Вместо этого придется потопить корабли компьютера за минимальное количество ходов. Кроме того, вы будете топить не корабли, а «сайты».

Цель: потопить все «сайты» компьютера за минимальное количество попыток. Вы будете получать баллы в зависимости от того, насколько хорошо играете.

Подготовка: при загрузке программы компьютер разместит три «сайта» на виртуальной доске (7x7). После этого вас попросят сделать первый ход.

Как играть: вы пока не научились создавать GUI (графический пользовательский интерфейс), поэтому данная версия будет работать в командной строке. Компьютер предложит вам ввести предполагаемую ячейку в виде A3, C5 и т. д. В ответ на это в командной строке вы получите результат — либо «Попал», либо «Мимо», либо «Вы потопили Pets.com» (или любой другой «сайт», который вам посчастливилось потопить в этот день). Как только вы отправите на корм рыбам все три «сайта», игра завершится выводом на экран вашего рейтинга.



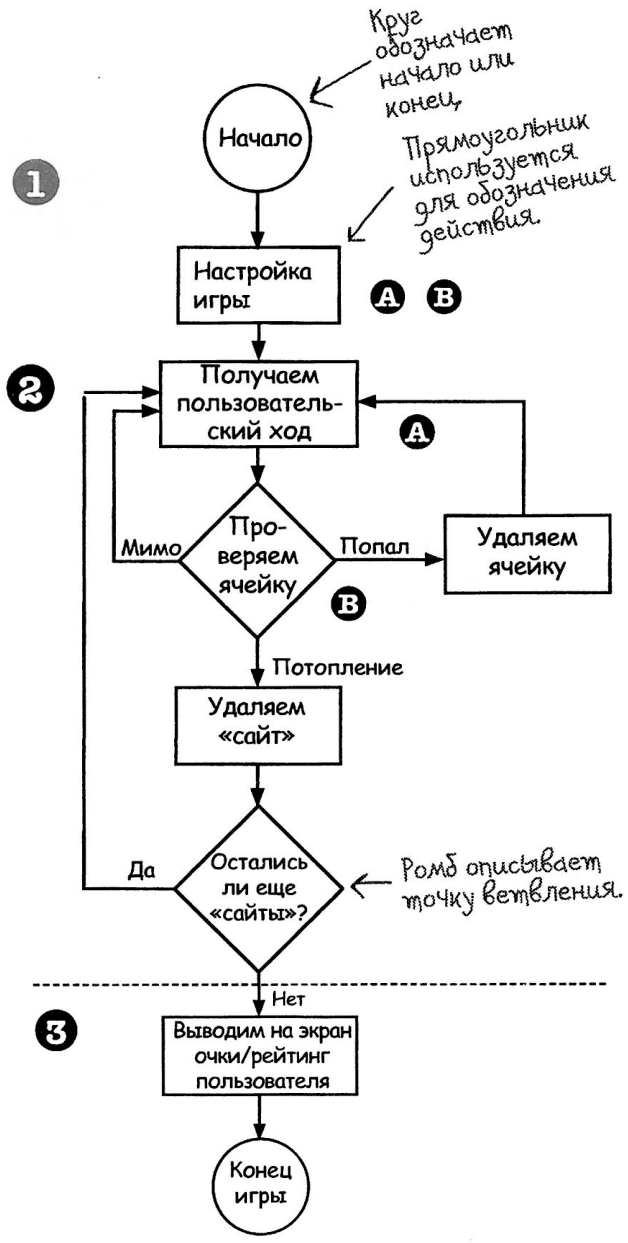
Фрагмент игрового диалога

```
File Edit Window Help Sell
%java DotComBust
Сделайте ход  A3
Мимо
Сделайте ход  B2
Мимо
Сделайте ход  C4
Мимо
Сделайте ход  D2
Попал
Сделайте ход  D3
Попал
Сделайте ход  D4
Ой! Вы потопили Pets.com  :(
Потопил
Сделайте ход  B4
Мимо
Сделайте ход  G3
Попал
Сделайте ход  G4
Попал
Сделайте ход  G5
Ой! Вы потопили AskMe.com  :(
```

Понятно, что нужны классы и методы, но какими они должны быть? Чтобы ответить на этот вопрос, необходимо получить больше информации о том, как игра должна себя вести.
Прежде всего нужно определиться с игровым процессом. Рассмотрим общую идею.

Итак, вы получили представление о том, что должна делать программа. Далее необходимо выяснить, какие для этого понадобятся объекты. Думайте об этом, как Брэд, а не как Ларри; в первую очередь сосредотачивайтесь на элементах, из которых состоит программа, а не на процедурах.

1. Пользователь запускает игру.
   - A. Игра создает три «сайта».
   - B. Игра размещает «сайты» на виртуальной игровой доске.
2. Игра начинается.
   Повторять следующие действия, пока не останется ни одного «сайта».
   - A. Предложить пользователю сделать ход (A2, C0 и т. д.).
   - B. Проверить все «сайты» на попадание, промах и потопление. Выбрать подходящее действие: если попадание — удалить ячейку (A2, D4 и т. д.). Если потопление — удалить «сайт».
3. Игра заканчивается.
   Показать пользователю рейтинг, основываясь на количестве попыток.



Ух ты! Настоящая блок-схема.

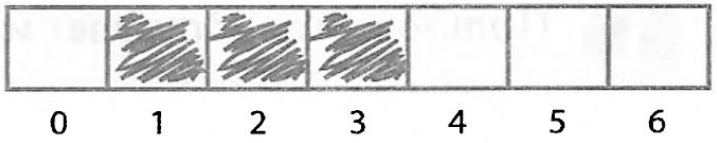
Похоже, потребуется минимум два класса — Game и DotCom. Но прежде чем написать полноценную игру, создадим ее упрощенную версию. В этой версии все просто. Вместо двумерной сетки мы будем использовать единственный ряд. И вместо трех «сайтов» у нас будет только один. Тем не менее принцип игры остается тем же: нужно создать экземпляр класса DotCom, присвоить ему положение где-нибудь в ряду, получить пользовательский ввод и, когда все три ячейки «сайта» будут поражены, закончить игру.

Упрощенная версия дает хороший старт для создания полноценной игры. Создав что-то маленькое, позже можно расширить и усложнить это. В текущем варианте класс игры не содержит переменных экземпляра, а весь игровой код находится в методе main(). Иными словами, после запуска игры и вызова главного метода будет создан только один экземпляр класса DotCom, после чего для него будет выбрано положение (три последовательные ячейки виртуального ряда). Далее программа предложит пользователю сделать ход, проверит его вариант, и предыдущие два действия станут повторяться, пока все три клетки не будут поражены.

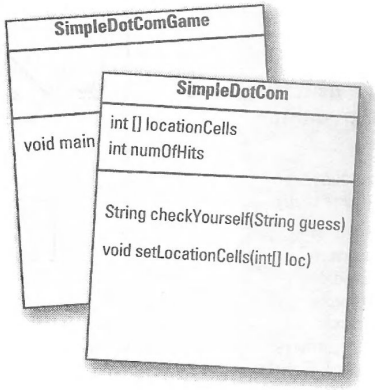
Не забывайте, что виртуальный ряд остается виртуальным, то есть не существует нигде в программе. Пока игре и пользователю известно, что «сайт» спрятан в трех последовательных ячейках из семи возможных (начиная с нулевой), ряд как таковой не нуждается в программном представлении. Может возникнуть соблазн создать массив из семи элементов и присвоить трем из них числа, представляющие «сайт», но это необязательно делать. Нужен массив, который хранит лишь три клетки, занимаемые «сайтом».

1 **Игра запускается и создает один «сайт»,** присваивая ему адрес из трех ячеек в ряду.

Вместо A2, C4 и подобных обозначений положение сайта представлено числами. Например, местоположение 1, 2, 3 показано на следующей картинке:



2 **Начинается игровой процесс.** Предлагаем пользователю сделать ход, после чего проверяем, попал ли он в одну из трех ячеек «сайта». Если попал, то увеличиваем значение переменной numOfHits на 1.



3 **Игра завершается,** если все три ячейки поражены (значение переменной numOfHits достигло 3), а пользователю сообщается, сколько ходов ему потребовалось для потопления «сайта»

Полная версия игрового диалога.

```
File Edit Window Help Destroy
%java SimpleDotComGame
Введите число  2
Попал
Введите число  3
Попал
Введите число  4
Мимо
Введите число  1
Потопил
Вам потребовалось 4 попыток(и)
```

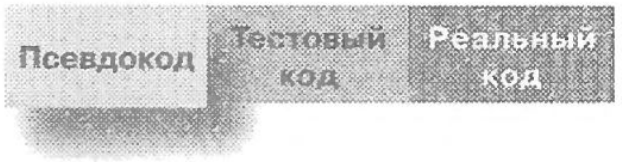
Разработка класса

- ☐ **Выясняем, что должен *делать* класс.**
- ☐ **Перечисляем переменные экземпляра и методы.**
- ☐ **Пишем псевдокод для методов (очень скоро вы увидите, о чем речь).**
- ☐ **Пишем тестовый код для методов.**
- ☐ **Реализуем класс.**
- ☐ **Тестируем методы.**
- ☐ **Отлаживаем и при необходимости корректируем.**
- ☐ **Радуемся, что не нужно проверять нашу так называемую *учебную программу* на реальных пользователях.**

Три кода, которые мы напишем для каждого класса:

Эта полоса будет размещена на следующих страницах, чтобы напоминать вам, над какой частью вы в данный момент работаете. Например, если вы видите ее вверху страницы, это означает, что вы пишете псевдокод для класса SimpleDotCom.

Класс SimpleDotCom

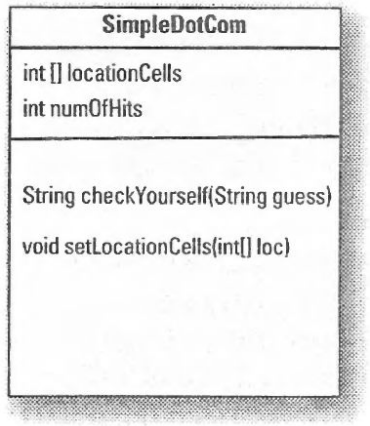


Псевдокод
Алгоритм, который поможет вам сосредоточиться на логике, не вникая в синтаксис.

Тестовый код
Класс или метод, с помощью которого можно проверять реальный код и подтверждать, что он работает правильно.

Реальный код
Непосредственная реализация класса. Это рабочий код на языке Java.

Псевдокод | Тестовый код | Реальный код



Ознакомившись с этим примером, вы поймете, как работает вариант псевдокода. Это что-то среднее между реальным кодом на языке Java и описанием класса на человеческом языке. Большая часть псевдокода содержит три раздела: объявление переменных экземпляра, объявление методов, логика методов. Из этих трех составляющих наиболее важна последняя, так как в ней описывается, что должно произойти.

ОБЪЯВЛЯЕМ целочисленный массив для хранения адреса ячеек. Даем ему имя *locationCells*.
ОБЪЯВЛЯЕМ переменную типа int для хранения количества попаданий. Называем ее *numOfHits* и **ПРИСВАИВАЕМ** ей 0.

ОБЪЯВЛЯЕМ метод *checkYourself()*, который принимает ход пользователя в качестве параметра String (1, 3 и т. д.), проверяет его и возвращает результат: «Попал», «Мимо» или «Потопил».
ОБЪЯВЛЯЕМ сеттер *setLocationCells()*, который принимает целочисленный массив (хранящий адрес трех ячеек в виде переменных типа int — 2, 3,4 и т. д.).

МЕТОД: ***String checkYourself(String userGuess)***
ПОЛУЧАЕМ ход пользователя в виде строкового параметра.
ПРЕОБРАЗУЕМ полученные данные в тип int.
ПОВТОРЯЕМ это с каждой ячейкой массива.
//СРАВНИВАЕМ ход пользователя с местоположением клетки.
ЕСЛИ пользователь угадал,
ИНКРЕМЕНТИРУЕМ количество попаданий.
//ВЫЯСНЯЕМ, была ли это последняя ячейка.
ЕСЛИ количество попаданий равно 3, **ВОЗВРАЩАЕМ** «Потопил».
ИНАЧЕ потопления не произошло, значит, **ВОЗВРАЩАЕМ** «Попал».
КОНЕЦ ВЕТВЛЕНИЯ
ИНАЧЕ пользователь не попал, значит, **ВОЗВРАЩАЕМ** «Мимо».
КОНЕЦ ВЕТВЛЕНИЯ
КОНЕЦ ПОВТОРЕНИЯ
КОНЕЦ МЕТОДА

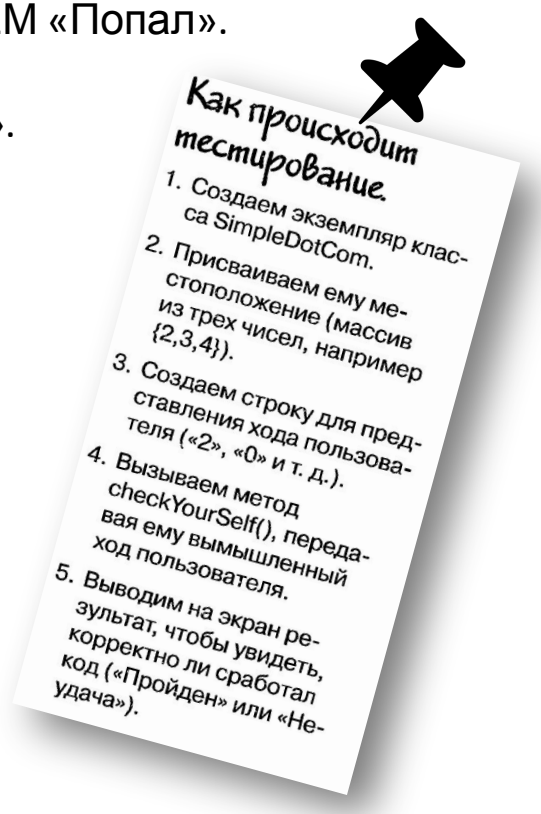
МЕТОД: ***void setLocationCells(int[] cellLocations)***
ПОЛУЧАЕМ адреса ячеек в виде параметра с целочисленным массивом.
ПРИСВАИВАЕМ полученный параметр полю, хранящему адреса ячеек.
КОНЕЦ МЕТОДА

Псевдокод | Тестовый код | Реальный код

Нужно написать тестовый код, который позволит создать объект *SimpleDotCom* и запустить его методы. В данном случае нам интересен только метод *chackYourSelf()*, но для его правильной работы нам придется также реализовать метод *setLocationCells()*. Внимательно изучите псевдокод для метода *chackYourSelf()*, приведенный ниже. Здесь метод *setLocationCells()* — обычный сеттер, поэтому мы не обращаем на него внимания. Однако в реальном приложении может понадобиться более сложный метод с необходимостью тестирования.

Теперь спросите себя: «Если метод *checkYourSelf()* уже реализован, какой тестовый код я могу написать, чтобы убедиться в его корректной

МЕТОД: ***String checkYourself(String userGuess)***
ПОЛУЧАЕМ ход пользователя в виде строкового параметра.
ПРЕОБРАЗУЕМ полученные данные в *int*.
ПОВТОРЯЕМ это с каждой ячейкой массива.
    // СРАВНИВАЕМ ход пользователя с адресом ячейки.
    ЕСЛИ ход пользователя совпал,
        ИНКРЕМЕНТИРУЕМ количество попаданий.
        //ВЫЯСНЯЕМ, была ли это последняя ячейка.
        ЕСЛИ количество попаданий равно 3, ВОЗВРАЩАЕМ «Потопил»
        ИНАЧЕ потопления не произошло, то ВОЗВРАЩАЕМ «Попал».
        КОНЕЦ ВЕТВЛЕНИЯ
    ИНАЧЕ пользователь не попал, **ВОЗВРАЩАЕМ** «Мимо».
    КОНЕЦ ВЕТВЛЕНИЯ
КОНЕЦ ПОВТОРЕНИЯ
КОНЕЦ МЕТОДА



Это не глупые вопросы

В: Может быть, я что-то упустил, но объясните, как именно вы тестируете элементы, которых даже не существует?

О: Вы ничего не упустили. Мы никогда и не говорили, что начинать нужно с *выполнения* тестов; вы начинаете с их *написания*. При создании тестового кода у вас нет ничего, на чем можно его опробовать, так что вы, вероятно, не сумеете его скомпилировать, пока не напишете формально работающую «заглушку», но это всегда будет приводить к провалу теста (например, возвращать null).

В: Я все еще не понимаю, в чем суть. Почему бы не подождать, пока код будет написан, и только затем добавлять тесты?

О: Тщательное обдумывание тестового кода (и его написания) помогает более четко представить, какие действия требуются от метода.

После реализации метода вам останется только проверить его с помощью уже готового тестового кода. Кроме того, вы *знаете*, что если не напишете его сейчас, то не напишете уже *никогда*. Всегда найдется более интересное занятие.

Старайтесь писать небольшой тестовый код, после чего создавайте реализацию лишь в том объеме, который позволит ей пройти этот тест. Затем напишите *чуть больше* тестового кода и создайте новую реализацию для его прохождения. На каждом этапе запускаются все уже написанные тесты, поэтому вы всегда можете быть уверены, что последние изменения не нарушают код, протестированный ранее.

Тестовый код для класса SimpleDotCom

```
public class SimpleDotComTestDrive {

   public static void main (String[] args) {

      SimpleDotCom dot = new SimpleDotCom();

      int[] locations = {2,3,4};

      dot.setLocationCells(locations);

      String userGuess = "2";
      String result = dot.checkYourself(userGuess);

      String testResult = "Неудача";

      if (result.equals("Попал") ) {

          testResult = "Пройден";

       }

       System.out.println(testResult);

   }

}
```

Создаем экземпляр класса SimpleDotCom.

Создаем массив для местоположения «сайта» (три последовательных числа из семи).

Вызываем сеттер «сайта».

Делаем ход от имени пользователя.

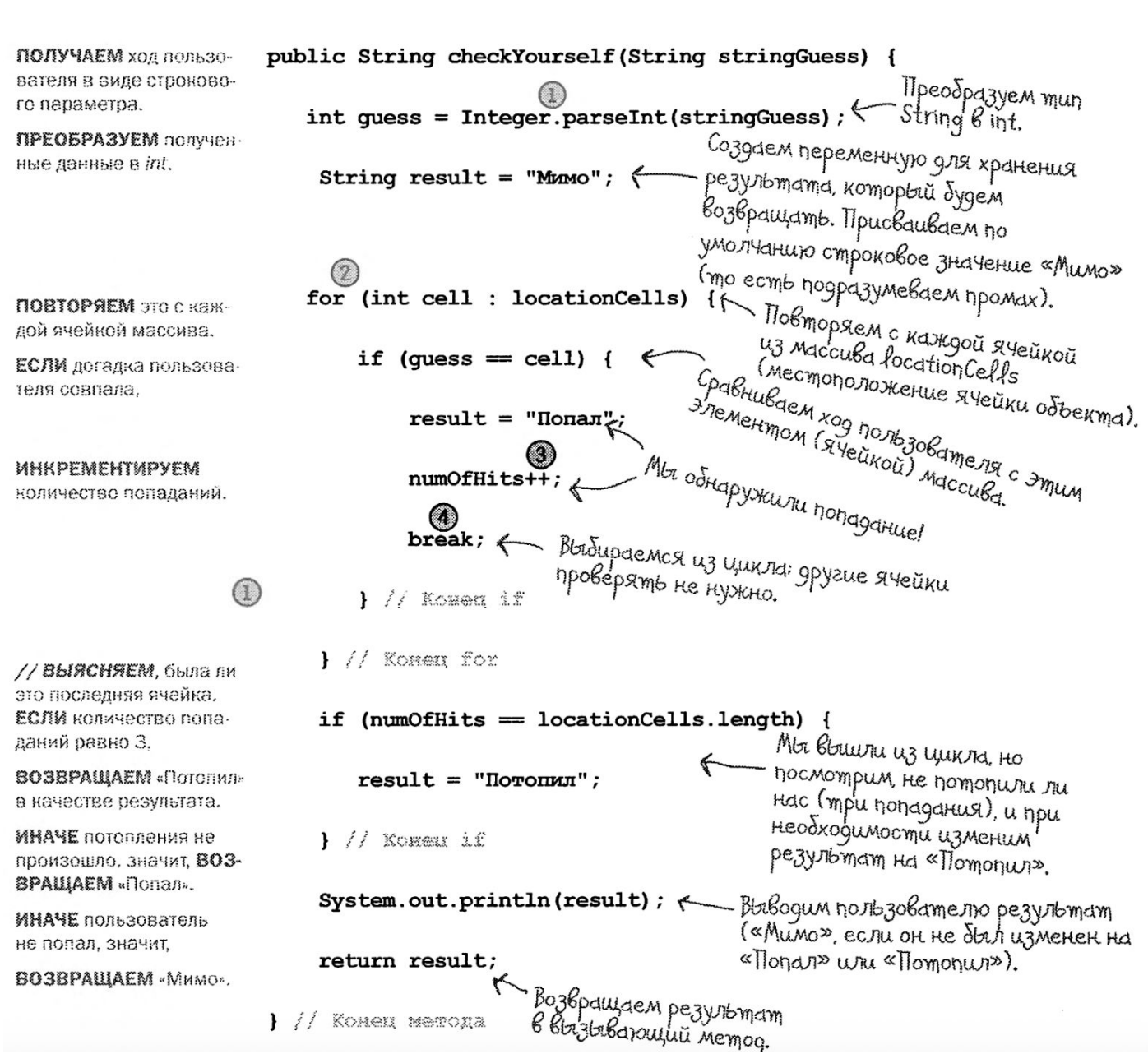
Вызываем метод checkYourself() объекта SimpleDotCom.

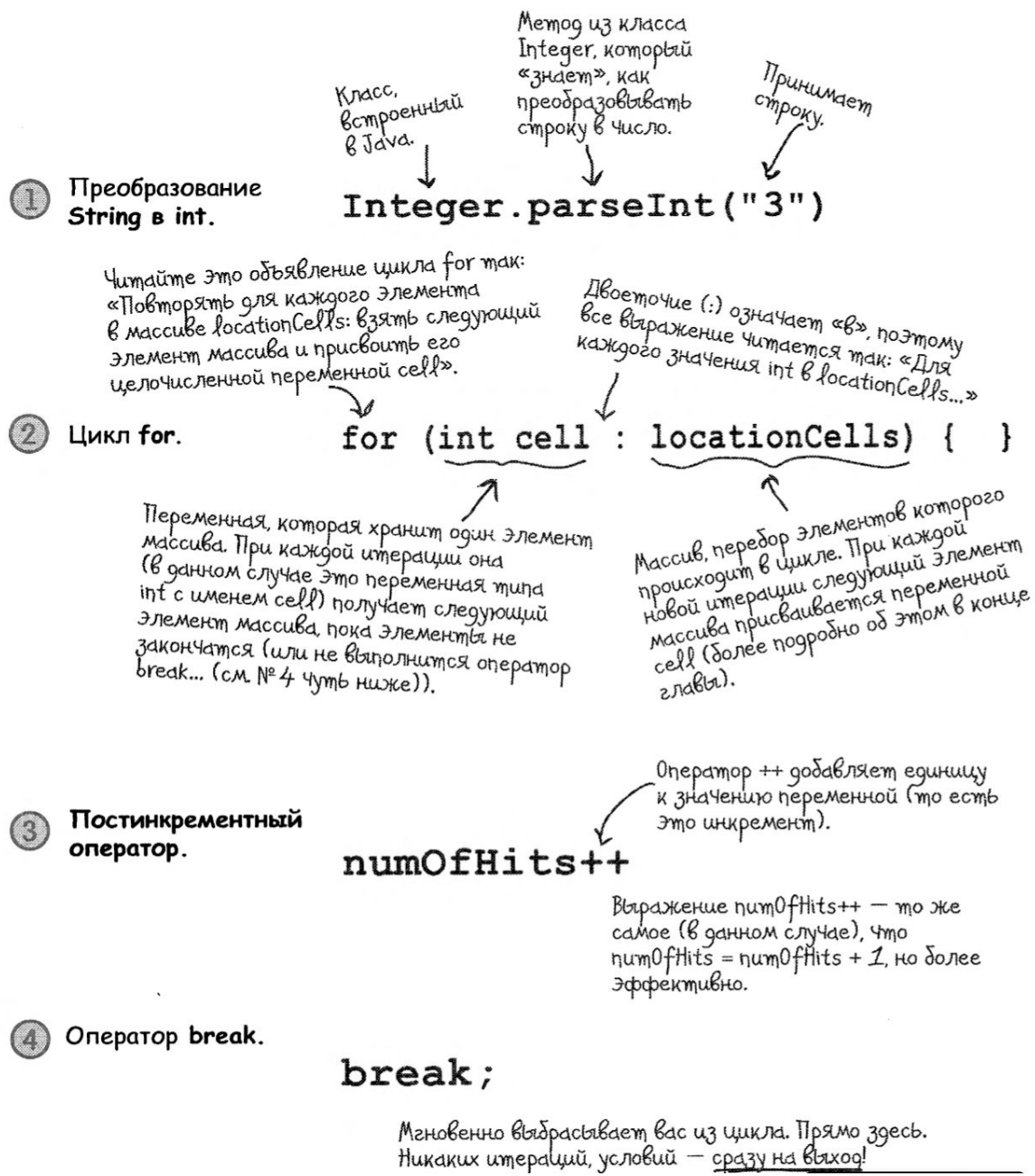
Если ход (2) возвращает строку «Попал», значит, все работает

Напечатайте результат прохождения теста («Пройден» или «Неудача»).

Псевдокод | Тестовый код | Реальный код

Не существует идеального перехода от псевдокода к коду на языке Java всегда будут вноситься некоторые изменения. Псевдокод позволил лучше понять, что должен делать метод, и теперь нужно подобрать Java-код, с помощью которого можно выразить, как ему это следует делать. Посмотрите на этот код и подумайте, что бы вам хотелось в нем улучшить. Числами ① обозначаются такие особенности синтаксиса и языка, с которыми вы еще не знакомы. Они будут подробно разобраны на следующем слайде.




Класс, встроенный в Java.
Метод из класса Integer, который «знает», как преобразовывать строку в число.
Принимает строку.
1 Преобразование String в int.
Integer.parseInt("3")
Читайте это объявление цикла for так: «Повторять для каждого элемента в массиве locationCells: взять следующий элемент массива и присвоить его целочисленной переменной cell».
Двоеточие (:) означает «в», поэтому все выражение читается так: «Для каждого значения int в locationCells...»
2 Цикл for.
for (int cell : locationCells) { }
Переменная, которая хранит один элемент массива. При каждой итерации она (в данном случае это переменная типа int с именем cell) получает следующий элемент массива, пока элементы не закончатся (или не выполнится оператор break... (см. № 4 чуть ниже)).
Массив, перебор элементов которого происходит в цикле. При каждой новой итерации следующий элемент массива присваивается переменной cell (более подробно об этом в конце главы).
Оператор ++ добавляет единицу к значению переменной (то есть это инкремент).
3 Постинкрементный оператор.
numOfHits++
Выражение numOfHits++ — то же самое (в данном случае), что numOfHits = numOfHits + 1, но более эффективно.
4 Оператор break.
break;
Мгновенно выбрасывает вас из цикла. Прямо здесь. Никаких итераций, условий — сразу на выход!

Окончательный вариант кода для SimpleDotCom и SimpleDotComTester

```
public class SimpleDotComTestDrive {

   public static void main (String[] args) {
      SimpleDotCom dot = new SimpleDotCom();
      int[] locations = {2,3,4};
      dot.setLocationCells(locations);
      String userGuess = "2";
      String result = dot.checkYourself(userGuess);
    }
}
```

```
public class SimpleDotCom {

   int[] locationCells;
   int numOfHits = 0;

   public void setLocationCells(int[] locs) {
      locationCells = locs;
   }

   public String checkYourself(String stringGuess) {
      int guess = Integer.parseInt(stringGuess);
      String result = "Мимо";
      for (int cell : locationCells) {
         if (guess == cell) {
            result = "Попал";
            numOfHits++;
            break;
         }
      } // Выходим из цикла

      if (numOfHits ==
          locationCells.length) {
          result = "Потопил";
      }
      System.out.println(result);
         return result;
    } // Завершаем метод
} // Завершаем класс
```

Здесь есть небольшая ошибка. Программа компилируется и запускается, но иногда... Сейчас не будем об этом думать, но *придется* разобраться с проблемой чуть позже.

Что мы должны увидеть при запуске этого кода?

Тестовый код создает объект SimpleDotCom и передает ему местоположение под номерами 2, 3, 4. Затем он шлет вымышленный пользовательский код 2 в метод checkYourSelf(). Если код работает правильно, то мы должны увидеть следующий результат:

```
java SimpleDotComTestDrive
Попал
```

Псевдокод | Тестовый код | Реальный код

Псевдокод для класса SimpleDotComGame

Все это находится внутри main()

Есть некоторые вещи, которые нужно принять на веру. Например, одна из строк псевдокода гласит: «ПОЛУЧАЕМ пользовательский ввод из командной строки». На текущий момент это немного больше, чем нам нужно реализовать. К счастью, мы используем ООП, то есть можно попросить другой класс/объект о выполнении определенного действия и не задумываться, как именно он это сделает. При написании псевдокода вы должны понимать, что когда-нибудь у вас будет возможность сделать что угодно, а сейчас нужно направить все умственные усилия на продумывание логики.

public static void main (String [] args)

ОБЪЯВЛЯЕМ переменную *numOfGuesses* типа int для хранения количества ходов пользователя.

СОЗДАЕМ новый экземпляр класса SimpleDotCom.

ВЫЧИСЛЯЕМ случайное число от 0 до 4 для местоположения начальной ячейки.

СОЗДАЕМ целочисленный массив с тремя элементами, используя сгенерированное случайным образом число и увеличивая его на 1, а затем на 2 (например, 3, 4, 5).

ВЫЗЫВАЕМ метод *setLocationCells()* из экземпляра SimpleDotCom.

ОБЪЯВЛЯЕМ булеву переменную *isAlive* для хранения состояния игры. **ПРИСВАИВАЕМ** ей значение true.

ПОКА «сайт» не потоплен (isActive == true):

ПОЛУЧАЕМ пользовательский ввод из командной строки.

// ПРОВЕРЯЕМ полученную информацию.

ВЫЗЫВАЕМ метод *checkYourSelf()* из экземпляра SimpleDotCom.

ИНКРЕМЕНТИРУЕМ *переменную* numOfGuesses.

// ПРОВЕРЯЕМ, не потоплен ли «сайт».

ЕСЛИ результат равен Потопил,

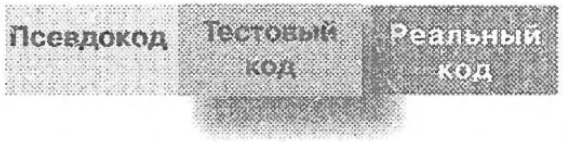
ПРИСВАИВАЕМ переменной *isAlive* значение false (это значит, что мы не хотим снова заходить в цикл).

ВЫВОДИМ количество попыток.

КОНЕЦ ВЕТВЛЕНИЯ

КОНЕЦ ЦИКЛА

КОНЕЦ МЕТОДА



Как и в случае с классом SimpleDotCom, подумайте о фрагментах кода, которые вам хочется (или нужно) улучшить. Отметки предназначены для элементов, на которые нам хотелось бы обратить ваше внимание. Они рассматриваются на следующей странице. Мы пропустили создание тестового класса для игры, потому что он просто здесь не нужен. Класс SimpleDotComGame состоит лишь из одного метода, поэтому нет смысла писать для него проверочный код. Делать отдельный класс, который будет вызывать метод main() из этого класса? Нет необходимости.

ОБЪЯВЛЯЕМ переменную numOfGuesses типа int для хранения количества ходов пользователя; присваиваем ей 0.

СОЗДАЕМ новый экземпляр класс SimpleDotCom.

ВЫЧИСЛЯЕМ случайное число от 0 до 4 для местоположения начальной ячейки.

СОЗДАЕМ целочисленный массив с местоположением трех ячеек и

ВЫЗЫВАЕМ метод setLocationCells() из экземпляра.

ОБЪЯВЛЯЕМ булеву переменную isAlive.

ПОКА «сайт» не потоплен.

ПОЛУЧАЕМ пользовательский ввод.

// ПРОВЕРЯЕМ его.

ВЫЗЫВАЕМ метод checkYourself() из SimpleDotCom.

ИНКРЕМЕНТИРУЕМ переменную numOfGuesses.

ЕСЛИ результат равен «Потопил».

ПРИСВАИВАЕМ переменной isAlive значение false.

ВЫВОДИМ количество попыток.

```
public static void main(String[] args) {
    int numOfGuesses = 0;
    GameHelper helper = new GameHelper();
    SimpleDotCom theDotCom = new SimpleDotCom();
    int randomNum = (int) (Math.random() * 5);
    int[] locations = {randomNum, randomNum+1, randomNum+2};
    theDotCom.setLocationCells(locations);
    boolean isAlive = true;
    while(isAlive == true) {
      String guess = helper.getUserInput("Введите число");
      String result = theDotCom.checkYourself(guess);
      numOfGuesses++;
      if (result.equals("Потопил")) {
        isAlive = false;
        System.out.println("Вам потребовалось" + numOfGuesses + "попыток(и)");
      } // Завершаем if
    } // Завершаем while
} // Завершаем метод main
```

Создаем переменную, чтобы следить за количеством ходов пользователя.

Это специальный класс, который содержит метод для приема пользовательского ввода. Пока сделаем вид, что это часть Java.

Создаем объект «сайт».

Генерируем случайное число для первой ячейки и используем его для формирования массива ячеек.

Передаем «сайту» местоположение его ячеек (массив).

Создаем булеву переменную, чтобы проверять в цикле, не закончилась ли игра.

Получаем строку, вводимую пользователем.

Просим «сайт» проверить полученные данные; сохраняем возвращенный результат в переменную типа String.

Инкрементируем количество попыток.

Потоплен ли «сайт»? Если да, то присваиваем isAlive значение false (так как не хотим продолжать цикл) и выводим на экран количество попыток.

randomO и getUserInputO

Эта страница посвящена методам random() и getUserInput(). Здесь приведен лишь краткий обзор, позволяющий получить общее представление об их работе.
О классе GameHelper вы подробнее прочтете в конце этой главы.

1 Генерируем случайное число.

Это приведение типов, в результате которого последующий элемент меняет свой тип на заданный в скобках. Math.random возвращает double, поэтому необходимо привести его к int (нам нужна последовательность целых чисел между 0 и 4). В процессе отсекается дробная часть double.

Метод Math.random возвращает число от 0 до 1 (не включительно), так что эта формула (включая приведение типов) возвращает число от 0 до 4 (то есть 0–4.999, приведенное к int).

```
int randomNum = (int) (Math.random() * 5)
```

Мы объявляем переменную типа int для хранения случайного числа.

Класс, встроенный в Java.

Метод из класса Math.

2 Получаем пользовательский ввод с помощью класса GameHelper.

Ранее созданный нами экземпляр вспомогательного класса. Он называется GameHelper, и вы пока с ним не знакомы (но еще познакомитесь).

Метод принимает строковый аргумент, который применяется для обращения к пользователю в командной строке. Что бы вы ему ни передали, это будет выведено в терминале до того, как программа начнет принимать пользовательский ввод.

```
String guess = helper.getUserInput("Введите число");
```

Мы объявили строковую переменную для хранения пользовательского ввода, который получаем обратно («3», «5» и т. д.).

Метод класса GameHelper, предлагающий пользователю ввести данные. Он читает их после того, как пользователь нажал кнопку Enter, и возвращает результат в виде строки.

Псевдокод | Тестовый код | Реальный код

Последний класс: GameHelper

Мы создали классы *SimpleDotCom* и *SimpleDotComGame*.

Остался вспомогательный класс, который содержит метод getUserInput(). Код для получения пользовательского ввода включает в себя элементы, которые мы пока не рассматривали. Сейчас вам многое будет непонятно, поэтому оставим его на потом.

Просто перепишите код, приведенный справа, и скомпилируйте его в класс GameHelper. Поместите все три класса (SimpleDotCom, SimpleDotComGame, GameHelper) в один каталог, который будет рабочим.

```
import java.io.*;
public class GameHelper {
   public String getUserInput(String prompt) {
      String inputLine = null;
      System.out.print(prompt + "  ");
      try {
        BufferedReader is = new BufferedReader(
        new InputStreamReader(System.in));
       inputLine = is.readLine();
       if (inputLine.length() == 0 )  return null;
      } catch (IOException e) {
        System.out.println("IOException: " + e);
      }
      return inputLine;
   }
}
```

Сыграем

Вот что произойдет, если мы запустим игру и введем числа 1, 2, 3, 4, 5, 6. Выглядит неплохо.

Полный игровой диалог

Количество попыток может изменяться

```
File Edit Window Help Smile
%java SimpleDotComGame
Введите число  1
Мимо
Введите число  2
Мимо
Введите число  3
Мимо
Введите число  4
Попал
Введите число  5
Попал
Введите число  6
Потопил
Вам потребовалось 6 попыток(и)
```

Что такое? Ошибка?

Только не это!

Вот что будет, если мы введем 1, 1, 1.

Другой пример игрового диалога

Не может быть!

```
File Edit Window Help Faint
%java SimpleDotComGame
Введите число  1
Попал
Введите число  1
Попал
Введите число  1
Потопил
Вам потребовалось 3 попыток(и)
```

Использование библиотеки Java

Вместе с Java поставляются сотни классов . Можете не тратить время на изобретение собственного велосипеда, если знаете, как отыскать нужное в библиотеке Java, называемой еще Java API. Думаем, что у вас найдутся более важные дела. При написании кода сосредоточьтесь на той части, которая уникальна для вашего приложения. Наверняка вы знаете программистов, которые приходят на работу не раньше 10 часов утра и уходят ровно в 5 часов вечера. Они используют Java API.

На следующих страницах вы займетесь тем же. Стандартная библиотека Java представляет собой гигантский набор классов, готовых к применению в качестве «строительных блоков». Они позволят вам создавать приложения преимущественно из готового кода. Java API содержит большое количество кода, который даже набирать не нужно. Все, что от вас требуется, — научиться его использовать.

В кейсе №1 мы оставили небольшую интригу, а именно -

Посмотрите, что произойдет, если мы запустим программу и введем числа 1, 2, 3, 4, 5, 6. Выглядит нормально.

Полная версия игрового диалога
Количество попыток может изменяться

```
File Edit Window Help Smile
%java SimpleDotComGame
Введите число  1
Мимо
Введите число  2
Мимо
Введите число  3
Мимо
Введите число  4
Попал
Введите число  5
Попал
Введите число  6
Потопил
Вам потребовалось 6 попыток(и)
```

Вот что случится, если мы введем 2, 2, 2.

Другой игровой диалог
Не может быть!

```
File Edit Window Help Faint
%java SimpleDotComGame
Введите число  2
Попал
Введите число  2
Попал
Введите число  2
Потопил
Вам потребовалось 3 попыток(и)
```

В текущей версии при попадании вы можете просто повторить два раза удачный ход и потопить «сайт»!

Что же случилось?

```
public String checkYourself(String stringGuess) {

    int guess = Integer.parseInt(stringGuess);
    String result = " Мимо";

    for (int cell : locationCells) {

      if (guess == cell) {
        result = "Попал";
        numOfHits++;
        break;
      } // Конец if
    } // Конец for
```

Преобразуем тип String в int.

Создаем переменную для хранения результата, который будем возвращать. Присваиваем по умолчанию строку «Мимо» (то есть подразумеваем промах).

Повторяем это с каждым элементом массива.

Сравниваем ход пользователя с этим элементом (ячейкой) массива.

Мы попали!

Выходим из цикла. Нет необходимости проверять другие ячейки.

Здесь кроется ошибка. Мы засчитываем попадание каждый раз, когда пользователь угадывает адрес ячейки, *даже если она уже была поражена!*

Нужно научиться узнавать, попадал ли пользователь в конкретную ячейку. При повторном попадании в эту же ячейку не надо засчитывать ход.

```
    if (numOfHits == locationCells.length) {
      result = "Потопил";
    } // Конец if

    System.out.println(result);

    return result;
  } // Конец метода
```

Выходим из цикла, но проверяем, «потоплены» ли мы (три попадания), и изменяем результат на «Потопил».

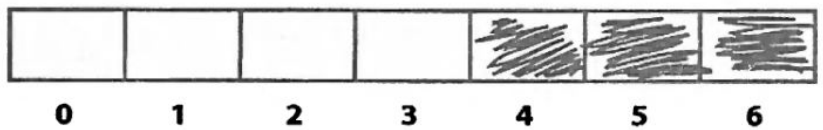
Выводим пользователю результат («Мимо», если он не был изменен на «Попал» или «Потопил»).

Возвращаем результат в вызывающий метод.

Как мы это исправим?

Нужно найти способ узнавать, попал ли игрок в конкретную ячейку. Рассмотрим возможные варианты, но перед этим сформулируем, что уже известно.

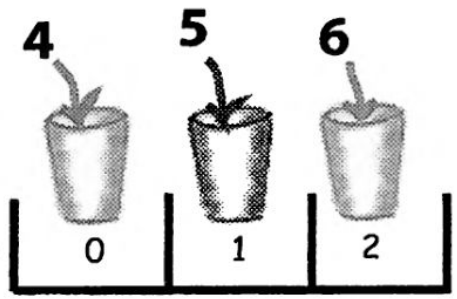
У нас есть виртуальный ряд, состоящий из семи ячеек, три из которых будет занимать объект DotCom. В этом виртуальном ряду показан «сайт», состоящий из ячеек 4, 5 и 6.



Виртуальный ряд с тремя ячейками для объекта DotCom.

Объект DotCom содержит переменную экземпляра — целочисленный массив, хранящий адреса занимаемых ячеек.

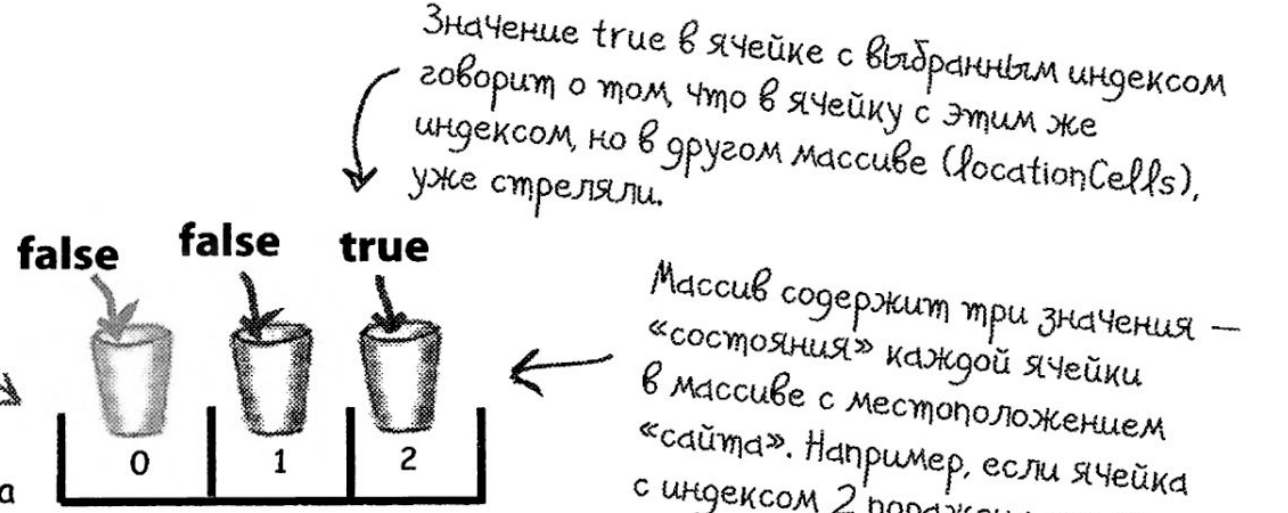
locationCells (переменная экземпляра объекта DotCom).



Переменная-массив, которая хранит местоположение ячеек «сайта». Объект DotCom размещен в ячейках 4, 5 и 6. Это числа, которые пользователь должен угадать.

1 Вариант первый

Можно создать второй массив и при каждом попадании делать в нем соответствующую запись, а затем проверять, стреляли ли в эту ячейку раньше.



Первый вариант слишком усложнен

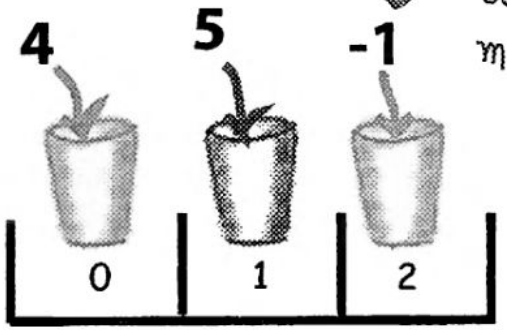
В первом варианте выполняется больше действий, чем можно ожидать. При каждом попадании необходимо изменять состояние второго массива (hitCells), но перед этим нужно проверить его и выяснить, не стреляли ли в эту ячейку раньше. Конечно, это будет работать, но должно существовать более изящное решение.

② Вариант второй

Можно ограничиться оригинальным массивом, изменяя значение пораженной ячейки на -1. При таком подходе мы будем использовать для проверки и изменений один массив.

Значение -1 говорит о том, что ячейка уже была поражена, поэтому в массиве мы ищем только положительные числа или ноль.

locationCells
(переменная экземпляра объекта DotCom)



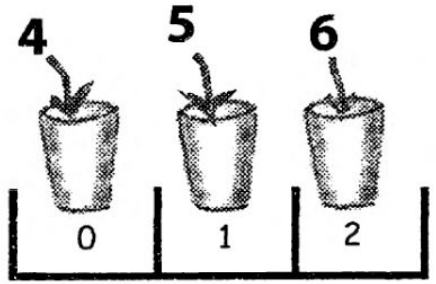
Второй вариант немного лучше, но все еще сложноват

Второй вариант не такой сложный, как первый, но тоже не отличается особой эффективностью. Вам по-прежнему придется перебирать в цикле три элемента массива, даже если в один или два из них уже попали (и они имеют значение -1). Должно существовать лучшее решение...

③ Вариант третий

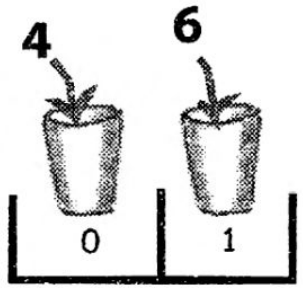
Можно удалить местоположение пораженной ячейки, сделав массив меньше. Однако такие объекты не могут изменять свой размер, поэтому придется создать **новый** массив с меньшей вместимостью и копировать в него оставшиеся ячейки.

Массив locationCells до того, как произошло первое попадание.



Сначала массив состоит из трех элементов. Мы перебираем три ячейки (их позиции в массиве), чтобы проверить, совпадает ли пользовательский ход со значением одной из ячеек (4, 5, 6).

Массив locationCells после того, как ячейка 5 с индексом 1 была поражена.



Когда пользователь попадает в ячейку 5, мы создаем новый массив меньшего размера для хранения оставшихся ячеек и присваиваем ему ссылку на оригинальный массив locationCells.

Третий вариант будет куда лучше, если массив сможет уменьшаться. Нам не придется создавать новый массив меньшего размера, копировать оставшиеся значения и переопределять ссылку.

Оригинальный псевдокод для части метода checkYourself():

ПОВТОРЯЕМ то же самое с каждой ячейкой массива.

// СРАВНИВАЕМ ход пользователя с местоположением клетки.

ЕСЛИ ход пользователя совпал,

ИНКРЕМЕНТИРУЕМ количество попаданий.

// ВЫЯСНЯЕМ, была ли это последняя ячейка.

ЕСЛИ количество попаданий равно 3, **ВОЗВРАЩАЕМ** «Потопил».

ИНАЧЕ потопления не произошло, значит, **ВОЗВРАЩАЕМ** «Попал».

КОНЕЦ ВЕТВЛЕНИЯ

ИНАЧЕ пользователь не попал, значит, **ВОЗВРАЩАЕМ** «Мимо».

КОНЕЦ ВЕТВЛЕНИЯ

КОНЕЦ ПОВТОРЕНИЯ

Жизнь сразу наладится, если мы изменим псевдокод:

ПОВТОРЯЕМ то же самое с ***оставшимися*** ячейками.

// СРАВНИВАЕМ ход пользователя с местоположением клетки.

ЕСЛИ ход пользователя совпал,

УДАЛЯЕМ эту ячейку из массива.

// ВЫЯСНЯЕМ, была ли это последняя ячейка.

ЕСЛИ массив теперь пустой, **ВОЗВРАЩАЕМ** «Потопил».

ИНАЧЕ потопления не произошло, значит, **ВОЗВРАЩАЕМ** «Попал».

КОНЕЦ ВЕТВЛЕНИЯ

ИНАЧЕ пользователь не попал, значит, **ВОЗВРАЩАЕМ** «Мимо».

КОНЕЦ ВЕТВЛЕНИЯ

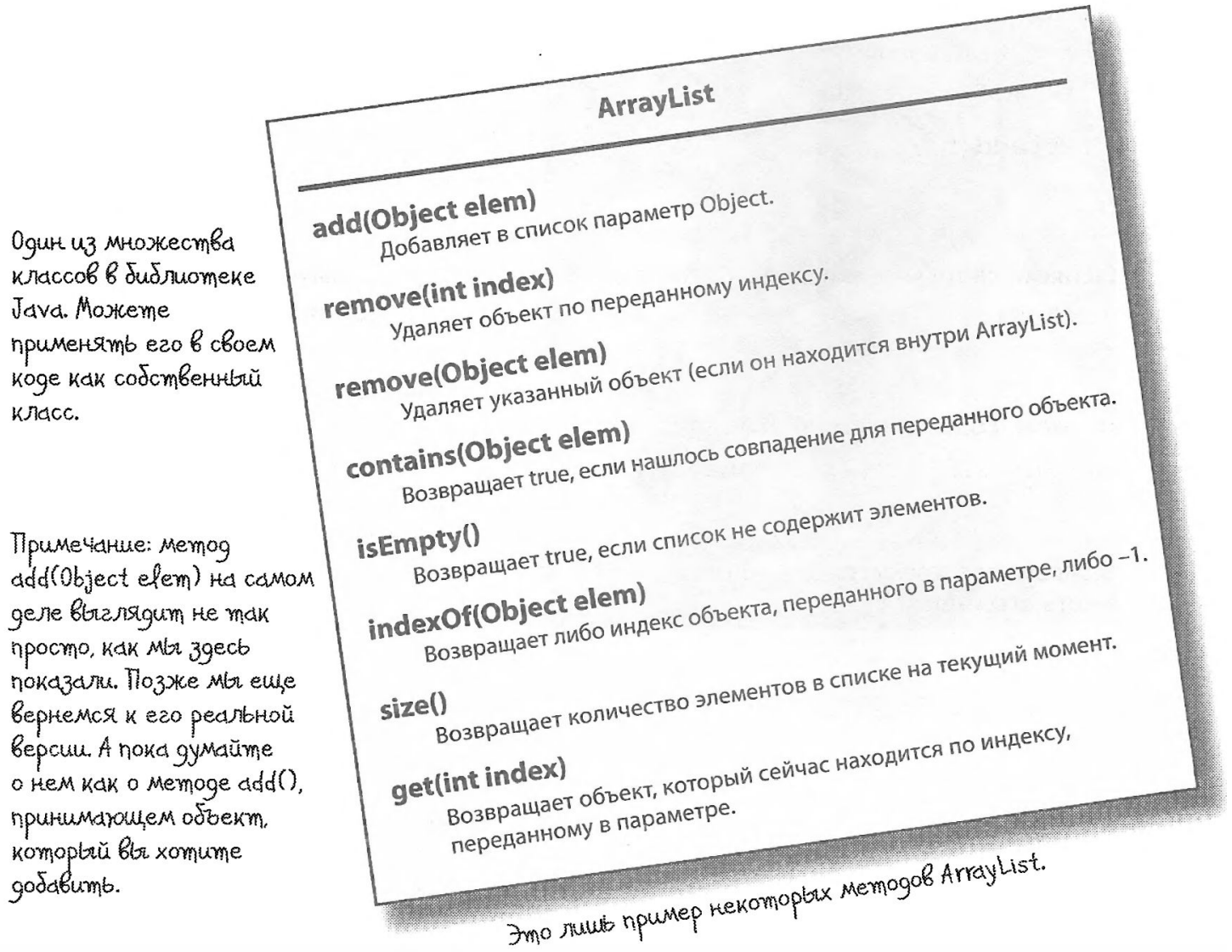
КОНЕЦ ПОВТОРЕНИЯ

Проснитесь и почувствуйте дух библиотеки

Словно по волшебству, такой массив действительно появился.
Но это не совсем массив — это ArtayList.
Это класс из стандартной библиотеки Java (API).

Java Standard Edition (версия Java, с которой вы сейчас работаете; поверьте, вы бы знали, если бы это была Micro Edition, предназначенная для небольших устройств) поставляется с сотнями готовых классов. Это похоже на код, который мы подготовили для вас, где стандартные классы уже скомпилированы.

Это значит, что их не нужно перепечатывать. Просто используйте их.



Несколько примеров использования ArrayList

1. **Создаем один такой:**

```
ArrayList<Egg> myList = new ArrayList<Egg>();
```

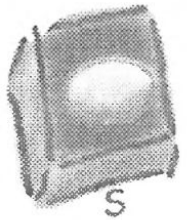
Пока не обращайте внимания на новый синтаксис с угловыми скобками. Это означает «сделай из этого список объектов Egg».

В куче создан новый объект ArrayList. Он маленький, потому что пустой.

2. **Кладем в него что-нибудь:**

```
Egg s = new Egg();

myList.add(s);
```

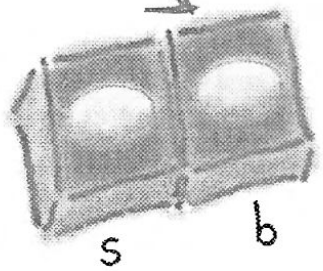


Теперь ArrayList «подрос», чтобы вместить объект Egg.

3. **Кладем в него еще что-либо:**

```
Egg b = new Egg();

myList.add(b);
```



ArrayList опять увеличился, чтобы вместить еще один объект Egg.

4. **Выясняем, сколько элементов в нем хранится:**

```
int theSize = myList.size();
```

ArrayList хранит два объекта, поэтому метод size() возвращает 2.

4. **Выясняем, сколько элементов в нем хранится:**

```
int theSize = myList.size();
```

ArrayList хранит два объекта, поэтому метод size() возвращает 2.

5. **Выясняем, содержит ли он что-либо:**

```
boolean isIn = myList.contains(s);
```

ArrayList действительно содержит объект Egg, на который ссылается s, поэтому contains() возвращает true.

6. **Выясняем, где хранится элемент (то есть его индекс):**

```
int idx = myList.indexOf(b);
```

Индексация элементов в ArrayList начинается с нуля, поэтому indexOf() возвращает 1, хотя объект, на который ссылается b, идет вторым в списке.

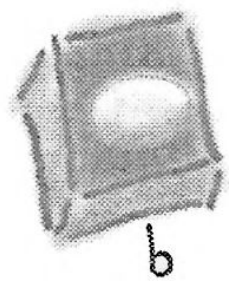
7. **Выясняем, не пустой ли он:**

```
boolean empty = myList.isEmpty();
```

Он точно не пустой, поэтому isEmpty() возвращает false.

8. **Удаляем из него что-нибудь:**

```
myList.remove(s);
```



Эй, смотрите — он уменьшился!

Заполните таблицу, глядя на код слева и подбирая его аналог для обычных массивов. Скорее всего, у вас не получится избежать ошибок, поэтому просто попытайтесь угадать.

Arraylist	Обычный массив
`ArrayList<String> myList = new ArrayList<String>();`	`String [] myList = new String[2];`
`String a = new String("Ура");`	`String a = new String("Ура");`
`myList.add(a);`	
`String b = new String("Лягушка");`	`String b = new String("Лягушка");`
`myList.add(b);`	
`int theSize = myList.size();`	
`Object o = myList.get(1);`	
`myList.remove(1);`	
`boolean isIn = myList.contains(b);`	

1 Обычный массив в момент создания должен знать свой размер.

Используя ArrayList, вы каждый раз просто создаете объект соответствующего типа. Объекту все равно, большим он должен быть или маленьким, потому что он увеличивается и уменьшается по мере добавления и удаления объектов.

```
new String[2]
```

Нужно указать размер.

```
new ArrayList<String>()
```

Нет необходимости задавать размер (хотя при желании вы можете это сделать).

2 При добавлении объекта в обычный массив нужно присвоить ему конкретный индекс.

Индекс должен быть в промежутке от нуля до числа, меньшего, чем длина массива.

```
myList[1] = b;
```

Требует указания индекса.

Если этот индекс выходит за границы массива (например, массив объявлен с размером 2, а теперь вы пытаетесь присвоить что-нибудь индексу 3), то во время выполнения программы будет выдана ошибка.

С ArrayList вы можете указать индекс через метод *add(anInt, anObject)* или просто написать *add(anObject)*, и объект увеличит свой размер, чтобы вместить новый элемент.

```
myList.add(b);
```

Без индекса.

До версии 5.0 в Java не было возможности объявлять *тип* сущностей, которые должны храниться в ArrayList, поэтому с точки зрения компилятора все экземпляры ArrayList представляли собой разнородные коллекции объектов. Но теперь, используя синтаксис <типУказываетсяЗдесь>, можно объявить и создать ArrayList, знающий (и ограничивающий) типы объектов, которые он способен хранить. Мы подробно обсудим параметризованные типы, когда дойдем до коллекций, а пока не забивайте себе голову синтаксисом с угловыми скобками <>, который вам будет встречаться при использовании ArrayList. Просто помните, что этот способ заставляет компилятор допускать для ArrayList только конкретный тип объектов *(в угловых скобках)*.

3 Массивы используют специальный синтаксис, который в Java больше нигде не применяется.

Но в случае с ArrayList мы работаем с обычными Java-объектами, которые не имеют особого синтаксиса.

```
myList[1]
```

Квадратные скобки [] — элемент специального синтаксиса, используемого только для массивов.

4 В Java 5.0 класс ArrayList параметризован.

Мы сказали, что, в отличие от массивов, ArrayList не имеет специального синтаксиса. Но у него *есть* кое-что необычное, появившееся в Java с версией 5.0 Tiger — это ***параметризованные типы***.

```
ArrayList<String>
```

<String> в угловых скобках — это типовой аргумент. ArrayList<String> означает «список объектов String», а ArrayList<Dog> читается как «список объектов Dog».

Псевдокод | Тестовый код | Реальный код

Вспомните, как выглядит версия с ошибкой.

Мы изменили название класса на DotCom (вместо DocComSimple) для новой, улучшенной версии, но это тот же код, который вы могли видеть в предыдущей главе.

```
public class DotCom {

   int[] locationCells;
   int numOfHits = 0;

   public void setLocationCells(int[] locs) {
      locationCells = locs;
   }

   public String checkYourself(String stringGuess) {
      int guess = Integer.parseInt(stringGuess);
      String result = "Мимо";

      for (int cell : locationCells) {
         if (guess == cell) {
            result = "Попал";
            numOfHits++;
            break;
         }
      } // Выходим из цикла
      if (numOfHits == locationCells.length) {
          result = "Потопил";
      }
      System.out.println(result);
         return result;
    } // Конец метода
} // Конец класса
```

Здесь все пошло не так. Мы засчитывали каждый ход как попадание, не проверяя, была ли поражена эта ячейка ранее.

Псевдокод | Тестовый код | Реальный код

Новый и улучшенный класс DotCom

```
import java.util.ArrayList;

public class DotCom {

   private ArrayList<String> locationCells;
   // private int numOfHits;
   // сейчас это нам не нужно

   public void setLocationCells(ArrayList<String> loc) {
      locationCells = loc;
   }


   public String checkYourself(String userInput) {

      String result = "Мимо";

        int index = locationCells.indexOf(userInput);

        if (index >= 0) {

          locationCells.remove(index);


          if (locationCells.isEmpty()) {
            result = "Потопил";
          } else {
            result = "Попал";
          } // Конец if

        } // Конец внешнего if

        return result;
   } // Конец метода
} // Конец класса
```

Пока не обращайте внимания на эту строку; мы поговорим о ней в конце главы.

Изменяем строковый массив на ArrayList, чтобы хранить объекты String.

Новое и усовершенствованное имя аргумента.

Проверяем, содержится ли загаданная пользователем ячейка внутри ArrayList, запрашивая ее индекс. Если ее нет в списке, то indexOf() возвращает -1.

Если индекс больше или равен нулю, то загаданная пользователем ячейка определенно находится в списке, поэтому удаляем ее.

Если список пустой, значит, это было роковое попадание!

Теперь со вкусом ArrayList

НО !!!
Ошибка осталась !!!
НУЖНО РАЗОБРАТЬСЯ !!!!