Конструкторы

Перед тем, как мы перейдем к рассмотрению наследования, я хотел бы коснуться одного вопроса, который на мой взгляд заслуживает внимания. По моему опыту изучения и преподавания программирования могу сказать следующее: многие темы настолько сильно пересекаются друг с другом, что изучение материала часто происходит не линейно, по темам, а по спирали. Сначала мы затрагиваем тему кратко и что-то в ней начинаем понимать. Потом при изучении другой темы мы возвращаемся к уже пройденной, потому что новые знания позволят нам более глубоко рассмотреть то, что кажется уже пройдено.
Наверно именно по этой причине книжки достаточно часто пишут так, что по ним сложно обучаться. Особенно это касается введения в какую-либо область. Ведь хочется сразу написать программу, которая что-то умеет делать. Если подходить академично, линейно, то я вам дожен был рассказать про классы, про статические методы, про типы данных, про параметры и только после этого вы смогли бы понять все слова в важном методе main. Перед автором возникает проблема — хотелось бы вернуться к тому, что пройдено. Но тогда надо держать в голове о чем там говорилось. Или перечитывать снова. На это надо тратить время, а издатель торопит, денежку надо зарабатывать. Хотя на самом деле сразу все описывать совершенно не обязательно. Ведь не всегда нужно знать все слова, чтобы пользоваться фразой на незнакомом языке — можно ее выучить просто так, лишь приблизительно понимая что внутри. Потом, со временем, можно к ней вернуться и понять более глубоко. Поэтому построение материала я решил делать спирально — мы будем регулярно возвращаться к тому, о чем мы уже говорили. Возможно я буду

повторяться, но как говориться «повторение — мать учения». Так что возвращаться мы будем обязательно. И конструкторы не исключение.

Итак, что же такое конструкторы? Некоторое время назад мы касались вопроса создания объектов — через служебное слово new. Вот как это выглядело:

Robot r = new Robot();

За словом new мы видим некоторое подобие вызова метода с названием Robot. Причем без параметров. И этот несколько специфический метод называется **конструктор**. Самое главное — это имя метода-конструктора. Он называется так же, как и сам класс. Возможно это только я придумал такое название — метод-конструктор. В нормальной литературе все просто говорят «конструктор». Обратимся к коду (в котором уже есть конструктор) для разъяснений. Это наш робот, который умеет двигаться вперед и поворачиваться.

```
1  public class Robot
2  {
3      // Текущая координата X
4      private double x = 0;
5      // Текущая координата Y
6      private double y = 0;
7      // Текущий курс (в градусах)
8      private double course = 0;
9
10     // Вот наш конструктор, который пока не очень полезен
11     public Robot() {
12     }
13
14     // Передвижение на дистанцию distance
15     public void forward(int distance) {
16         // Обращение к полю объекта X
17         x = x + distance * Math.cos(course / 180 * Math.PI);
18         // Обращение к полю объекта Y
19         y = y + distance * Math.sin(course / 180 * Math.PI);
```

```
    }

    // Печать координат робота
    public void printCoordinates() {
        System.out.println(x + "," + y);
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getCourse() {
        return course;
    }

    public void setCourse(double course) {
        this.course = course;
    }
}
```

В коде я привел вариант конструктора (который не очень пока нам полезен). Кроме названия Robot (которое обязательно должно совпадать с именем класса) можно обратить внимание, что конструктор НЕ ВОЗВРАЩАЕТ никаких данных. Причем мы не пишем слово void для этого. Отметим и такую особенность конструктора. На самом деле вы можете написать void — но тогда метод перестанет быть конструктором. Никто не запрещает написать int или double. Правда надо будет тогда обязательно сделать return. Можете потом провести эксперименты. Но продолжим наше повествование.
Наш робот уже показал свои возможности, но у него есть большой недостаток — в момент своего создания его начальные координаты всегда равны нулю. Что не может не вызывать неудовольствия. Мы можем конечно сделать методы set (сеттеры) для наших координат X и Y, но это снизит «защищенность» нашего робота. Нужен метод, который бы вызывался один раз при создании объекта и принимал нужные нам параметры X и Y. Так вот конструктор — это именно такой метод. Обратите внимание на один тонкий момент — конструктор (как метод) не создает объект. В момент вызова конструктора объект уже существует. В конструкторе можно присвоить нужные параметры и что еще приятнее — в конструктор можно передать параметры. Т.е. объявить конструктор с параметрами. Вот наш новый вариант класса Robot.

```
public class Robot
{
    // Текущая координата X
    private double x = 0;
    // Текущая координата Y
    private double y = 0;
```

```
    // Текущий курс (в градусах)
    private double course = 0;

    // Теперь конструктор выполняет присваивание. Это удобно.
    public Robot(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // Передвижение на дистанцию distance
    public void forward(int distance) {
        // Обращение к полю объекта X
        x = x + distance * Math.cos(course / 180 * Math.PI);
        // Обращение к полю объекта Y
        y = y + distance * Math.sin(course / 180 * Math.PI);
    }

    // Печать координат робота
    public void printCoordinates() {
        System.out.println(x + "," + y);
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getCourse() {
        return course;
    }

    public void setCourse(double course) {
        this.course = course;
    }
}
```

Но если вы исправите конструктор класса Robot, то наш второй класс для управления роботами RobotManager не будет компилироваться. Мы должны исправить создание робота — использовать новый конструктор с параметрами. Наш RobotManager теперь выглядит несколько иначе.

```
public class RobotManager
{
```

```
3
4       public static void main(String[] args) {
5           // Создаем объекта класса Robot - теперь с параметрами
6           Robot robot = new Robot(20, 20);
7
8           // Вперед на 20 метров
9           robot.forward(20);
10          // Напечатать координаты
11          robot.printCoordinates();
12
13          // Это более корректный способ менять курс.
14          // Реализация внутри робота не сильно отличается, но
15          // мы в любой момент сможем сделать более продвинутую версию
16          // Но класс RobotManager об этом даже не узнает
17          robot.setCourse(90);
18          // Вперед на 20 метров
19          robot.forward(20);
20          // Напечатать координаты
21          robot.printCoordinates();
22
23          // Курс 45 градусов
24          robot.setCourse(45);
25          // Вперед на 20 метров
26          robot.forward(20);
27          // Напечатать координаты
28          robot.printCoordinates();
29      }
30  }
```

Класс RobotManager изменился очень мало — только строка создания робота — там я сделал соответствующий комментарий. Сейчас мы на некоторое время отойдем от конструкторов, но вернемся к ним обязательно. Спираль продолжает раскручиваться 🙂

Наследование

Итак, мы уже можем создавать описания классов. На основании описания классов создавать реальные объекты внутри программы. Причем доступ к полям объектов может быть ограничен, что создает возможность приблизить поведение програмных объектов к поведению реальных. Но давайте посмотрим еще дальше. Классификация всегда подразумеавает древообразную структуру. Наглядным примером может служить биологическая классфикация, предложенная еще в 18 веке Карлом Линнеем.
ООП прекрасно усвоило эту идею. Все представители определенного класса имеют общие признаки и общее поведение. Но должна быть возможность создать класс, который ведет себя немного не так. Т.е. должна быть возможность «переопределить» методы. Это открывает большие возможности. Представим себе, что мы получили от сторонних разработчиков замечательный набор готовых классов для графического интерфейса. В нем есть кнопки, списки, таблицы, изображения. Мы этим пользуемся и в какой-то момент времени нам настоятельно требуется несколько изменить поведение определенного класса. Например надо, чтобы стандартный выпадающий список имел возможность постраничного вывода. В процессе

изучения Java мы познакомимся со стандартной библиотекой графических элементов Swing. Это не означает, что мы будем ее подробно изучать — это просто очень наглядный способ познакомиться с идеями ООП. На примерах мы посмотрим как работает парадигма наследования.

Само по себе наследование реализуется достаточно просто — при указании класса вы указываете дополнительно класс от которого вы хотите унаследоваться. Обратите внимание, что ваш новый класс сразу станет уметь делать то, что есть в классе-родителе (еще используют выражение родительский класс).
Рассмотрим немного искусственный пример — расширим возможности нашего робота. Научим его двигаться назад. Для этого нам потребуется добавить новый метод — back с параметром, сколько метров надо проехать назад. Алгоритм в данном случае простой — нам надо проехать вперед, но со знаком минус.

```
public class Robot
{
    private double x = 0;
    private double y = 0;
    private double course = 0;

    // Передвижение на дистанцию distance
    public void forward(int distance) {
        x = x + distance * Math.cos(course / 180 * Math.PI);
        y = y + distance * Math.sin(course / 180 * Math.PI);
    }

    // Печать координат робота
    public void printCoordinates() {
        System.out.println(x + "," + y);
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getCourse() {
        return course;
    }

    public void setCourse(double course) {
        this.course = course;
    }
}
```

```
// Для того, чтобы унаследоваться нужно просто написать слово
// extends и имя класса, от которого нам надо унаследоваться
public class RobotExt extends Robot
{

    public void back(int distance) {
        forward(-distance);
    }
}
```

Как видим, пока не все так сложно — мы просто написали слово extends (англ. расширять, раздвигать) и далее имя класса Robot. Также в коде видно обращение к методу forward внутри нашего нового методаback. Думаю, что все достаточно очевидно. Но даже в этом коде у нас есть достаточно много важных моментов, на которых мы сосредоточимся.
Во-первых, в коде класса Robot мы не видим наш замечательный конструктор с парамметрами. Во-вторых — нам повезло, что метод back не требует прямого обращения к координатам. Но наверняка такие действия нам могут потребоваться. Например, если мы попробуем написать еще более расширенный класс, который умеет двигаться по кругу. Нам точно потребуется обращаться к координатам, которые закрыты для доступа.
Давайте сначала разберем первый вопрос. Вы можете попробовать просто добавить в описание класса Robot наш конструктор с параметрами. И мы получим ошибку компиляции. Текст ее вот такой:

error: constructor Robot in class Robot cannot be applied to given types;

Проблема в том, что создавать новый класс-наследник можно только либо имея такой же конструктор, что и у родительского класса, либо создать собственный конструктор (возможно с другим набором параметров) — но об этом мы еще поговорим. Пока продолжим — т.к. у нашего родительского класса Robot есть конструктор только с параметрами, то нам нужно создать такой же. Ну что же — сделаем так, как требуется. Теперь наш код будет выглядеть вот так:

```
public class Robot
{

    private double x = 0;
    private double y = 0;
    private double course = 0;

    public Robot(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // Передвижение на дистанцию distance
    public void forward(int distance) {
```

```
        x = x + distance * Math.cos(course / 180 * Math.PI);
        y = y + distance * Math.sin(course / 180 * Math.PI);
    }

    // Печать координат робота
    public void printCoordinates() {
        System.out.println(x + "," + y);
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getCourse() {
        return course;
    }

    public void setCourse(double course) {
        this.course = course;
    }
}
```

```
public class RobotExt extends Robot
{
    // Конструктор с параметрами
    public RobotExt(double x, double y) {
        // вызов конструктора родительского класса
        super(x, y);
    }

    public void back(int distance) {
        forward(-distance);
    }
}
```

Но здесь мы встречаем еще один новый поворот — внутри конструктора класса RobotExt мы видим специальное слово super с параметрами. Это слово означает, что мы вызываем конструктор родительского класса, который принимает два наших параметра. Несколько позже мы познакомимся еще с некоторыми особенностями вызова конструктора родителя.
Теперь снова попробуем внести изменения в наш замечательный класс RobotExt. В нашем классе Robot есть небольшой недостаток — при его

создании мы не задаем ему курс. Т.к. у нас есть метод setCourse, то кажется, что проблема не так уж и велика. Но на самом деле все не так безоблачно. В отличии от изначального класса Robot, который у нас может разворачиваться на месте, может появиться робот, для которого это невозможно. Как обычная машина не может развернуться прямо вокруг своей оси, так и робот «нового поколения» может не уметь вытворять такие фокусы. Значит нам важно иметь возможность устанавливать курс тоже. Давайте сделаем новый конструктор не с двумя, а с тремя параметрами — X, Y и курс.

```
    public RobotExt(double x, double y, double course) {
        super(x, y);
        this.course = course;
    }
```

Упс. Не получается. Компилятор опять ругается. Что мы сделали не так? Да очень просто — мы обратились к закрытому полю. Но у нас же есть прекрасный метод setCourse. Да, в данный момент это выход, можно воспользоваться. Сделаем так:

```
public class RobotExt extends Robot
{
    public RobotExt(double x, double y, double course) {
        super(x, y);
        setCourse(course);
    }

    public void back(int distance) {
        forward(-distance);
    }
}
```

Это решение работает — компилятор не выдает ошибку. Но на самом деле все не так просто, но об этом мы поговорим в другой раз. А пока отметим еще один момент — ключевое слово super ДОЛЖНО СТОЯТЬ ПЕРВОЙ ОПЕРАЦИЕЙ В КОНСТРУКТОРЕ. Вы можете попробовать поменять строки местами — вам будет выдано сообщение об ошибке.

call to super must be first statement in constructor

Так что будьте внимательны.

Но вернемся к ситуации, когда нам надо обратиться к полям родительского класса. Хоть мы и вывернулись, но тем не менее это не всегда возможно. К тому же я обещал вас убедить в том, что такое решение тоже не корректно. Доказательство некорректности мы отложим, а вот попробовать обратиться к закрытому полю нам надо.
Нам действительно не удастся обратиться к полю, которое объявлено как private. Но я предлагаю вам вспомнить, что в разделе об инкапсуляции мы говорили о вариантах доступа, среди которых упоминалось слово protected. Время этого слова пришло. Тогда я написал, что «переменные будут

видны в классах-наследниках, и в классах, находящихся в том же пакете». про пакеты мы пока не говорили, но классы-наследники у нас появились. Так что теперь самое время использовать наши знания. Смотрим код наших классов и обращаем внимание на определение свойства course в классе Robot. Все остальное мы оставили без изменений.

```
public class Robot
{

    private double x = 0;
    private double y = 0;
    // Мы изменили доступ к переменной course
    protected double course = 0;

    public Robot(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // Передвижение на дистанцию distance
    public void forward(int distance) {
        x = x + distance * Math.cos(course / 180 * Math.PI);
        y = y + distance * Math.sin(course / 180 * Math.PI);
    }

    // Печать координат робота
    public void printCoordinates() {
        System.out.println(x + "," + y);
    }

    public double getX() {
        return x;
    }

    public double getY() {
        return y;
    }

    public double getCourse() {
        return course;
    }

    public void setCourse(double course) {
        this.course = course;
    }
}
```

```java
public class RobotExt extends Robot
{
    public RobotExt(double x, double y, double course) {
        super(x, y);
        this.course = course;
    }

    public void back(int distance) {
        forward(-distance);
    }
}
```

Теперь вы можете попробовать обращаться к нашему более расширенному роботу из класса RobotManager. Обратите внимание, что мы можем вызывать методы класса Robot — forward, setCourse для объекта класса RobotExt. Мы **унаследовали** эти методы и можем ими пользоваться.

```java
public class RobotManager
{
    public static void main(String[] args) {
        // Создаем объекта класса RobotExt - X, Y, course
        RobotExt robot = new RobotExt(0, 0, 0);

        // Вперед на 20 метров
        robot.forward(20);
        // Напечатать координаты
        robot.printCoordinates();

        robot.setCourse(90);
        // Вперед на 20 метров
        robot.forward(20);
        // Напечатать координаты
        robot.printCoordinates();

        // Курс 45 градусов
        robot.setCourse(45);
        // Вперед на 20 метров
        robot.forward(20);
        // Напечатать координаты
        robot.printCoordinates();
        // И назад на 10 метров - это метод для RobotExt
        robot.back(10);
        // Напечатать координаты
        robot.printCoordinates();
```

```
28      }
29  }
```

Исходные коды проекта можно взять здесь: Robot2.

И теперь нас ждет следующая статья: Пакеты

34 comments to *Наследование*

- 
  Январь 20, 2015 at 11:24
  Alexey says:

  В классе RobotExt в конструкторе вы устанавливаете значение поля this.course = coures; — и подразумевается что устанавливается поле родителя. Правильно я понимаю что так можно делать если в самом классе RobotExt не определена переменная double coursep; ? Иначе устанавливаться будет уже она а не переменная родителя ?

  Reply

  - 
    Январь 20, 2015 at 12:52
    admin says:

    Да, так и получается. Если вы определяете поле в наследнике, то обращение к этому полю в методах наследника — обращаетесь к нему. Но в методах предка при обращении к этому полю — обращаетесь к полю предка. В общем на мой взгляд получилось не очень хорошо. Я бы сделал так, что если поле предка видимо в потомках, то поле с таким именем определять нельзя. Только если поле приватное — тогда можно.

    Reply

- 
  Февраль 23, 2015 at 16:16
  Kirill says:

Я правильно понял, в одном классе можно задать только один метод-конструктор? И в случае создания класса наследника необходимо в нем тоже запилить конструктор(пусть и с другими параметрами)?

Reply

Февраль 24, 2015 at 06:16
admin says:

Конструкторов может быть сколько угодно — конечно если они с разными параметрами. Если у предка нет конструкторов без параметров, то в наследнике придется реализовать хотя бы один конструктор — с параметрами или без не имеет значения. Но что важно — если у предка нет конструктора без параметров, то внутри конструктора наследника придется вызывать конструктор предка через super. Например, класс B обязан вызывать super с аргументом целым число (в моем случае это super(99)). Иначе не соберется:

```
class A {
public A(int a1) {
}
}

class B extends A {
public B() {
super(99);
}
}
```

Reply

Январь 7, 2017 at 15:58
Виктор says:

На сколько мне не изменяет память из прочитанного в других источниках.
1. Конструктор по умолчанию (без параметров) присутствует в любом классе, компилятор JAVA внедряет его туда по-дефолту.
2. Если добавить конструктор с параметрами, то дефолтный уже вставлен не будет, только ручками
3. При создании наследника, наследник вызывает конструктор родителя по умолчанию (без параметров), который мы просто не видим.

По сему так и вышло:
Чтобы была возможность создать себя classB вынужден вызвать конструктор предка super(99), потому-что при создании public A(int a1); мы добавили новый конструктор с параметром int a1, тем самым отменив дефолтное внедрение компилятором конструктора по умолчанию.

Reply

- 

  Январь 7, 2017 at 16:20
  Виктор says:

  Довольно наглядно будет если проверить его на том же примере:

  ```
  class A {
  // создадим вручную конструктор по-умолчанию/дефолтный
  public A() { System.out.println(«Constructor A») }
  }

  class B extends A {
  public B() { System.out.println(«Constructor B») }
  }

  public class Main {
  public static void main(String[] args) {
  B b = new B();
  }
  }
  ```

  Reply

- 

  Январь 8, 2017 at 01:13
  admin says:

  Ну в общепм да — так оно и есть.

  Reply

- 
  Июль 18, 2015 at 13:51
  shyngys says:

  Почему бы переменной course в классе robot оставить private и в наследнике в конструкторе в super записать в месте с x, y?

  Reply

  - 
    Июль 20, 2015 at 06:19
    admin says:

    Такое тоже возможно. Тут уже на вкус и цвет.

    Reply

- 
  Октябрь 6, 2015 at 23:42
  Сергей Гуренко says:

  Огромное спасибо за уроки. Легко воспринимается.
  Рад, что есть этот сайт

  Reply

- 
  Декабрь 29, 2015 at 17:19
  Виталий says:

  Сайт — бомба! Самый эффективный способ подачи материала сравнительно с прочими! Респект и благодарность автору!

  Reply

- 

  Сентябрь 20, 2016 at 19:49
  Антонио says:

  Признателен также за предоставление столь прекрасной возможности получения знаний!

  Reply

- 

  Октябрь 23, 2016 at 18:29
  Дмитрий says:

  А если наша задача такая:

  Есть абстрактный класс Плагина, который содержит абстрактные методы, которые должен уметь исполнять любой наследник, и некоторые поля, которые должны быть общие для плагина-наследника (ставим их протектед), и поля специфические для этого наследника (ставим прайват). Получается, что общие поля мы засовываем в Абстрактный класс родителя, а спецефические — в наследника.

  Когда нужны общие поля, любой из наследников-плагинов — обращается к полю родителя (при помощи протектед), к своим или напрямую, или через методы.

  Но если например «плагин-форматирования» приводится к «ссылке родителя». Родительская ссылка может обращаться только к своим методам и полям, но ничего не знает о полях и специфических методах наследника. В итоге, когда мы унифицируем все разные плагины, приводя к виду ссылки родителя, мы фактически не можем сделать нормальное управления спецификами дочерних классов.

  Это работает только в виде, когда обхект-наследника, передается в реализацию абстрактного метода родителя (использовать специфику), и в этот абстрактный метод передается Object, который в классе наследника преобразовывается опять конкретно к классу наследника и может использовать его поля и методы, а также обращаться к методам родителя протектед.

  То есть шифратор — дешифратор по сути, когда общая программа содержит абстрактную коллекцию плагинов-наследников, но преобразовыввает через прописанную реализацию к конкретному наследнику, его же наследуемым методом.

  Я сильно усложняю? Или как то можно проще сохранять список разных реализаций плагина в основной программе, и работать с их спепецификой, преобразовывая через абстрактный метод дешифровки родителя, в объекте наследнике?

  * З.ы. Я не курил

Reply

Ноябрь 9, 2016 at 00:32
admin says:

Абстрактные классы должны создаваться с учетом расширения в наследниках — если требуются какие-то привязки к реализации, то это уже плохое проектирование.
Что касается плагинв, то чаще их реализуют через интерфейсы.

Reply

Октябрь 25, 2016 at 17:32
AlexTes says:

Большое спасибо за материал и его замечательную подачу!

Reply

Ноябрь 15, 2016 at 09:05
v says:

Тоже внес вклад в заполнении квадратной спирали, выложу основной код, заполняет, как думает человек, т.е. сначала верхнюю сторону, потом правую, потом нижнюю и левую.

```
        for (int k = 0; k < loop; k++) {     // k=число квадратных кругов
            for (int i = 0; i < size - 1; i++) {
                for (int j = 0; j  0) {
                            if (arr[x0 + k][y0 + k] == 0) {
                                arr[x0 + k][y0 + k] = arr[x0 + k - 1][y0 + k] + 1;  //расчет новой линии
                            }
                        }
```

```
                    if (arr[x0 + xV1 + k][y0 + k] == 0) {
                        arr[x0 + xV1 + k][y0 + k] = arr[xV1 - 1 + k][y0 + k] + 1;
                        xV1++;
                    }
                }
            }
        }
        for (int i = 0; i < size - 1; i++) {
            for (int j = 0; j < size - 1; j++) {
//              правая сторона
                if (yV1 == size) {
                    break;
                } else {
                    if (arr[side - k][y0 + yV1 + k] == 0) {
                        arr[side - k][y0 + yV1 + k] = arr[side - k][yV1 - 1 + k] + 1;
                        yV1++;
                    }
                }
            }
        }
        for (int i = 0; i < size - 1; i++) {
            for (int j = 0; j < size - 1; j++) {
//              нижняя сторона
                if (xV2 == size) {
                    break;
                } else {
                    if (arr[x0 + k][side - k] == 0) {
                        arr[x0 + k][side - k] = arr[side - k][side - k] + side - k * 2;  //расчет новой линии
                    }
                    if (arr[x0 + xV2 + k][side - k] == 0) {
                        arr[x0 + xV2 + k][side - k] = arr[xV2 - 1 + k][side - k] - 1;
                        xV2++;
                    }
                }
            }
        }
        for (int i = 0; i < size - 1; i++) {
            for (int j = 0; j  loopGate) {
                    if (arr[x0 + k][y0 + yV2 + 1 + k] == 0) {
                        arr[x0 + k][y0 + yV2 + 1 + k] = arr[x0 + k][y0 + yV2 + k] - 1;
                        yV2++;
                    }
                }
            }
        }
        xV1 = yV1 = xV2 = yV2 = 1; //сброс значений
```

```
55          }
```

Сделал тестовый образец, сравнил с кодом Grif’a на заполнении спиралью массивов от 0x0 до 200x200 на тачке E5200 код Grif’a быстрее на 3-4 секунды (27сек. и 24сек.), скорее всего за счет минимального использования счетчиков.
В качестве ввода сторон можно использовать простой способ ввода в консоли – сканер

```
1 public class ArraySpiralScannerMng {
2     public static void main(String[] args) {
3         Scanner sc = new Scanner(System.in);
4         ArraySpiralScanner ass=new ArraySpiralScanner();
5         while (sc.hasNextInt()) {
6             ass.ArraySpiral(sc.nextInt());
7         }
8     }
9 }
```

Reply

- 
  Декабрь 4, 2016 at 21:05
  ЮРА says:

  Если мы устанавливаем RobotExt управляет RobotManadger то какой класс управляет Robot?

  Reply

  - 
    Декабрь 5, 2016 at 09:56
    admin says:

    Судя по вопросу, Вы не увидели идеи ООП. RobotManager — это класс, который мы сами учим управлять обхектом типа Robot или RobotExt — зависит от того, что мы хотим. Это мы сами решаем.

    Reply

- 

  Декабрь 5, 2016 at 11:48
  JAVA says:

  Две проблемы:
  1)Компилятор выдает ошибку:нельзя устанавливать параметры в значение Robot robot = new Robot( ) ;
  2)В конструкторе super(x ,y) пишется что фактическое и формальное значения имеют разную длину.
  Как исправить?

  Reply

  - 

    Декабрь 5, 2016 at 12:06
    admin says:

    Я не экстрасенс 🙂 Без конкретного кода ответить на такого рода вопросы я не могу. Если это проблема кода, указанного в статье, то хорошо бы уточнить, где именно такое происходит.

    Reply

- 

  Декабрь 5, 2016 at 21:09
  JAVA says:

  public class Robot {
  double x =10;
  double y =10;
  protected double course=0;

  public Robot() {
  this.x= x;
  this.y= y;
  }
  public void forward(int distance) {

```
x = x + distance * Math.cos(course / 180 * Math.PI);

y = y + distance * Math.sin(course / 180 * Math.PI);
}

public void printCoordinates() {
System.out.println(x + «,» + y);
}

public double getX() {
return x;
}

public double getY() {
return y;
}

public double getCourse() {
return course;
}
public void setCourse(double course) {
this.course = course;
}

public class RobotExt extends Robot{
public RobotExt(double x, double y, double course) {
//Пишется actual and formal lists differ in length
super(x, y);
this.course = course;
}

public void back(int distance) {
forward(-distance);
}
}
}
```

Reply

Декабрь 6, 2016 at 12:03
admin says:

— Ошибка раз:
public Robot() {
this.x= x;
this.y= y;
}

Почему в конструкторе без параметров такое появилось ? Это не будет выдавать ошибку при компиляции, но это бессмысленное выражение. Догадывайтесь сами.

— Ошибка два
public RobotExt(double x, double y, double course) {
//Пишется actual and formal lists differ in length
super(x, y);
this.course = course;
}

Так Вы вызываете конструктор у класса Robot, которого нет. Он же у вас с параметрами, а на самом деле такого нет — см. ошибку раз.

Reply

Декабрь 5, 2016 at 21:11
JAVA says:

Возможно я не так понял куда нужно вводить класс RobotExt ?

Reply

Декабрь 6, 2016 at 12:04

admin says:

См. комментарий к первому сообщению. Вам надо более глубоко покопаться в конструкторах.

Reply

- 
Декабрь 6, 2016 at 22:14
JAVA says:

Спасибо большое!Все получилось.

Reply

- 
Февраль 14, 2017 at 19:29
ooooo says:

Можете объяснить чем собственно отличается конструктор от метода set если смысл тот же-замена данных при вызове метода?

Reply

  - 
Февраль 15, 2017 at 11:24
admin says:

Конструктор ОБЯЗЫВАЕТ указывать эти параметры. В то время как сеттер это не требует. Вот собственно и вся разница.

Reply

- 
Январь 17, 2018 at 16:50
Вячеслав says:

Возможно вместо Robot имелось ввиду RobotExt во фразе «Вы можете попробовать просто добавить в описание класса Robot наш конструктор с параметрами.»?

Reply

- 

  Январь 17, 2018 at 17:10
  admin says:

  По идее там все правильно, но что-то мне не нравится — надо будет переделать текст. Какой-то он не совсем понятный и «чистый» получился.

  Reply

- 

  Июнь 1, 2018 at 15:04
  Galina says:

  Спасибо за такой понятный язык.
  Без super компилятор предлагает создать методы setCourse(); и forward() внутри класса RobotExt

```
public class RobotExt extends Robot
{
public RobotExt(double x, double y, double course) {
super(x, y);
super.setCourse(course);
}

public void back(int distance) {

super.forward(-distance);
}
}
```

  Reply

Июль 19, 2018 at 15:29
v says:

Второй раз просматриваю этот курс и (по мере прокачивания скила□) появились замечание и пожелание.
1. Не лучше ли использовать более компактный вид метода forward?

```
void forward(int distance) {
    x += distance * Math.cos(course / 180 * Math.PI);
    y += distance * Math.sin(course / 180 * Math.PI);
}
```

2. Помимо методов робота forward и back можно сделать методы left и right, которые держат автокурс на -90 и +90 градусов (по аналогии игры «Might and magic legacy x» — там можно поворачиваться только на 90 градусов). Более точный курс указывается методом setCourse, например:

```
void left() {
    setCourse(getCourse() - 90);     //поворот на -90 градусов
}
void right() {
    setCourse(getCourse() + 90);     //поворот на 90 градусов
}
```

Reply

Июль 21, 2018 at 05:01
admin says:

1. Результат в итоге получится такой же — просто мне так показалось нагляднее. Хотя может быть и такой вариант
2. Можно наверно. Но надо ли ? Есть понятный метод — поворот на определенное количество градусов. Хотя опять же — зависит от потребностей. Если часто используется поворот на 90 градусов налево-направо, то вполне может быть.

Программирование, по моему мнению, исключительно инженерная специальность. Т.е. тут может быть несколько решений и они всегда могут быть исправлены.

Reply

- Август 28, 2018 at 16:24
  andy says:

  «Обратите внимание на один тонкий момент — конструктор (как метод) не создает объект. В момент вызова конструктора объект уже существует.» — тут вы говорите что-то невразумительное. Если конструктор не создает объект, то что его создает? Если в момент вызова конструктора объект уже есть, то откуда он взялся?

  Reply

  

  - Август 31, 2018 at 05:54
    admin says:

    Объект создается в области памяти, которая называется «куча». Создает его там сама JVM. Потом идет инициализация полей значениями по умолчанию или, если есть присваивание, то делает их и только потом вызывает конструктор. Если хотите знать точно — почитайте спецификацию языка.

    Reply

Leave a reply

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя *

E-mail *

Сайт

2 + 9 =

Add comment