package com.javarush.task.task23.task2301;  
  
/\*   
Запрети наследование  
Запрети наследование от класса Listener.  
\*/  
public class Solution {  
  
 public static class Listener {  
 public void onMouseDown(int x, int y) {  
 //do something on mouse down event  
 }  
  
 public void onMouseUp(int x, int y) {  
 //do something on mouse up event  
 }  
 }  
  
 public static void main(String[] args) {  
  
 }  
}

package com.javarush.task.task23.task2302;  
  
/\*   
Запрети переопределение  
Запрети переопределение метода onMouseDown.  
\*/  
public class Solution {  
  
 public static class Listener {  
 public void onMouseDown(int x, int y) {  
 //do something on mouse down event  
 }  
  
 public void onMouseUp(int x, int y) {  
 //do something on mouse up event  
 }  
 }  
  
 public static void main(String[] args) {  
  
 }  
}

package com.javarush.task.task23.task2303;  
  
/\*   
Запрети создание экземпляров класса  
Запрети создание экземпляров класса Listener.  
\*/  
public class Solution {  
  
 public static class Listener {  
 public void onMouseDown(int x, int y) {  
 //do something on mouse down event  
 }  
  
 public void onMouseUp(int x, int y) {  
 //do something on mouse up event  
 }  
 }  
  
 public static void main(String[] args) {  
  
 }  
}

package com.javarush.task.task23.task2304;  
  
/\*   
Inner 3  
Внутри класса Solution:  
1) реализуйте private class TaskDataProvider используя Task и DbMock, цель которого — обновить поле tasks.  
2) реализуйте private class NameDataProvider используя String и DbMock, цель которого — обновить поле names.  
\*/  
public class Solution {  
  
 private List<Task> tasks;  
 private List<String> names;  
  
 private DbDataProvider taskDataProvider = new TaskDataProvider();  
 private DbDataProvider nameDataProvider = new NameDataProvider();  
  
 public void refresh() {  
 Map taskCriteria = ViewMock.*getFakeTasksCriteria*();  
 taskDataProvider.refreshAllData(taskCriteria);  
  
 Map nameCriteria = ViewMock.*getFakeNamesCriteria*();  
 nameDataProvider.refreshAllData(nameCriteria);  
 }  
  
 private interface DbDataProvider<T> {  
 void refreshAllData(Map criteria);  
 }  
  
 class Task {  
 }  
  
 public static void main(String[] args) {  
  
 }  
}

package com.javarush.task.task23.task2305;  
  
/\*   
Inner  
Реализовать метод getTwoSolutions, который должен возвращать массив из 2-х экземпляров класса Solution.  
Для каждого экземпляра класса Solution инициализировать поле innerClasses двумя значениями.  
Инициализация всех данных должна происходить только в методе getTwoSolutions.  
\*/  
public class Solution {  
 public InnerClass[] innerClasses = new InnerClass[2];  
  
 public class InnerClass {  
 }  
  
 public static Solution[] getTwoSolutions() {  
  
 return null;  
 }  
  
 public static void main(String[] args) {  
  
 }  
}

package com.javarush.task.task23.task2306;  
  
/\*   
Inner 2  
В классе SuperUser метод getDescription должен учитывать страну и город, т.е. возвращать результат аналогичный следующему:  
My name is George. I'm from the USA, Seattle.  
  
Используй возможности иннер класса.  
\*/  
public class Solution {  
 private String country;  
 private String city;  
  
 public Solution(String country, String city) {  
 this.country = country;  
 this.city = city;  
 }  
  
 /\*т.к. модификатор иннер класса private, то чтобы вызвать метод getDescription из другого внешнего класса,  
 нужно обернуть его вызов в какой-то public метод\*/  
 public String getDescriptionOfUser(String name) {  
 return getTrickyUser(name).getDescription();  
 }  
  
 public SuperUser getTrickyUser(String name) {  
 return new SuperUser(name);  
 }  
  
 private class SuperUser {  
 private String name;  
  
 //доступ к этому методу возможен только внутри класса Solution, т.к. модификатор иннер класса private  
 public SuperUser(String name) {  
 this.name = name;  
 }  
  
 //доступ к этому методу возможен только внутри класса Solution, т.к. модификатор иннер класса private  
 public String getDescription() {  
 return String.*format*("My name is %s.", this.name);  
 }  
 }  
  
 public static void main(String[] args) {  
 Solution solution = new Solution("the USA", "Seattle");  
 //внутри класса Solution (а сейчас мы внутри) к методу getDescription можно обращаться обоими способами  
 System.*out*.println(solution.getTrickyUser("George").getDescription());  
 //а из любого другого внешнего класса только так:  
 System.*out*.println(solution.getDescriptionOfUser("George"));  
 }  
}

package com.javarush.task.task23.task2307;  
  
/\*   
Как выбрать нужное?  
В методе main присвой объекту Object obj экземпляр класса TEST.  
Не изменяй ничего кроме метода main.  
\*/  
public class Solution {  
 public static final String *TEST* = "test";  
  
 public static class TEST {  
 @Override  
 public String toString() {  
 return "test class";  
 }  
 }  
  
 static Object *obj*;  
  
 public static void main(String[] args) {  
 *obj* = Solution.*TEST*;  
 System.*out*.println(*obj*);  
 }  
}

package com.javarush.task.task23.task2308;  
  
/\*   
Рефакторинг, вложенные классы  
Отрефакторите класс Solution: вынесите все константы в public вложенный(nested) класс Constants.  
Запретите наследоваться от Constants.  
\*/  
public class Solution {  
 public class ServerNotAccessibleException extends Exception {  
 public ServerNotAccessibleException() {  
 super("Server is not accessible for now.");  
 }  
  
 public ServerNotAccessibleException(Throwable cause) {  
 super("Server is not accessible for now.", cause);  
 }  
 }  
  
 public class UnauthorizedUserException extends Exception {  
 public UnauthorizedUserException() {  
 super("User is not authorized.");  
 }  
  
 public UnauthorizedUserException(Throwable cause) {  
 super("User is not authorized.", cause);  
 }  
 }  
  
 public class BannedUserException extends Exception {  
 public BannedUserException() {  
 super("User is banned.");  
 }  
  
 public BannedUserException(Throwable cause) {  
 super("User is banned.", cause);  
 }  
 }  
  
 public class RestrictionException extends Exception {  
 public RestrictionException() {  
 super("Access is denied.");  
 }  
  
 public RestrictionException(Throwable cause) {  
 super("Access is denied.", cause);  
 }  
 }  
  
 public static void main(String[] args) {  
  
 }  
}

package com.javarush.task.task23.task2309;  
  
import com.javarush.task.task23.task2309.vo.\*;  
  
import java.util.List;  
  
/\*   
  
Анонимность иногда так приятна!  
1. В пакете vo создайте public классы User, Location, Server, Subject, Subscription, которые наследуются от NamedItem  
2. В классе Solution для каждого класса создайте свой метод, который возвращает список экземпляров класса.  
  
Например, для класса User это будет — public List<User> getUsers()  
Для класса Location это будет — public List<Location> getLocations()  
  
3. Внутри каждого такого метода создайте анонимный класс от AbstractDbSelectExecutor и вызовите его нужный метод.  
Подсказка:  
тело метода должно начинаться так: return new AbstractDbSelectExecutor  
  
4. Пример вывода для User и Location:  
Id=5, name='User-5', description=Got by executing 'SELECT \* FROM USER'  
Id=1, name='Location-1', description=Got by executing 'SELECT \* FROM LOCATION'  
  
5. Проанализируйте пример вывода и сформируйте правильный query для всех классов.  
\*/  
public class Solution {  
 public static void main(String[] args) {  
 Solution solution = new Solution();  
 *print*(solution.getUsers());  
 *print*(solution.getLocations());  
 }  
  
 public static void print(List list) {  
 String format = "Id=%d, name='%s', description=%s";  
 for (Object obj : list) {  
 NamedItem item = (NamedItem) obj;  
 System.*out*.println(String.*format*(format, item.getId(), item.getName(), item.getDescription()));  
 }  
 }  
}

package com.javarush.task.task23.task2310;  
  
/\*   
Напряги извилины!  
Метод printName должен выводить свое собственное имя, т.е. «sout»  
Сделайте минимум изменений.  
\*/  
public class Solution {  
 private String name;  
  
 Solution(String name) {  
 this.name = name;  
 }  
  
 private String getName() {  
 return name;  
 }  
  
 private void sout() {  
 new Solution("sout") {  
 void printName() {  
 System.*out*.println(getName());  
 }  
 }.printName();  
 }  
  
 public static void main(String[] args) {  
 new Solution("main").sout();  
 }  
}

package com.javarush.task.task23.task2311;  
  
/\*   
Повторяем threads  
Сделать так, чтобы в методе someActions вызывались только методы класса Solution.  
  
Ожидаемый вывод в методе main:  
Amigo: Mmmmm, beef  
Amigo: knock knock  
Amigo: Zzzzzzz...1 sec  
\*/  
public class Solution {  
 public final String name;  
 public final String food;  
 public final String sound;  
  
 public Solution(String name, String food, String sound) {  
 this.name = name;  
 this.food = food;  
 this.sound = sound;  
 }  
  
 public void eat() {  
 System.*out*.println(name + ": Mmmmm, " + food);  
 }  
  
 public void play() {  
 System.*out*.println(name + ": " + sound + " " + sound);  
 }  
  
 public void sleep(long milliseconds) {  
 System.*out*.println(name + ": Zzzzzzz..." + (milliseconds / 1000) + " sec");  
 }  
  
 public void live() throws InterruptedException {  
 Thread thread = new Thread() {  
 public void run() {  
 try {  
 someActions();  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
 }  
  
 private void someActions() throws InterruptedException {  
 eat();  
 play();  
 *sleep*(1000);  
 }  
 };  
 thread.start();  
 thread.join();  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 new Solution("Amigo", "beef", "knock").live();  
 }  
}

Арканоид(1)

Давай сегодня напишем игру **Арканоид**.

В оригинале это выглядит примерно так *http://www.youtube.com/watch?v=Th-Z6QQ5AOQ*  
Для начала мы напишем **упрощенную** **версию** этой игры.  
У нас будут кубики, шарик и летающая «***подставка***«, которая не дает шарику упасть.

Поэтому нам понадобятся классы:  
а) **Arkanoid** — класс в котором происходят основные действия. Также в нем будет наша точка входа в программу, метод main.  
б) **Ball** — **шарик**.  
в) **Brick** — «***кирпичи***«, которые мы сбиваем шариком.  
г) **Stand** — **летающая** **подставка**.

Создай их, и не забудь о методе public static void main (String[] **args**) в класе **Arkanoid**.

### Арканоид(2)

Главному **классу** (**Arkanoid**) нужно будет хранить информацию о размерах поля, где будут происходить все действия.

Поэтому:  
а) добавь в класс **Arkanoid** два приватных поля width(**ширина**) и height(**высота**) типа int  
б) создай для них публичные **геттеры** и **сеттеры**

### Арканоид(3)

Пожалуй, пора добавить конструктор в класс **Arkanoid**.  
Конструктор должен принимать два параметра типа int (width и height) и корректно устанавливать значения соответствующих полей класса.

### Арканоид(4)

Также классу **Arkanoid** нужно будет хранить ссылку на шарик (**Ball**), «***подставку***» **Stand** и список «***кирпичей***«.

**Надо:**  
а) создать в классе **Arkanoid** два private поля: ball типа **Ball** и stand типа **Stand**  
б) добавить private поле bricks типа **List**<Brick>  
в) добавь для них публичные **геттеры** и **сеттеры**

### Арканоид(5)

Чего еще не хватает классу **Arkanoid**?  
Во-первых ему нужен метод run(), в котором будет описана основная логика программы.  
Еще нужен метод move() — который будет двигать на один шаг все объекты требующие движения.  
Создай методы run() и move().

Еще нам понадобится **НЕ** приватная статическая переменная game типа **Arkanoid**, которая будет хранить ссылку на созданный экземпляр класса **Arkanoid**.

**Выглядеть это должно примерно так:**  
static Arkanoid game;

### Арканоид(6)

У нас будут «***кирпичи***«, «***шарик***» и «***подставка***» и у них будет много общего.  
Они будут перемещаться по полю и отрисовываться.  
Значит у них у всех будут координаты и размер.  
А еще методы move() — для перемещения и draw() для отрисовки.

Есть интересное предложение: давай введем один базовый класс для все объектов.  
Пусть это будет класс **BaseObject**.  
А классы **Ball**, **Stand**, **Brick** от него наследуются.

Создай класс **BaseObject** и добавь его родителем к классам **Ball**, **Stand**, **Brick**

Еще нам понадобится класс **Canvas**  
Он будет ответственным за «***отрисовку***» объектов.  
С помощью его они будут отрисовывать себя.  
Вернее даже на нем, но детали я сообщу позднее.

**Создай и этот класс.**

### Арканоид(7)

Теперь перейдем к классу **BaseObject**.  
Я хочу сделать несколько предложений.

**Во-первых** для простоты считать все объекты у нас будут круглыми.  
Нет, отрисовывать их мы будем фигурными, как и раньше.  
А вот при расчетах из взаимодействия исходить из того, что они круглые.  
Так — гораздо проще.

**Во-вторых**. Пусть координаты объектов и радиус будут вещественными числами.  
Это придаст плавность движениям и точность всем вычислениям.  
А при отрисовке мы будем их округлять.

**Итак:**  
а) Добавь в класс **BaseObject** приватные поля x (double), y (double), radius (double).  
б) Добавь **геттеры** и **сеттеры**.  
в) Добавь конструктор BaseObject(double **x**, double **y**, double **radius**).  
г) Пройдись по все **классам-наследникам** и поправь у них конструкторы.

Если вы пользуйтесь **Intellij IDEA** — Alt+Insert вам в помощь.

### Арканоид(8)

Но и это еще не все.  
Классу **BaseObject** нужны еще методы.  
Пока это будут пустые методы draw(Canvas canvas) и move().  
Классы-наследники должны будут переопределить их у себя и реализовать необходимую функциональность (пока можешь оставить их пустыми, просто чтобы код компилировался).  
Объяви эти методы, но сделай их абстрактными.  
Также сделай абстрактным сам класс **BaseObject**.

А еще нам нужно будет определять попал шарик в кирпич или в подставку.  
**Это будем делать так:**  
В этом же классе, создадим специальный метод: boolean isIntersec(BaseObject **o**)  
Он будет определять — «***пересеклись***» объекты или нет. Если пересеклись — возвращать **true**, если нет — **false**.

Т.к. объекты мы условно считаем кругами, то предлагаю такую формулу взаимодействия  
Если центр круга одного объекта попал в круг другого, то будем считать, что они столкнулись.

(this.x-o.getX()<=this.radius+o.getRadius()&&this.y-o.getY()<=this.radius+o.getRadius()){  
 return true;

**Или еще проще:**  
дистанция\_между\_объектами <= max (радиус\_первого\_объекта, радиус\_второго\_объекта)

### Арканоид(9)

Теперь займемся классом **Canvas**.  
Он у нас будет содержать матрицу (***двумерный массив***), куда мы будем «***рисовать***«.  
У матрицы есть ширина и высота.  
А еще будем в ней хранить не **числа** (int), а **символы** (char).

**Надо:**  
а) Добавить в класс два приватных поля width и height.  
б) Добавить в класс приватное поле matrix (char[][]).  
в) Добавить конструктор с двумя параметрами типа int (width и height), поле matrix должно быть инициализировано пустым массивом размерностью **[**height**+2][**width**+2]**.  
г) Добавить геттеры и сеттеры для всех приватных полей класса.

### Арканоид(10)

Что мы будем делать с **Canvas**?  
Мы будем рисовать на нем (в его матрице).  
Поэтому нам понадобятся два метода  
void setPoint(double **x**, double **y**, char **c**)  
void drawMatrix(double **x**, double **y**, int[][] **matrix**, char **c**)

Первый метод — setPoint будет «**ставить точку в координатах** x**,**y**цветом**c«.

**В методе надо:**  
а) округлить x и y до целых чисел  
б) занести в matrix[y][x] значение с  
в) ничего не делать, если x < **0** или y < **0** или y > **matrix.length** или x > **matrix[0].length**

Второй метод — drawMatrix копирует переданную ему картинку (***матрицу***) в матрицу **Canvas**.  
И не просто копирует, а начиная с координат x, y

**В методе надо:**  
а) с помощью двух вложенных циклов пройтись по всем ячейкам переданной картинки  
б) если значение ячейки элемента **[ I ][ j ]**  полученной матрицы не равно**0**, то покрасить в матрице объекта **Canvas**точку **(**x**+j,**y**+i)** в цвет c:  
setPoint(x+j, y+i, c)

### Арканоид(11)

Еще **Canvas** понадобится два метода, напиши их.  
а) метод clear()  
Этот метод будет очищать матрицу, чтобы на ней снова можно было рисовать.  
Можешь просто заново инициализировать matrix пустым массивом правильной размерности.  
б) метод print()  
Этот метод рисует матрицу на экран.  
Тут уже ты должен сам разобраться: вывести набор символов не так уж и сложно.

### Арканоид(12)

Теперь напишем класс **Brick**.

Во-первых займемся конструктором и в нем точно зададим «***радиус***» всех кирпичей.  
**Пусть он выглядит так:**  
public Brick(double x, double y) {  
  super(x, y, 3);  
}

Еще нам нужно переопределить два метода move() и draw(Canvas **canvas**).  
Метод move() не делает ничего (не содержит кода), т.к. кирпич никуда не двигается.

Напиши еще метод draw(Canvas **canvas**).  
Его кодом я займусь сам.

### Арканоид(13)

Класс **Ball** уже посложнее — шарик же двигается.

Нам понадобятся приватные поля:  
а) speed (скорость шарика) типа double;  
б) direction (направление движения в градусах: от**0**до**360**) типа double;  
в) dx (расстояние по x, которое проходит шарик за один шаг. вычисляется на основеspeedиdirection) типа double;  
г) dy (расстояние по y, которое проходит шарик за один шаг. вычисляется на основе speed и direction) типа double;  
д) isFrozen («**истина**» если шарик «***заморожен***» — не двигается) типа boolean.

Не забудь добавить геттеры для всех полей этого класса, кроме isFrozen (его мы будем использовать только внутри класса **Ball**).

**А еще сделай-ка конструктор:**  
а) параметры — x, y, speed, direction;  
б) радиус (для вызова родительского класса) всегда равен **1**;  
в) не забудь установить isFrozen в **true** — в начале игры шарик никуда не летит.

### Арканоид(14)

Чего-то не хватает в классе **Ball**:

Во-первых надо реализовать метод move(), унаследованный от **BaseObject**:  
а) x должен увеличиваться на dx каждый ход  
б) y должен увеличиваться на dy каждый ход  
если шарик «***заморожен***«, то x и y меняться не должны

Во-вторых надо реализовать метод draw(Canvas **canvas**):  
на объекте canvas необходимо вызвать метод setPoint с параметрами (x, y, ‘**O**‘)

В-третьих надо реализовать метод start()  
Именно его вызов «***размораживает***» шарик.  
Что для этого надо сделать — это ты уже сам реши.

**P.S.** Чуть не забыл! Поляx, yиradiusобъявлены в классе**BaseObject**с модификатором доступа private,  
что не очень-то удобно. Измени его на**protected**, чтобы все потомки класса**BaseObject**имели контроль над своими  
координатами и размерами, без необходимости пользоваться «**чужими**»***геттерами***и***сеттерами***.

### Арканоид(15)

Не поверишь, но и это еще не все.

Во-первых нужен метод setDirection,  
который не только устанавливает значение переменной direction,  
но и вычисляет новые значения переменных dx и dy.

**Код должен выглядеть примерно так:**  
this.direction = direction;  
double angel = Math.toRadians(direction);  
dx = Math.cos(angle) \* speed;  
dy = -Math.sin(angle) \* speed.

Во-вторых шарик может удариться о стенку.  
При этом он должен от нее отскочить.

**Для этого нам понадобится еще один метод:**  
void checkRebound(int **minx**, int **maxx**, int **miny**, int **maxy**)

Создай его, а кодом я займусь сам.

### Арканоид(16)

И наконец «***подставка***«!  
Ей понадобятся такие приватные поля:  
а) speed (скорость шарика) типа double;  
б) direction (направление движения по оси x: **1** — вправо, **-1** — влево, **0** — начальное значение, стоим на месте) типа double.

**Также создай для них геттеры.**

**А еще с тебя конструктор, примерно вот такой:**  
public Stand(double x, double y) {  
super(x, y, 3);  
speed = 1;  
direction = 0;  
}

### Арканоид(17)

**Еще подставке нужны методы:**  
а) move — см. **move** в **BaseObject**  
Движение доски осуществляется горизонтально, поэтому мы меняем только координату х.  
Подумай, как координата х зависит от направления (direction) и скорости (speed). Реализуй зависимость.  
б) draw — см. draw в **BaseObject**  
Его кодом я займусь сам.  
в) moveLeft() — задает постоянное движение «***подставки***» **влево**  
Просто присвой правильное значение переменной **direction**.  
г) moveRight() — задает постоянное движение «***подставки***» **вправо**  
Просто присвой правильное значение переменной direction.

### Арканоид(18)

Теперь вернемся к классу **Arkanoid**.

Реализуй методы:  
а) move()  
В этом методе нужно двигать все движимые объекты (stand**,**ball).  
б) draw(Canvas **canvas**)  
В этом методе надо вызвать метод draw всех существующих объектов, которые его имеют.

### Арканоид(19)

Осталось совсем чуть-чуть.

В классе **Arkanoid** создай поле приватное isGameOver типа boolean.

Реализуй метод checkBricksBump.  
В этом методе надо проверить — не столкнулся ли шарик с каким-нибудь из «***кирпичей***«.  
Для проверки столкновения используй метод isIntersec.  
**Если шарик все-таки попал в кирпич, то:**  
а) шарик ***отлетает*** в случайном направлении:  
double angle = Math.random() \* 360;  
ball.setDirection(angle);  
б) кирпич ***умирает*** — надо удалить его из списка всех кирпичей.

Реализуй метод checkStandBump.  
В этом методе надо проверить — не ударился ли шарик о подставку.  
Для проверки столкновения используй метод isIntersec.  
Если шарик все-таки ударился о подставку, то:  
шарик отлетает в случайным направлении вверх:  
double angle = 90 + 20 \* (Math.random() — 0.5);  
ball.setDirection(angel);

Реализуй метод checkEndGame.  
Если координата y шарика больше чем высота поля игры (height), значит шарик улетел вниз за границу экрана.  
В этом случае надо переменную isGameOver установить в **true**.

### Арканоид(20)

**Отличная работа!** Я добавил пару методов, а также класс **KeyboardObserver**.

Немного отдохни перед новым уровнем и поиграй.

**P.S.** Только не забудь отрегулировать высоту консоли.