package com.javarush.task.task25.task2501;  
  
/\*   
Новые возможности!  
Используя возможности enum-а реализуй метод getLetterPosition, который должен возвращать позицию буквы.  
Позиция буквы A — 1.  
  
  
Требования:  
1. Класс Solution должен содержать enum Alphabet.  
2. Alphabet должен содержать по порядку 26 букв английского алфавита.  
3. Метод getLetterPosition у класса Alphabet должен возвращать позицию буквы в алфавите.  
4. Метод getLetterPosition должен вызывать метод ordinal.  
\*/  
public class Solution {  
 public static void main(String[] args) {  
 System.*out*.println(Alphabet.*G*.getLetterPosition());  
 }  
  
 public enum Alphabet {  
 *A*, *B*, *C*, *D*, *E*,  
 *F*, *G*, *H*, *I*, *J*,  
 *K*, *L*, *M*, *N*, *O*,  
 *P*, *Q*, *R*, *S*, *T*,  
 *U*, *V*, *W*, *X*, *Y*, *Z*;  
  
 int getLetterPosition() {  
 return -1;  
 }  
 }  
}

package com.javarush.task.task25.task2502;  
  
import java.util.List;  
  
/\*   
Машину на СТО не повезем!  
Инициализируй поле wheels используя данные из loadWheelNamesFromDB.  
Выкинь исключение в случае некорректных данных.  
  
Подсказка: если что-то не то с колесами, то это не машина!  
Сигнатуры не менять.  
  
  
Требования:  
1. Enum Wheel в классе Solution менять нельзя.  
2. Сигнатуры в классе Car менять нельзя.  
3. Во время создания машины нужно вызвать метод loadWheelNamesFromDB.  
4. В случае возврата неправильных данных о колесах, нужно кинуть исключение.  
5. Инициализируй поле wheels полученными данными.  
\*/  
public class Solution {  
 public static enum Wheel {  
 *FRONT\_LEFT*,  
 *FRONT\_RIGHT*,  
 *BACK\_LEFT*,  
 *BACK\_RIGHT* }  
  
 public static class Car {  
 protected List<Wheel> wheels;  
  
 public Car() {  
 //init wheels here  
 }  
  
 protected String[] loadWheelNamesFromDB() {  
 //this method returns mock data  
 return new String[]{"FRONT\_LEFT", "FRONT\_RIGHT", "BACK\_LEFT", "BACK\_RIGHT"};  
 }  
 }  
  
 public static void main(String[] args) {  
 }  
}

package com.javarush.task.task25.task2503;  
  
/\*   
Свой enum  
Реализуй интерфейс Columnable у Column, описание методов смотрите в джавадоках.  
Реализуй логику метода Column.getVisibleColumns.  
Метод Column.configureColumns уже реализован, его не меняй.  
Создавать дополнительные поля нельзя.  
Метод main не участвует в тестировании.  
  
  
Требования:  
1. Интерфейс Columnable менять нельзя.  
2. Класс Column должен реализовывать интерфейс Columnable.  
3. Создавать дополнительные поля в классе Column нельзя.  
4. Метод Column.configureColumns реализован. Менять его не нужно.  
5. Метод Column.getVisibleColumns должен возвращать список отображаемых колонок в скофигурированом порядке.  
6. Метод Column.getColumnName должен возвращать полное имя колонки.  
7. Метод Column.isShown должен возвращать true, если колонка видимая, иначе false.  
8. Метод Column.hide должен скрывать колонку и сдвигать индексы остальных отображаемых колонок  
\*/  
public class Solution {  
 */\*\*  
 \* Output:  
 \* <p/>  
 \* Available Amount  
 \* Account Number  
 \* Bank Name  
 \* --------------------  
 \* Available Amount  
 \* Bank Name  
 \*/* public static void main(String[] args) {  
  
 Column.*configureColumns*(Column.*Amount*, Column.*AccountNumber*, Column.*BankName*);  
  
 for (Columnable columnable : Column.*getVisibleColumns*()) {  
 System.*out*.println(columnable.getColumnName());  
 }  
  
 System.*out*.println("--------------------");  
 Column.*AccountNumber*.hide();  
  
 for (Columnable columnable : Column.*getVisibleColumns*()) {  
 System.*out*.println(columnable.getColumnName());  
 }  
 }  
}

package com.javarush.task.task25.task2503;  
  
import java.util.LinkedList;  
import java.util.List;  
  
public enum Column {  
 *Customer*("Customer"),  
 *BankName*("Bank Name"),  
 *AccountNumber*("Account Number"),  
 *Amount*("Available Amount");  
  
  
 private String columnName;  
  
 private static int[] realOrder;  
  
 private Column(String columnName) {  
 this.columnName = columnName;  
 }  
  
 */\*\*  
 \* Задает новый порядок отображения колонок, который хранится в массиве realOrder.  
 \* realOrder[индекс в энуме] = порядок отображения; -1, если колонка не отображается.  
 \*  
 \** ***@param*** *newOrder новая последовательность колонок, в которой они будут отображаться в таблице  
 \** ***@throws*** *IllegalArgumentException при дубликате колонки  
 \*/* public static void configureColumns(Column... newOrder) {  
 realOrder = new int[*values*().length];  
 for (Column column : *values*()) {  
 realOrder[column.ordinal()] = -1;  
 boolean isFound = false;  
  
 for (int i = 0; i < newOrder.length; i++) {  
 if (column == newOrder[i]) {  
 if (isFound) {  
 throw new IllegalArgumentException("Column '" + column.columnName + "' is already configured.");  
 }  
 realOrder[column.ordinal()] = i;  
 isFound = true;  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* Вычисляет и возвращает список отображаемых колонок в сконфигурированом порядке (см. метод configureColumns)  
 \* Используется поле realOrder.  
 \*  
 \** ***@return*** *список колонок  
 \*/* public static List<Column> getVisibleColumns() {  
 List<Column> result = new LinkedList<>();  
  
 return result;  
 }  
}

package com.javarush.task.task25.task2503;  
  
public interface Columnable {  
  
 */\*\*  
 \** ***@return*** *полное имя колонки  
 \*/* String getColumnName();  
  
 */\*\*  
 \* Возвращает true, если колонка видимая, иначе false  
 \*/* boolean isShown();  
  
 */\*\*  
 \* Скрывает колонку - маркирует колонку -1 в массиве realOrder.  
 \* Сдвигает индексы отображаемых колонок, которые идут после скрытой  
 \*/* void hide();  
  
 */\*\*  
 \* Возвращает порядок константы в энуме.  
 \*  
 \** ***@return*** *порядок константы в энуме  
 \*/* int ordinal();  
}

package com.javarush.task.task25.task2504;  
  
/\*   
Switch для нитей  
Обработай список нитей в зависимости от состояния:  
1. Если нить еще не запущена, то запусти ее.  
2. Если нить в ожидании, то прерви ее.  
3. Если нить работает, то проверь маркер isInterrupted.  
4. Если нить прекратила работу, то выведи в консоль ее приоритет.  
Используй switch.  
  
  
Требования:  
1. Метод processThreads принимает аргументом массив нитей.  
2. Если переданная нить не запущена, нужно ее запустить.  
3. Если переданная нить находится в ожидании, нужно ее прервать.  
4. Если переданная нить работает, то нужно проверить маркер isInterrupted.  
5. Если переданная нить завершила работу, нужно вывести в консоль ее приоритет.  
6. Метод processThreads должен использовать оператор switch.  
\*/  
public class Solution {  
 public static void processThreads(Thread... threads) {  
 //implement this method - реализуйте этот метод  
  
   
   
 }  
  
 public static void main(String[] args) {  
 }  
}

package com.javarush.task.task25.task2505;  
  
/\*   
Без дураков  
1. Создай private class MyUncaughtExceptionHandler, который на перехват исключения должен подождать половину секунды, а затем вывести на экран secretKey, имя трэда и сообщение возникшего исключения.  
Используй String.format(…).  
  
Пример:  
super secret key, Thread-0, it's an example  
  
2. Разберись в последовательности выполняемого кода и обеспечь логирование возникновения исключения в п.1.  
3. Метод main в тестировании не участвует.  
  
  
Требования:  
1. Создай private class MyUncaughtExceptionHandler с конструктором по-умолчанию в классе MyThread.  
2. Во время перехвата исключения, MyUncaughtExceptionHandler должен вызвать Thread.sleep(500).  
3. Затем, MyUncaughtExceptionHandler должен выводить в консоль secretKey, имя трэда и сообщение возникшего исключения.  
4. Для вывода сообщения используй String.format(...).  
5. Нужно обеспечить логирование возникновения исключения.  
\*/  
public class Solution {  
  
 public static void main(String[] args) {  
 MyThread myThread = new Solution().new MyThread("super secret key");  
 myThread.start();  
 }  
  
 public class MyThread extends Thread {  
 private String secretKey;  
  
 public MyThread(String secretKey) {  
 this.secretKey = secretKey;  
 setUncaughtExceptionHandler(new MyUncaughtExceptionHandler());  
 setDaemon(true);  
 }  
  
 @Override  
 public void run() {  
 throw new NullPointerException("it's an example");  
 }  
 }  
  
}

package com.javarush.task.task25.task2506;  
  
/\*   
Мониторинг состояния нити  
Cоздай класс нити LoggingStateThread(в отдельном файле ) , которая будет выводить в консоль все состояния (State) переданной в конструктор нити.  
Нить LoggingStateThread должна сама завершаться после остановки переданной в конструктор нити.  
Метод main не участвует в тестировании.  
  
  
Требования:  
1. Создай класс LoggingStateThread. Он должен наследовать класс Thread.  
2. Класс LoggingStateThread должен содержать поле нити, за которой он будет следить. Это поле должно инициализироваться через конструктор.  
3. Переопредели метод run в классе LoggingStateThread.  
4. После запуска класс LoggingStateThread должен выводить в консоль изменения состояния переданной нити.  
5. После завершения работы наблюдаемой нити, LoggingStateThread так же должен завершить работу.  
\*/  
public class Solution {  
 public static void main(String[] args) throws InterruptedException {  
 Thread target = new Thread();  
 LoggingStateThread loggingStateThread = new LoggingStateThread(target);  
  
 loggingStateThread.start();  
 target.start(); //NEW  
 Thread.*sleep*(100); //RUNNABLE  
 target.join(100);  
 Thread.*sleep*(400);  
 target.interrupt(); //TERMINATED  
 Thread.*sleep*(500);  
 }  
}

package com.javarush.task.task25.task2507;  
  
import java.io.IOException;  
import java.io.InputStream;  
import java.net.Socket;  
  
/\*   
Работать в поте лица!  
Реализуй логику метода interrupt, который должен прерывать трэд предварительно закрыв используемые ресурсы.  
Используй метод super-класса в блоке finally.  
  
  
Требования:  
1. Сигнатуру метода interrupt менять нельзя.  
2. Метод interrupt должен закрывать используемые классом Solution ресурсы.  
3. Затем, метод interrupt должен прерывать трэд (вызов super.interrupt).  
4. Трэд должен быть прерван в любом случае, даже если во время закрытия ресурсов было выкинуто исключение.  
\*/  
public class Solution extends Thread {  
 private static final int *BUFFER\_SIZE* = 2000; //2000 bytes  
 private final Socket socket;  
 private final InputStream in;  
  
 public Solution(Socket socket) throws IOException {  
 this.socket = socket;  
 this.in = socket.getInputStream();  
 }  
  
 public void interrupt() {  
 //implement logic here  
 }  
  
 public void run() {  
 try {  
 byte[] buf = new byte[*BUFFER\_SIZE*];  
 while (true) {  
 int count = in.read(buf);  
 if (count < 0) {  
 break;  
 } else {  
 if (count > 0) {  
 //process buffer here  
 }  
 }  
 }  
 } catch (IOException ignored) {}  
 }  
  
 public static void main(String[] args) {  
 }  
}

package com.javarush.task.task25.task2508;  
  
/\*   
Не валять дурака  
Восстанови логику класса TaskManipulator.  
  
  
Требования:  
1. Класс TaskManipulator должен реализовывать интерфейсы Runnable и CustomThreadManipulator.  
2. Метод run должен каждые 100 секунд выводить имя исполняемой нити в консоль.  
3. Класс TaskManipulator должен иметь внутреннее поле типа Thread.  
4. Метод start должен создавать, сохранять во внутреннее поле и запускать нить с именем, которое передано через аргумент метода.  
5. Метод stop должен прерывать последнюю созданную классом TaskManipulator нить.  
\*/  
public class Solution  
{  
 /\*  
 Output:  
 first  
 first  
 second  
 second  
 second  
 third  
 fifth  
 \*/  
 public static void main(String[] args) throws InterruptedException {  
 CustomThreadManipulator manipulator = new TaskManipulator();  
  
 manipulator.start("first");  
 Thread.*sleep*(150);  
 manipulator.stop();  
  
 manipulator.start("second");  
 Thread.*sleep*(250);  
 manipulator.stop();  
  
 manipulator.start("third");  
 Thread.*sleep*(50);  
 manipulator.stop();  
  
 manipulator.start("forth");  
 manipulator.stop();  
  
 manipulator.start("fifth");  
 Thread.*sleep*(1);  
 manipulator.stop();  
 }  
}

package com.javarush.task.task25.task2508;  
  
public interface CustomThreadManipulator {  
 public void start(String threadName);  
 public void stop();  
}

package com.javarush.task.task25.task2508;  
  
public class TaskManipulator implements Runnable {  
 @Override  
 public void run() {  
 }  
}

package com.javarush.task.task25.task2509;  
  
import java.util.concurrent.\*;  
  
/\*   
Все не так легко, как кажется  
1. Почитать в инете про Socket, ThreadPoolExecutor, RunnableFuture, Callable.  
2. Реализуй логику метода cancel в классе SocketTask.  
3. Реализуй логику метода cancel для локального класса внутри метода newTask в классе SocketTask.  
  
  
Требования:  
1. Определение класса SocketTask, его поля и сигнатуры методов менять нельзя.  
2. Метод cancel в классе SocketTask должен закрывать используемые классом ресурсы.  
3. Метод cancel для локального класса внутри метода newTask должен закрывать ресурсы SocketTask и вызвать cancel у родительского класса.  
4. Метод у родительского класса должен быть вызван, даже если во время закрытия ресурсов было выкинуто исключение.  
\*/  
public class Solution extends ThreadPoolExecutor {  
 public Solution(int corePoolSize, int maximumPoolSize, long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {  
 super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);  
 }  
  
 protected <T> RunnableFuture<T> newTaskFor(Callable<T> callable) {  
 if (callable instanceof CancellableTask) {  
 return ((CancellableTask<T>) callable).newTask();  
 } else {  
 return super.newTaskFor(callable);  
 }  
 }  
  
 public static void main(String[] args) {  
 }  
}

package com.javarush.task.task25.task2509;  
  
import java.net.Socket;  
import java.util.concurrent.FutureTask;  
import java.util.concurrent.RunnableFuture;  
  
public abstract class SocketTask<T> implements CancellableTask<T> {  
 private Socket socket;  
  
 protected synchronized void setSocket(Socket socket) {  
 this.socket = socket;  
 }  
  
 public synchronized void cancel() {  
 //close all resources here  
 }  
  
 public RunnableFuture<T> newTask() {  
 return new FutureTask<T>(this) {  
 public boolean cancel(boolean mayInterruptIfRunning) {  
 //close all resources here by using proper SocketTask method  
 //call super-class method in finally block  
 return false;  
 }  
 };  
 }  
}

package com.javarush.task.task25.task2509;  
  
import java.util.concurrent.Callable;  
import java.util.concurrent.RunnableFuture;  
  
public interface CancellableTask<T> extends Callable<T> {  
 void cancel();  
  
 RunnableFuture<T> newTask();  
}

package com.javarush.task.task25.task2510;  
  
/\*   
Поживем - увидим  
Все исключения, которые возникают в процессе работы нити Solution, должны быть обработаны одним из вариантов:  
1. Если это Error, то вывести в консоль «Нельзя дальше работать«.  
2. Если это Exception, то вывести в консоль «Надо обработать«.  
3. Если это Throwable, то вывести в консоль «ХЗ«.  
Реализуй эту логику.  
  
  
Требования:  
1. В конструкторе Solution должен устанавливаться свой UncaughtExceptionHandler, который будет перехватывать возникшие ошибки и выводить текст в консоль.  
2. Если выполнение нити Solution закончилось исключением Error, нужно вывести в консоль "Нельзя дальше работать".  
3. Если выполнение нити Solution закончилось исключением Exception, нужно вывести в консоль "Надо обработать".  
4. Если выполнение нити Solution закончилось исключением Throwable, нужно вывести в консоль "ХЗ".  
\*/  
public class Solution extends Thread {  
  
 public Solution() {  
 }  
  
 public static void main(String[] args) {  
 }  
}

package com.javarush.task.task25.task2511;  
  
import java.util.TimerTask;  
  
/\*   
Вооружаемся до зубов!  
Создай свой UncaughtExceptionHandler в виде локального класса внутри конструктора.  
UncaughtExceptionHandler должен маскировать звездочками имя трэда и выводить в консоль описание возникшей ошибки.  
«Thread-0» должно быть заменено на «\*\*\*\*\*\*\*\*».  
«Thread-4321» должно быть заменено на «\*\*\*\*\*\*\*\*\*\*\*».  
  
  
Требования:  
1. Определение класса Solution и его поля менять нельзя.  
2. Конструктор Solution должен создавать свой UncaughtExceptionHandler, и сохранять его в поле handler.  
3. Созданный UncaughtExceptionHandler должен выводить описание возникшей ошибки в консоль.  
4. В описании ошибки имя трэда должно быть замаскировано символами "\*".  
\*/  
public class Solution extends TimerTask {  
 protected TimerTask original;  
 protected final Thread.UncaughtExceptionHandler handler;  
  
 public Solution(TimerTask original) {  
 if (original == null) {  
 throw new NullPointerException();  
 }  
 this.original = original;  
 this.handler = null; //init handler here  
 }  
  
 public void run() {  
 try {  
 original.run();  
 } catch (Throwable cause) {  
 Thread currentThread = Thread.*currentThread*();  
 handler.uncaughtException(currentThread, new Exception("Blah " + currentThread.getName() + " blah-blah-blah", cause));  
 }  
 }  
  
 public long scheduledExecutionTime() {  
 return original.scheduledExecutionTime();  
 }  
  
 public boolean cancel() {  
 return original.cancel();  
 }  
  
 public static void main(String[] args) {  
 }  
}

package com.javarush.task.task25.task2512;  
  
/\*   
Живем своим умом  
В классе Solution реализуй интерфейс UncaughtExceptionHandler, который должен:  
1. прервать нить, которая бросила исключение.  
2. вывести в консоль стек исключений, начиная с самого вложенного.  
  
Пример исключения:  
new Exception("ABC", new RuntimeException("DEF", new IllegalAccessException("GHI")))  
  
Пример вывода:  
java.lang.IllegalAccessException: GHI  
java.lang.RuntimeException: DEF  
java.lang.Exception: ABC  
\*/  
public class Solution implements Thread.UncaughtExceptionHandler {  
  
 @Override  
 public void uncaughtException(Thread t, Throwable e) {  
 }  
  
 public static void main(String[] args) {  
 }  
}

package com.javarush.task.task25.task2513;  
  
import java.util.Random;  
  
/\*   
Обеспечение отсутствия прерывания важной операции  
Просмотри метод moveMoney.  
Если RANDOM.nextInt(5000) больше порогового значения THRESHOLD\_VALUE, то обеспечь переуступку кванта времени (переход хода для текущей нити).  
Добавь этот код в единственное допустимое место.  
  
  
Требования:  
1. Класс Solution должен содержать класс Account.  
2. Метод moveMoney должен перечислять сумму amount с счета from на счет to.  
3. В методе moveMoney вызови Thread.yield если RANDOM.nextInt(5000) больше порогового значения THRESHOLD\_VALUE.  
4. Thread.yield должен быть вызван в правильном месте.  
\*/  
public class Solution {  
 private static final Integer *THRESHOLD\_VALUE* = 500;  
 private static final Random *RANDOM* = new Random();  
  
 public synchronized void moveMoney(Account from, Account to, int amount) {  
 from.setBalance(from.getBalance() - amount);  
 to.setBalance(to.getBalance() + amount);  
 }  
  
 class Account {  
 private int balance;  
  
 public int getBalance() {  
 return balance;  
 }  
  
 public void setBalance(int balance) {  
 this.balance = balance;  
 }  
 }  
  
 public static void main(String[] args) {  
 }  
}

package com.javarush.task.task25.task2514;  
  
/\*   
Первый закон Финэйгла: если эксперимент удался, что-то здесь не так...  
Обеспечь переуступку кванта времени (переход хода для текущей нити) для последовательных выводов текста в консоль.  
  
  
Требования:  
1. Класс Solution должен содержать вложенный класс YieldRunnable, который реализует интерфейс Runnable.  
2. Класс YieldRunnable должен иметь поле с индексом, которое инициализируется через конструктор.  
3. Метод run() должен выводить в консоль сообщения с текущим индексом о начале и конце работы метода.  
4. В правильном месте должен быть вызван Thread.yield.  
\*/  
public class Solution {  
 public static class YieldRunnable implements Runnable {  
 private int index;  
  
 public YieldRunnable(int index) {  
 this.index = index;  
 }  
  
 public void run() {  
 System.*out*.println("begin-" + index);  
 System.*out*.println("end-" + index);  
 }  
 }  
  
 public static void main(String[] args) {  
 }  
}