



Java Memory Model

- Что такое JMM
- Reordering
- Happens-before relationship
- Гарантии volatile
- Double-check locking
- Singleton pattern и JMM
- Safe publication idioms
- Immutable и effectively immutable objects
- Гарантии final

МОЖНО ЛИ ВЫЗВАТЬ GREET() И НЕ УВИДЕТЬ “HELLO”?

```
public class HelloPrinter {  
    private boolean sayHello = true;  
  
    public synchronized void greet() {  
        if (sayHello) {  
            System.out.println("hello");  
        }  
        sayHello = false;  
    }  
}
```

```
variable = 3;
```

Отвечает на вопрос: "При каких условиях поток, который прочитал эту переменную, увидит значение 3?"

- Описывает как потоки взаимодействуют через память. Когда изменения, сделанные одним потоком, будут видны другому
- Описывает корректное поведение программы в многопоточной среде на различных архитектурах с различными hardware memory model
- Описывает работу с shared variables

Оригинальная Java memory model была разработана в 1995 году.

Была обновлена через Java Community Process (JSR-133) в 2004г.
Изменения появились в Java 5.0.

Описывается в 17 главе JLS:

http://java.sun.com/docs/books/jls/third_edition/html/memory.html

$x = y = 0$

Thread 1

$a = x;$

$y = 1;$

Thread 2

$b = y;$

$x = 1;$

$a = ? \ b = ?$

$x = y = 0$

Thread 1

$a = x;$

$y = 1;$

Thread 2

$b = y;$

$x = 1;$

-
1. $a = 0; b = 0;$
 2. $a = 1; b = 0;$
 3. $a = 0; b = 1;$

$x = y = 0$

Thread 1

$a = x;$

$y = 1;$

Thread 2


$b = y;$

$x = 1;$


-
1. $a = 0; b = 0;$
 2. $a = 1; b = 0;$
 3. $a = 0; b = 1;$
 4. $a = 1; b = 1; \text{ or } 0 ?$

$x = y = 0$

Thread 1

 $a = x;$
 $y = 1;$


Thread 2

$b = y;$ 
 $x = 1;$


-
1. $a = 0; b = 0;$
 2. $a = 1; b = 0;$
 3. $a = 0; b = 1;$
 4. $a = 1; b = 1; \text{оО ?}$

$x = y = 0$

Thread 1

 $y = 1;$
 $a = x;$

Thread 2

 $x = 1;$ 
 $b = y;$

-
1. $a = 0; b = 0;$
 2. $a = 1; b = 0;$
 3. $a = 0; b = 1;$
 4. $a = 1; b = 1;$

1. Компилятор может в качестве оптимизации свободно переупорядочивать определенные инструкции, если это не меняет семантику программы.
2. Процессору позволяется исполнять операции не по порядку в некоторых обстоятельствах.
3. Кэшу, как правило, позволяется выполнять обратную запись переменных в основную память не в том порядке, в котором они были записаны программой.

JMM оперирует терминами actions (read, write, lock, unlock,...)

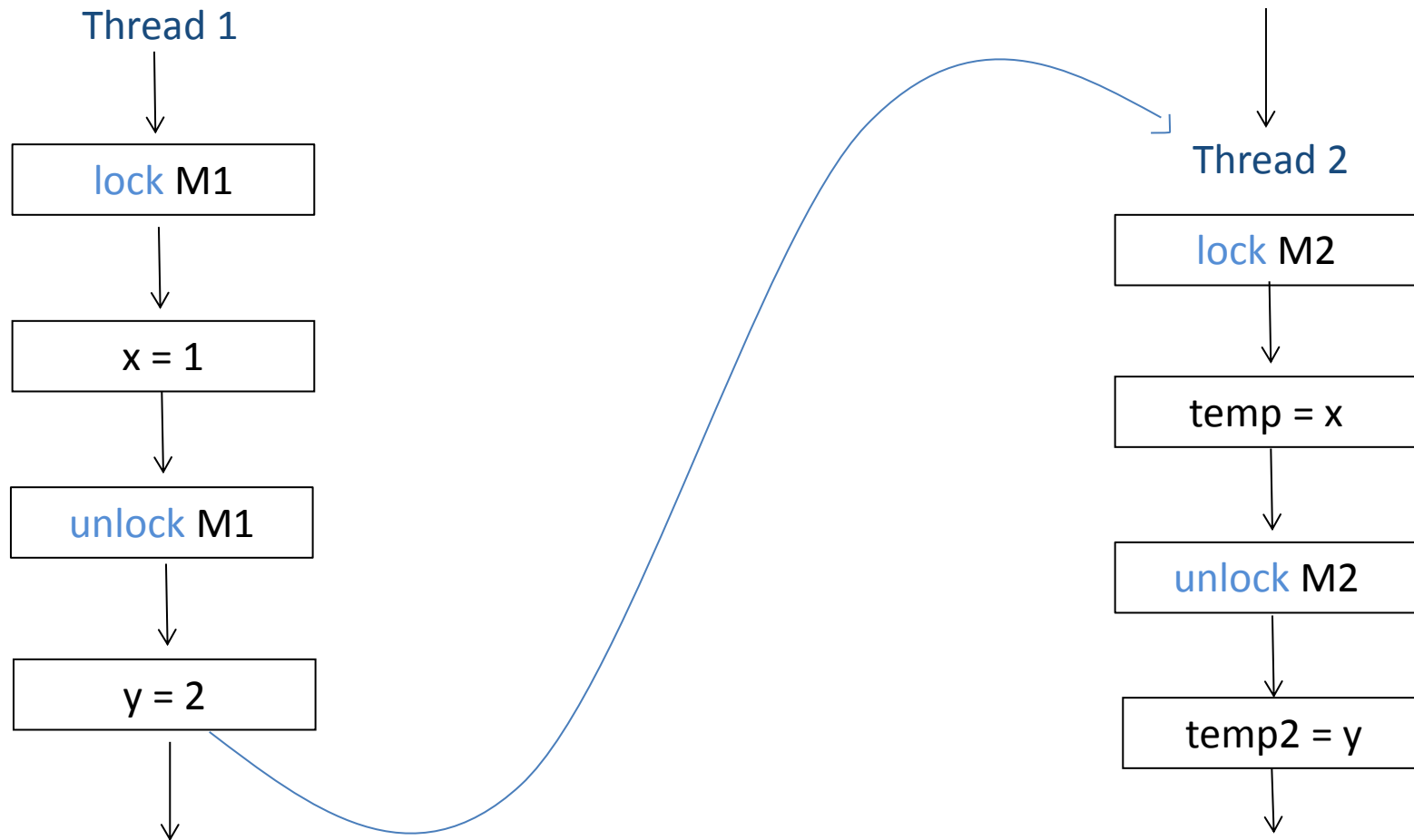
JMM гарантирует что поток, выполнивший action B увидит результат действия A, только если A и B состоят в отношении happens-before. (A happens-before B)

JMM оперирует терминами actions (read, write, lock, unlock,...)

JMM гарантирует что поток, выполнивший action В увидит результат действия А, только если А и В состоят в отношении happens-before. (А happens-before В)

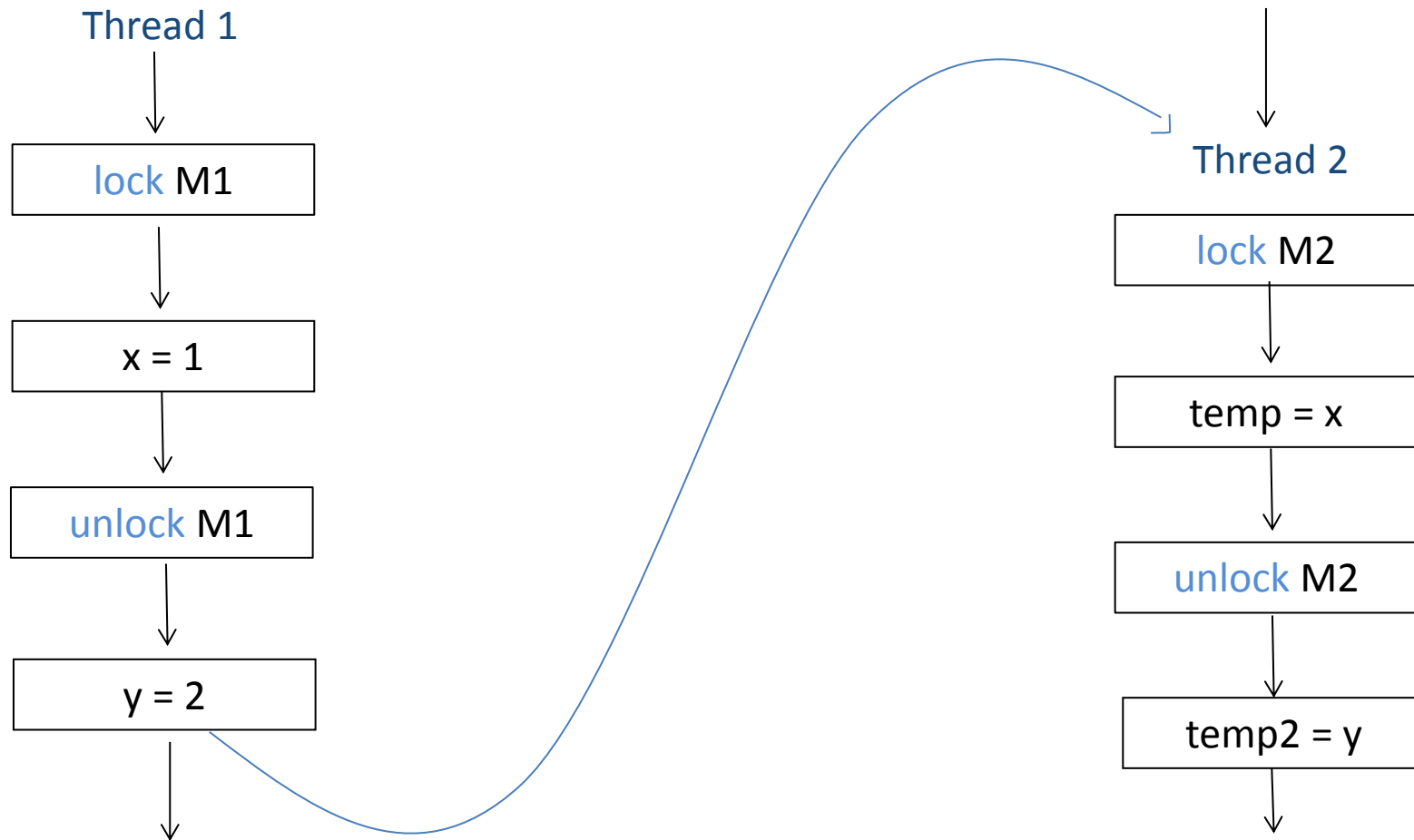
$hb(x, y)$ и $hb(y, z)$, то $hb(x, z)$

ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?

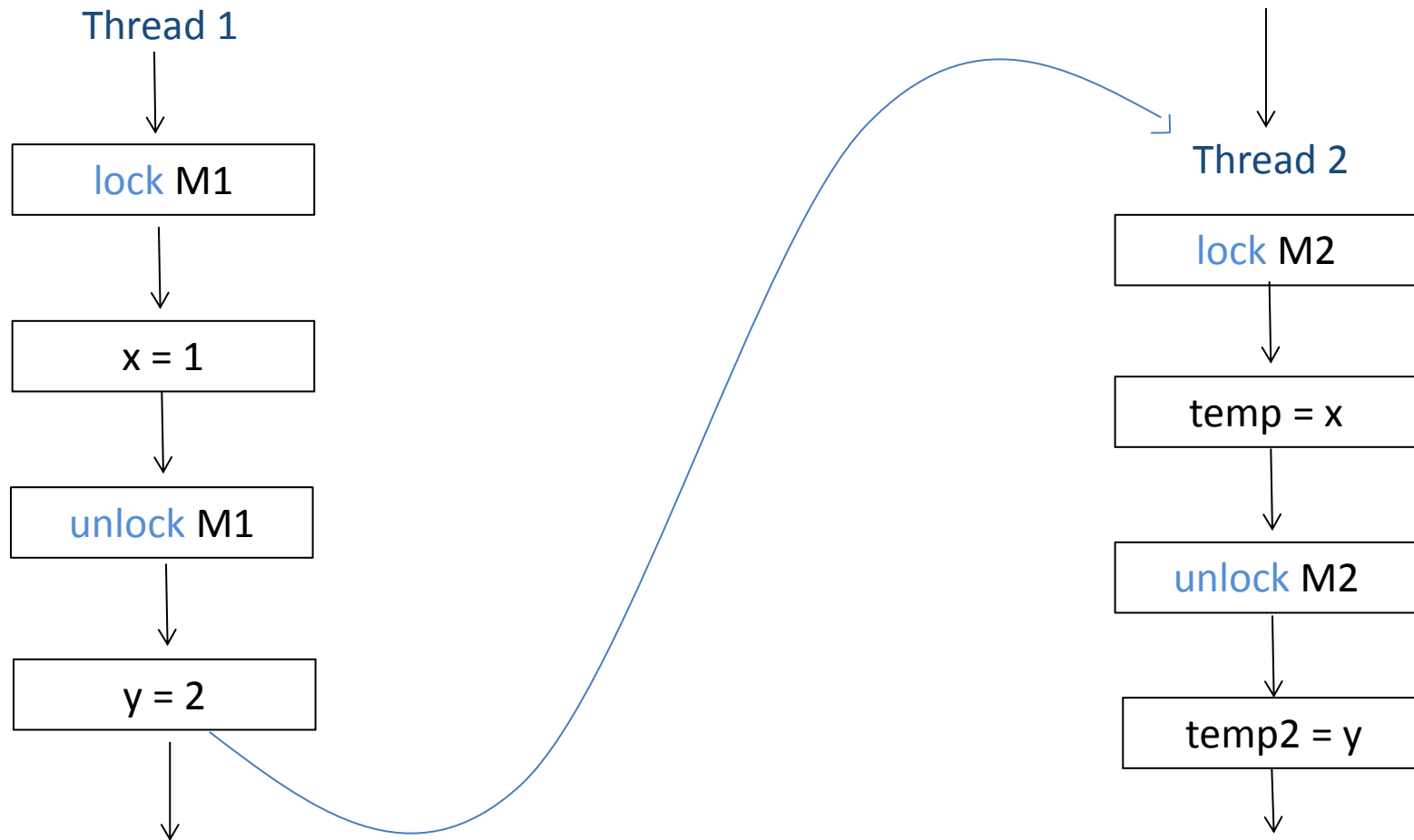


- Действия в потоке happens-before последующего действия в этом же потоке, которое идет позже в программном порядке
- Разблокировка монитора happens-before каждой последующей блокировки **того же монитора**
- Запись в поле volatile happens-before каждого последующего считывания **того же самого volatile**
- Вызов Thread::start() у потока happens-before первого действия в этом потоке
- Завершение потока T1 happens-before момента, когда поток T2 определил, что T1 завершился, вызвав T1.join() или T1.isAlive() (и получив false)
- Завершение конструктора объекта happens-before запуска финализации для него
- Еще несколько пунктов ...

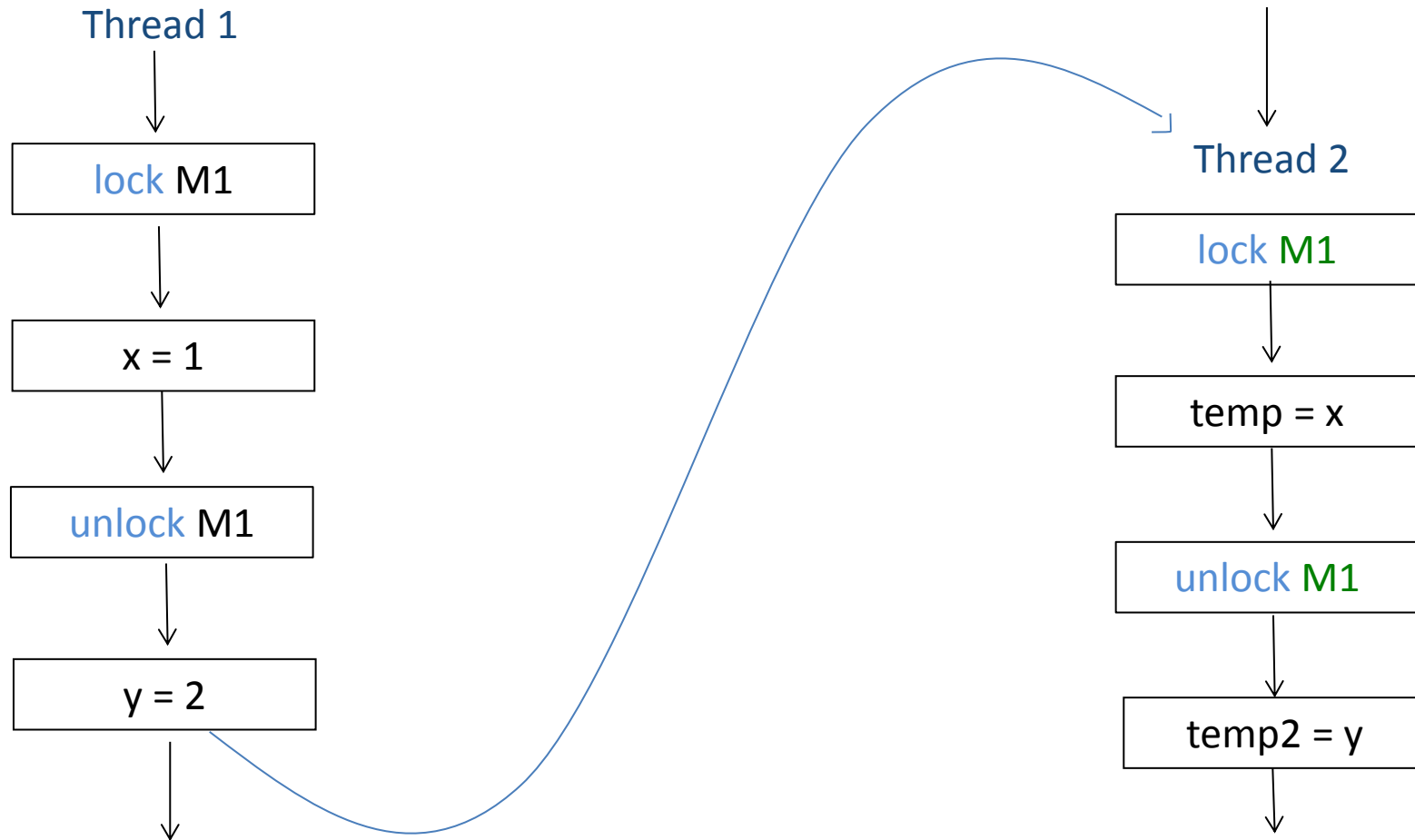
ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?



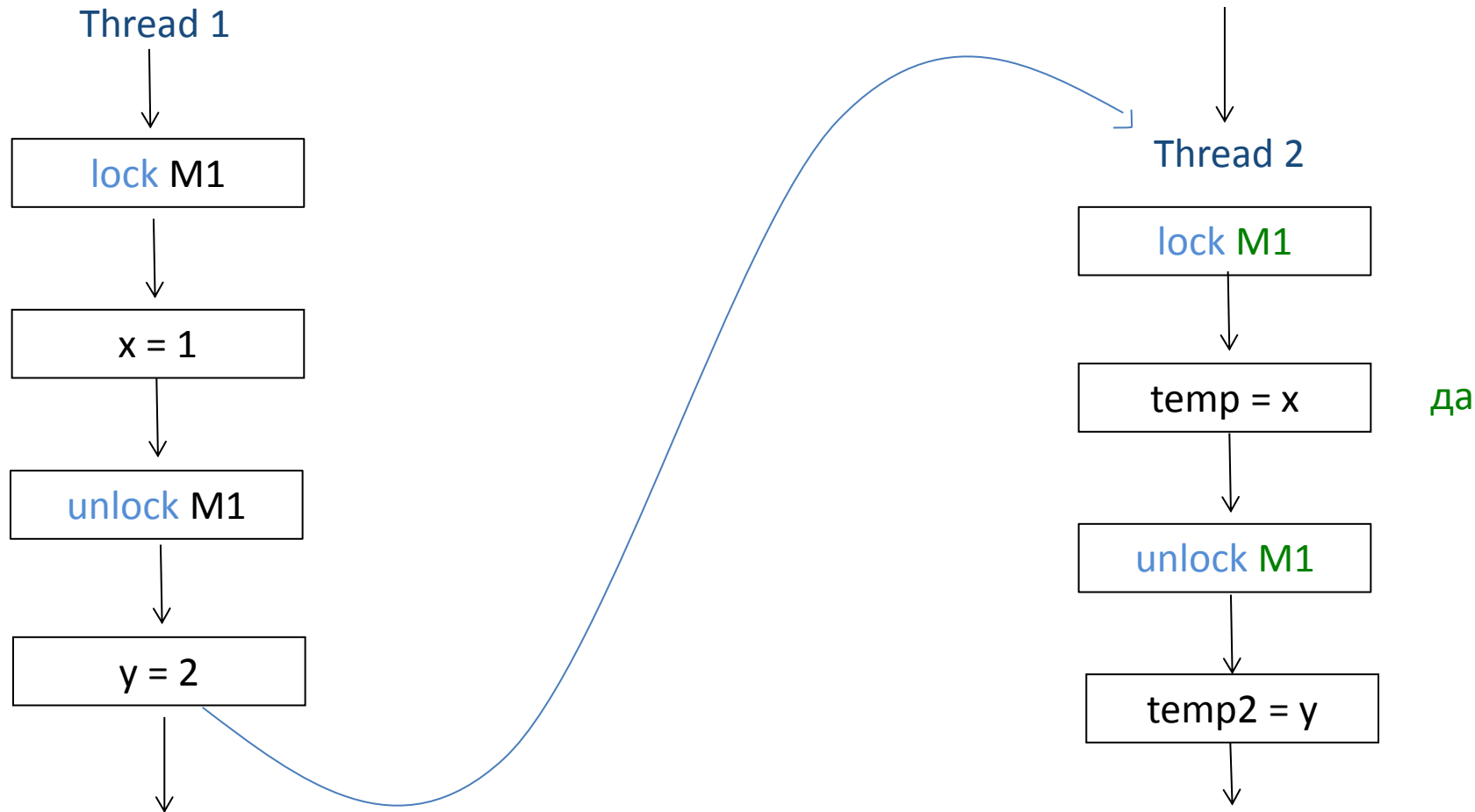
T2 НЕ ОБЯЗАН УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ T1



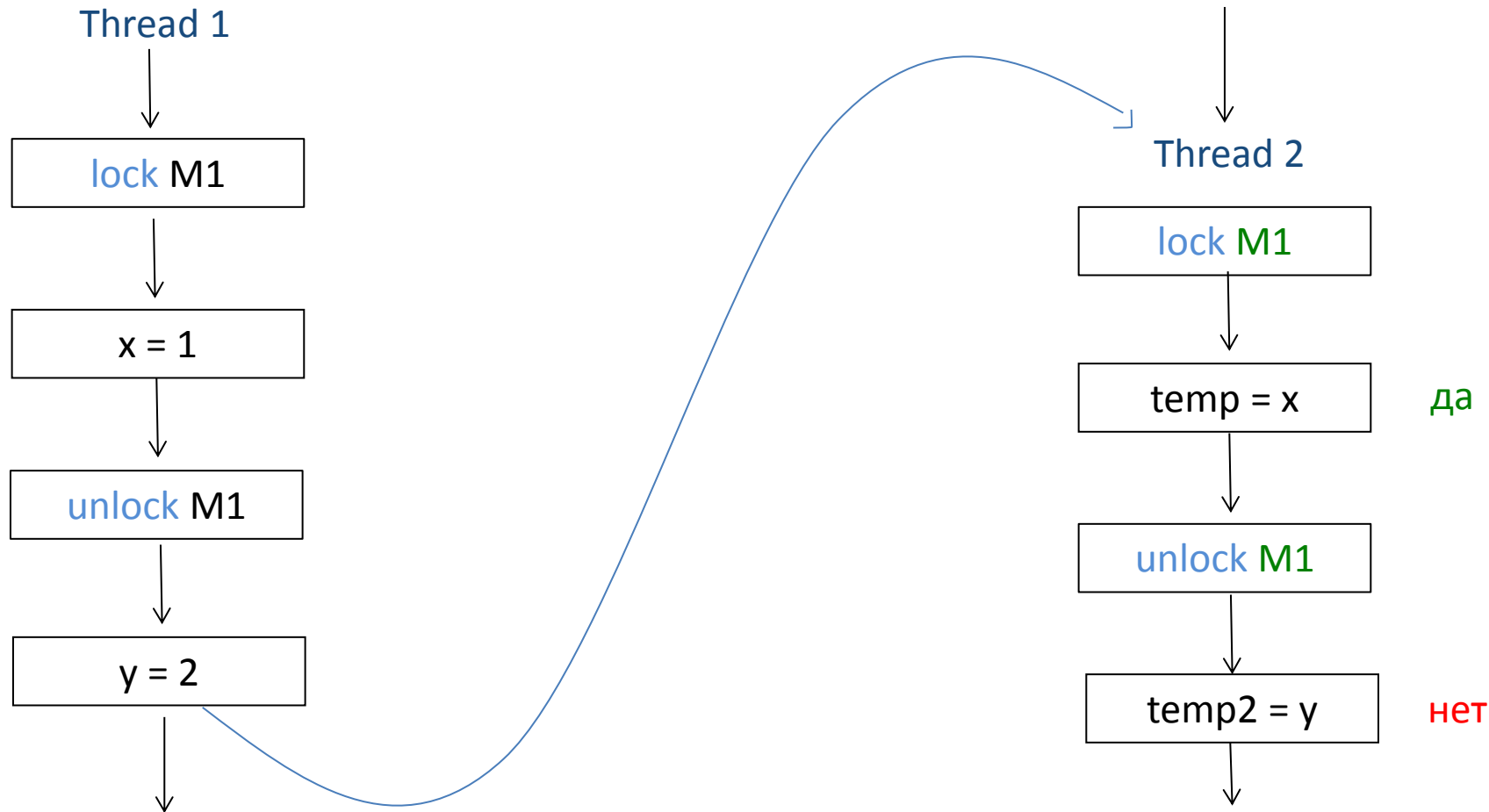
ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?



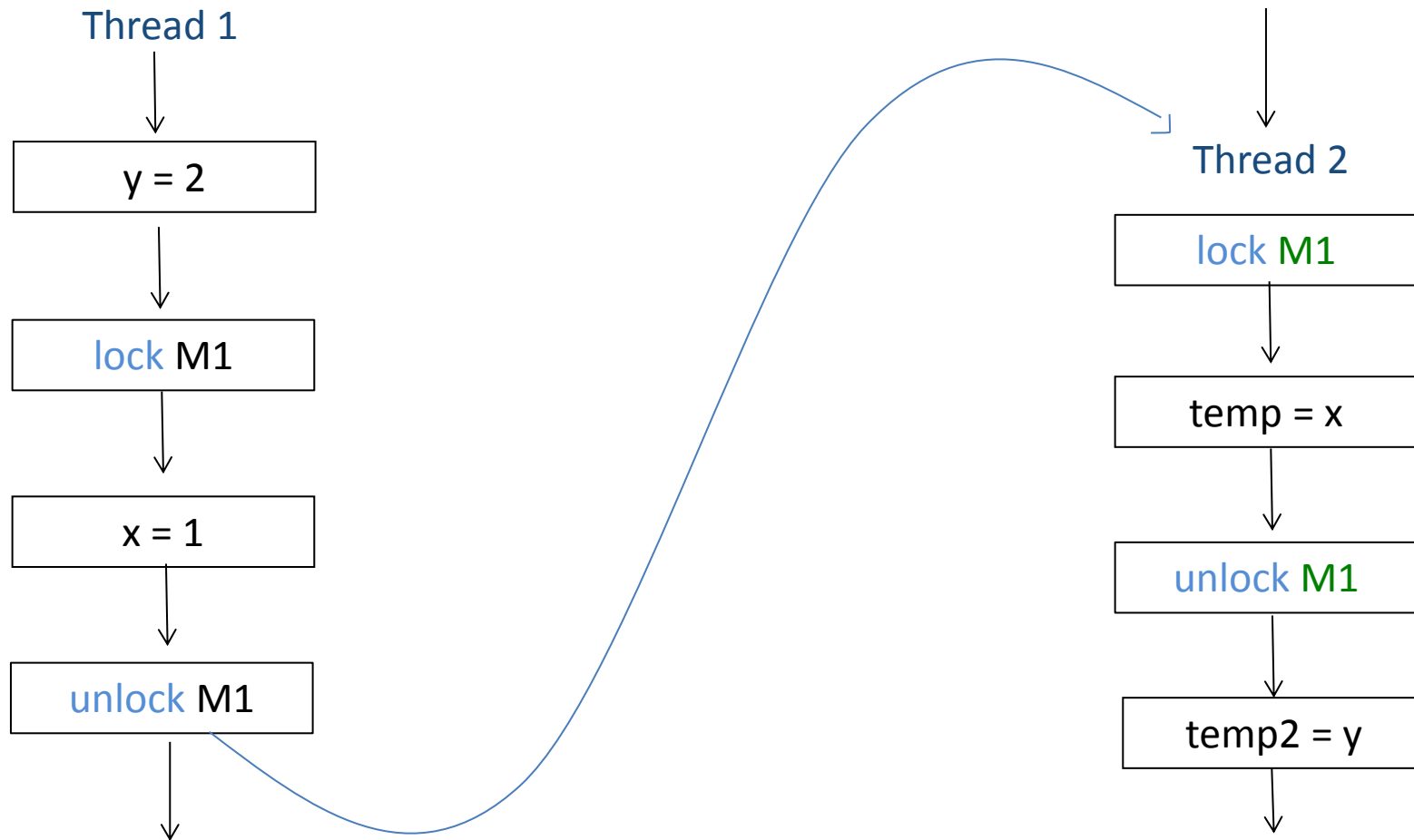
ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?



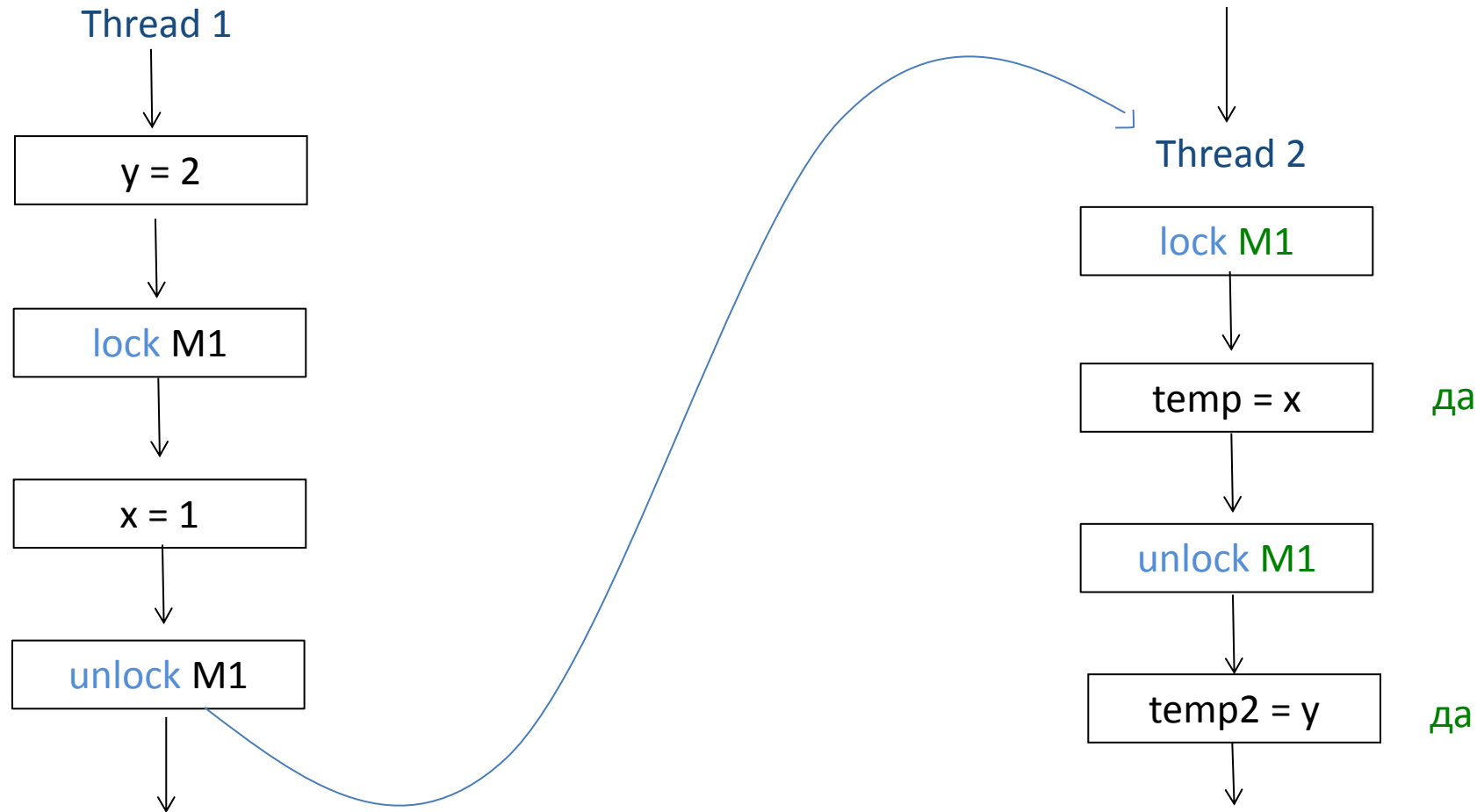
ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?



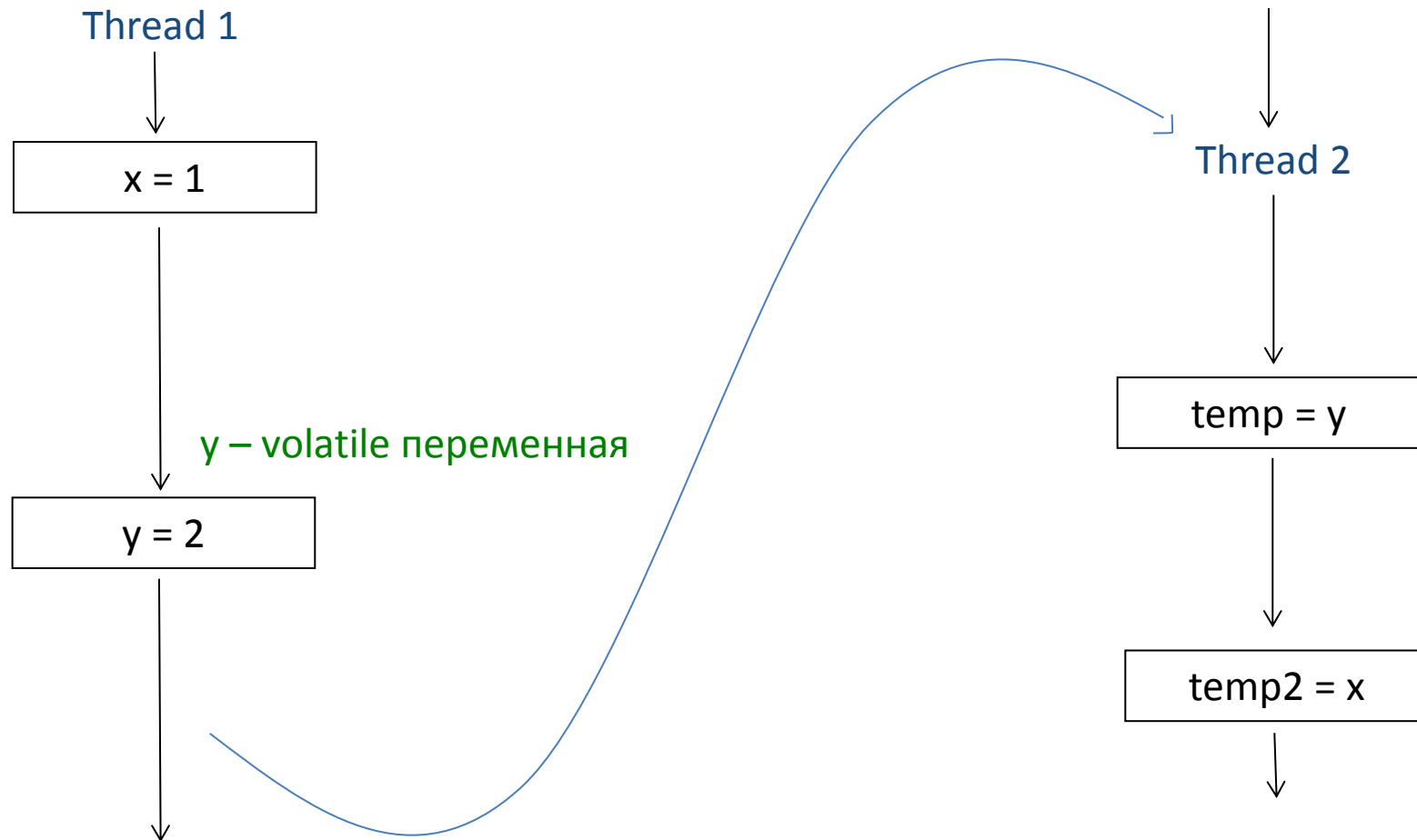
ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?



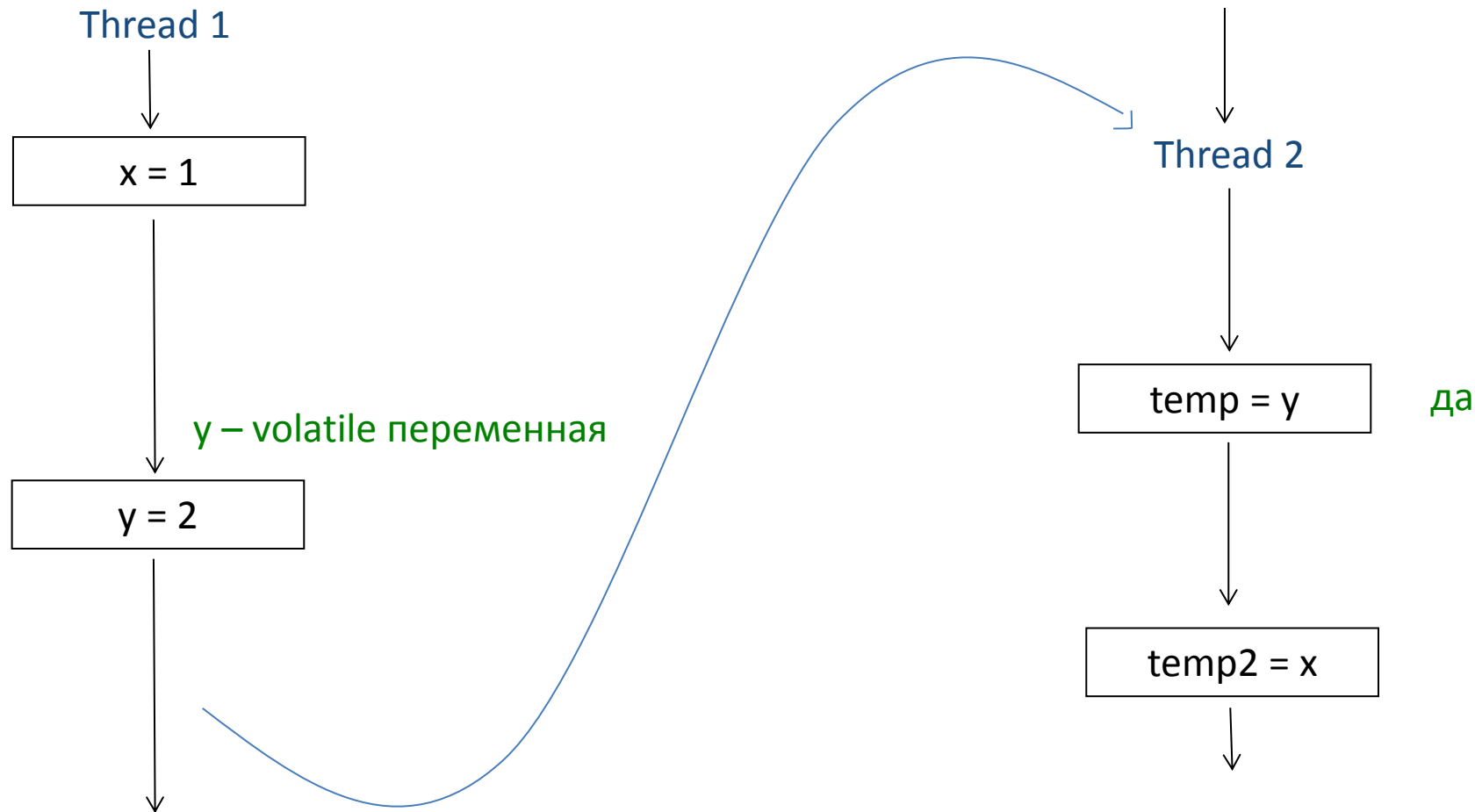
ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?



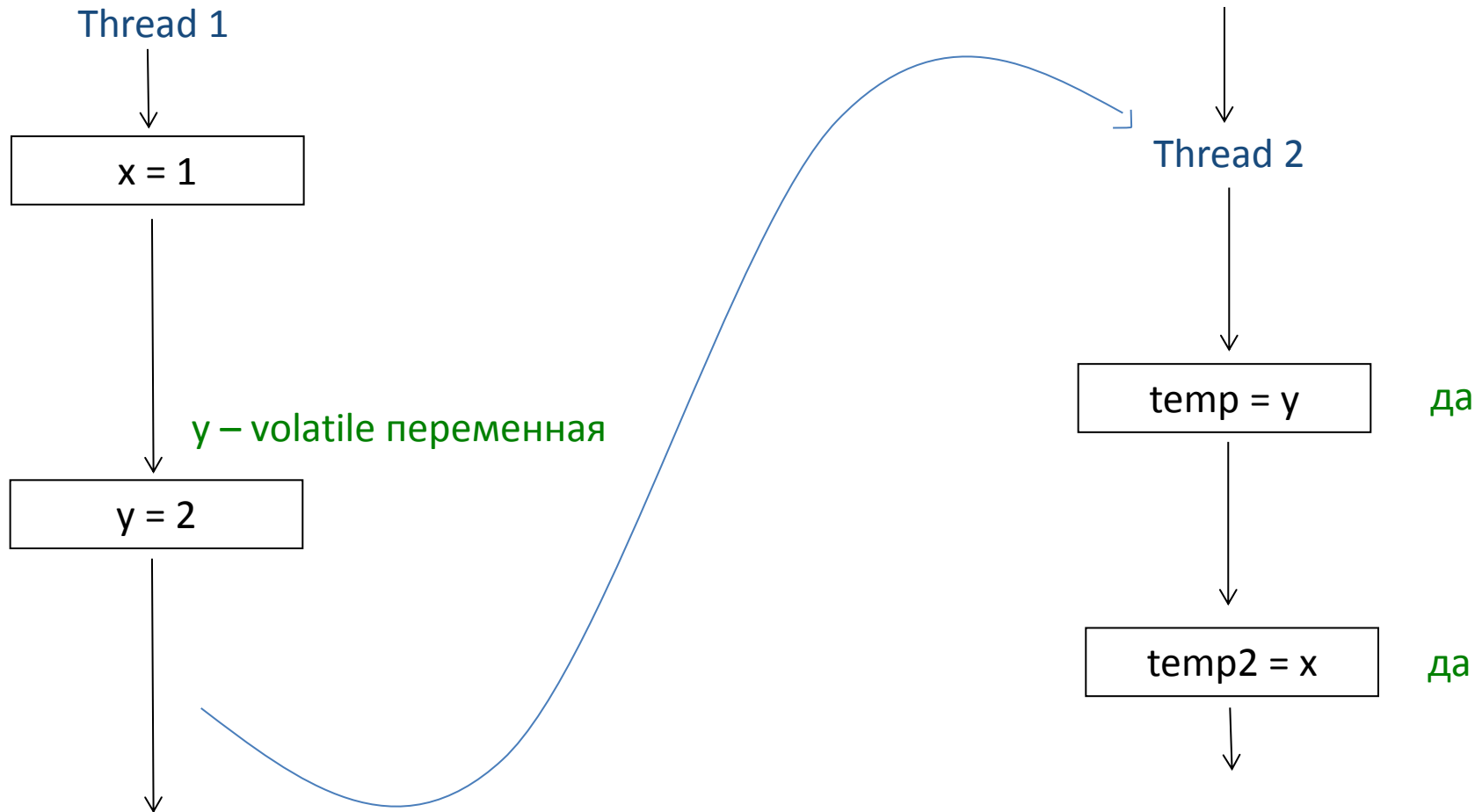
ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?



ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?



ОБЯЗАН ЛИ Т2 УВИДЕТЬ ИЗМЕНЕНИЯ СДЕЛАННЫЕ Т1?









- Гарантии happens-before
- Гарантии с reordering
- Атомарные запись и чтение актуальных значений (даже для Long и Double на 32х разрядных архитектурах)

```
volatile int x = 0;
```

```
public void increment() {  
    ++x; //todo не делать так  
}
```

```
boolean keepRunning = true;
```

```
public void run() {  
    while(keepRunning) {  
        ...  
    }  
}
```

```
public void setKeepRunning(boolean v){...}
```


Point p = new Point(1, 1);

Thread 1

p = new Point(2, 2);

Thread 2

a = p.getX();

a = ?

```
class Point {  
    int x,y;  
}
```

Point p = new Point(1, 1);

Thread 1	Thread 2
p = new Point(2, 2);	a = p.getX();

1. a = 1;

2. a = 2;

Point p = new Point(1, 1);

Thread 1	Thread 2
p = new Point(2, 2);	a = p.getX();

1. a = 1;
2. a = 2;
3. a = 0; oO ??

Point p = new Point(1, 1);

Thread 1

p = new Point(2, 2);

Thread 2

a = p.getX();

1. a = 1;

2. a = 2;

3. a = 0; oO ??

Point p = new Point(1, 1);

Thread 1	Thread 2
synchronized(this) { p = new Point(2, 2); }	a = p.getX();

1. a = 1;

2. a = 2;

3. a = 0; oO ??

Point p = new Point(1, 1);

Thread 1

Point temp = new Point(2, 2);
p = temp;

Thread 2

a = p.getX();

1. a = 1;

2. a = 2;

3. a = 0; oO ??

THREAD-SAFE ЛИ ДАННЫЙ КОД ?

```
public class HelloPrinter {  
    private boolean sayHello;
```

```
    HelloPrinter() {  
        sayHello = true;  
    }
```

Конструктор объекта и его синхронизованные методы
не находятся в отношении happens-before

```
    public synchronized void greet() {  
        if (sayHello) {  
            System.out.println("hello");  
        }  
        sayHello = false;  
    }  
}
```

```
public class HelloPrinterFactory {  
    private HelloPrinter instance;  
  
    public HelloPrinter getPrinter() {  
        if (instance == null) {  
            synchronized (this) {  
                if (instance == null) {  
                    instance = new HelloPrinter();  
                }  
            }  
        }  
        return instance;  
    }  
}
```



```
public class HelloPrinterFactory {  
    private HelloPrinter instance;  
  
    public HelloPrinter getPrinter() {  
        if (instance == null) {  
            synchronized (this) {  
                if (instance == null) {  
                    HelloPrinter temp = new HelloPrinter();  
                    instance = temp;  
                }  
            }  
        }  
        return instance;  
    }  
}
```

Ссылка и состояние объекта должны быть видимы потоку.

- Инициализация в static инициализаторе
- Сохранять ссылку на объект в volatile
- Сохранять ссылку в final (this не убегает из конструктора)
- Сохранять ссылку в поле, корректно используя синхронизацию

```
public class HelloPrinterFactory {  
    private volatile HelloPrinter instance;  
  
    public HelloPrinter getPrinter() {  
        if (instance == null) {  
            synchronized (this) {  
                if (instance == null) {  
                    instance = new HelloPrinter();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

```
public class HelloPrinterFactory {  
    private HelloPrinter instance; //если все поля объекта final  
  
    public HelloPrinter getPrinter() {  
        if (instance == null) {  
            synchronized (this) {  
                if (instance == null) {  
                    instance = new HelloPrinter();  
                }  
            }  
        }  
        return instance;  
    }  
}
```

```
public class HelloPrinterFactory {  
    private int value; // для 32 битных примитивов (не для double и long)  
  
    public int getValue() {  
        if (value == null) {  
            synchronized (this) {  
                if (value == 0) {  
                    value = compute();  
                }  
            }  
        }  
        return value;  
    }  
}
```

```
public class HelloPrinterFactory {  
    private static final HelloPrinter instance = new HelloPrinter();  
  
    public HelloPrinter getInstance() {  
        return instance;  
    }  
}
```

```
public class HelloPrinterFactory {  
    private static class Holder {  
        private static final HelloPrinter instance = new HelloPrinter();  
    }  
  
    public HelloPrinter getInstance() {  
        return Holder.instance;  
    }  
}
```

Ни Java Memory Model, ни Java language specification не дают определения `immutable`.


```
public final class Person {  
    private final int age;  
    private final String name;  
  
    public Person(int age, String name) {  
        this.age = age;  
        this.name = name;  
    }  
  
    public int getAge() { return age; }  
  
    public String getName() { return name; }  
}
```

```
public final class Person {  
    private final int age;  
    private final String name;  
  
    public Person(int age, String name) {  
        this.age = age;  
        this.name = name;  
    }  
}
```

final поля объекта будут корректно инициализированы до возвращения ссылки на конструируемый объект и видны всем потокам.

А ЧТО ЕСЛИ ПОЛЕ FINAL, НО ЕГО СОСТОЯНИЕ НЕ FINAL?

```
public final class Corporation {  
    private final Set<String> employees= new HashSet<String>();  
  
    public Corporation () {  
        employees.add("Alex");  
        employees.add("Ben");  
    }  
  
    public boolean isEmployee(String name) {  
        return employees.contains(name);  
    }  
}
```

```
public final class Corporation {  
    private final Set<String> employees= new HashSet<String>();  
  
    public Corporation () {  
        employees.add("Alex");  
        employees.add("Ben");  
    }  
}
```

`final` гарантирует, что ссылка на объект вернется после завершения конструктора, так что и тут все ок

```
public final class Person {  
    private int age;  
    private String name;  
  
    public Person(int age, String name) {  
        this.age = age;  
        this.name = name;  
    }  
  
    public int getAge() { return age; }  
  
    public String getName() { return name; }  
}
```

Одновременно final volatile - некорректная комбинация модификаторов

- Что такое JMM
- Reordering
- Happens-before relationship
- Гарантии volatile
- Safe publication idioms
- Гарантии final