# Базовые свойства многопоточности

- **Процесс** - это выполняющаяся программа с выделенным ей собственным адресным пространством
- **Поток** - это последовательность команд, исполняемых процессором
- Каждый процесс может иметь один или более потоков
- Механизм управления потоками (но не процессами) встроен в язык Java
- Преимущества многопоточности заключаются в
  - повышении производительности работы приложений
  - возможности программировать длительные операции в отдельных потоках, не разбивая их на короткие отрезки

# Цикл, который загрузит процессор

```
public class AppDemo extends JApplet {
        public AppDemo() {}
        public void init() {
                Container c = getContentPane();
                c.setLayout(new FlowLayout());
                JButton btn = new JButton("Начать вычисления");

                btn.addActionListener(new ActionListener() {
                                for (int i = 0; i < longInteger; i++){
                                    sum += values[i];
                                    …
                                }
                        }
                });
                c.add(btn);
                c.add(txt);
        }
}
```

- В таком примере после нажатия кнопки “Начать вычисления” приложение “подвиснет” до окончания метода обработки события.
- Если выполняются объемные вычисления, то для нормальной работы интерфейса целесообразно выполнять вычисления в **отдельном потоке**

# Создание потока

- Для создания потока в Java служит класс Thread

- Существует 2 варианта создания потока:
  - путем наследования класса Thread
  - путем реализации интерфейса Runnable

# Создание потока – наследника Thread

- Простейшим способом создания потока является создание класса, производного от Thread, с переопределенным в нем методом run()
- Далее следует создать объект созданного класса и вызвать его метод start()
  - Метод start() выполняет системные действия по созданию потока, после чего вызывает переопределенный нами метод run()
- С этого момента начнет свое существование новый поток. Он будет выполняться параллельно с другими потоками, конкурируя с ними за время процессора.
- Когда произойдет выход из метода run(), поток завершит свою работу

# Пример создания потока

Каждые полсекунды поток выводит сообщение на экран:

```
public class NewThread extends Thread {
    public void run() {
     try {
         for (int i = 5; i > 0; i-- ) {
             System.out.println("Дочерний поток: " + i);
             Thread.sleep(500);
         }
     } catch (InterruptedException ex) {
          // Исключение может выбросить метод sleep()
          System.out.println("Дочерний поток прерван.");
     }
     System.out.println("Дочерний поток завершен.");
    }
}
```

Из главного потока создадим дочерний поток:

```
    (new NewThread()).start();
```

# Создание потока путем реализации интерфейса Runnable

- Этот способ применяется, когда наследование класса от Thread невозможно (например, класс уже наследуется от другого класса)

- Шаги, которые нужно выполнить:
  - Создать класс MyTread, реализующий интерфейс Runnable, и реализовать в нем метод run()
  - Создать экземпляр этого класса
  - Создать объект Thread, передав в конструктор экземпляр класса MyThread.
  - Вызвать метод **start()** объекта Thread

# Пример создания потока

```
public class MyThread implements Runnable {
    public void run() {
     //  тело метода такое же,
     // как в предыдущем примере
    }
}
```

Из главного потока создадим дочерний поток:

```
MyTread m = new MyThread();
Thread t = new Thread(m);
t.start();
```

# Диспетчеризация потоков

- Параллельная работа потоков на одном процессоре обеспечивается операционной системой
- Для того, чтобы обеспечить правильную передачу управления между потоками на любой ОС, поток должен периодически отдавать управление операционной системе.
- Для этих целей в классе Thread служат 2 метода
  - sleep(long millis) – приостанавливает работу потока на заданное количество миллисекунд
  - yield() – не приостанавливает работу потока, а просто дает возможность ОС выполнить, при необходимости, переключение на другой поток или процесс
- В любом методе run нужно использовать либо sleep , либо yield, либо какой-то другой их эквивалент (wait или методы чтения из файла). В противном случае можно получить эффект монопольного захвата ресурсов машины одной нитью

# Приоритеты потоков

- Приоритеты потоков определяют порядок передачи управления между ними
- Правила передачи управления следующие:
  - если поток добровольно отказался от управления, оно переходит к потоку с самым высоким приоритетом
  - если поток приостановлен потоком с более высоким приоритетом, то управление передается ему
  - если приоритеты потоков равные, то поведение зависит от операционной системы
- Исходное значение приоритета потока совпадает с приоритетом потока-родителя.
- Приоритет может быть изменен вызовом метода Thread.setPriority() с аргументом из интервала MIN_PRIORITY-MAX_PRIORITY
  - константы определены в классе Thread
- По умолчанию приоритет потока равен NORM_PRIORITY

# Текущий поток

- Ссылку на объект Thread текущего потока можно получить с помощью статического метода
  - Thread.currentThread()

```
public static void main(String[] args) {
    // Распечатывает имя текущего потока
    System.out.println(Thread.currentThread().getName());
}
```

- Все статические методы класса Thread относятся именно к текущему потоку
  - Например, Thread.sleep(1000);

# Имена потоков

- Потоку можно присвоить имя
  - либо с помощью аргумента конструктора,
  - либо методом setName(String).
- Текущее значение имени потока можно получить методом getName()
- Имя потока не используется исполняющей системой и служит для удобства программиста
- Поток обязан иметь имя. Если оно не задано программистом, автоматически вызываемый конструктор потока без аргументов установит его сам
  - По умолчанию главный поток имеет имя main, остальные – Thread-1, 2 и т.д.

# Потоки-демоны

- Поток-демон, как правило, предоставляет некоторые общие услуги и работает в фоновом режиме, пока работает программа
- Незавершенные потоки-демоны разрушаются автоматически,  когда заканчивается последний из пользовательских потоков
- Для обеспечения "демоничности" потоков служат два метода класса Thread:
  - boolean isDaemon() – проверка, является ли демоном
  - setDaemon(boolean on) - устанавливает/снимает признак демона

# Проблема множественного доступа

- Когда несколько потоков разделяют один ресурс, бывает необходимо синхронизировать их работу так, чтобы только один поток имел доступ к ресурсу в каждый момент времени
- Доступ к ресурсу регулируется при помощи монитора
- **Монитор** – это объект, который используется для взаимоисключающей блокировки потоков (mutually exclusive lock)
- Когда один поток начинает выполнять любой из методов объекта-монитора (говорят, что поток входит в монитор), остальные, подойдя к монитору, останавливаются и ждут, пока монитор не освободится
- В Java монитором может служить любой объект.
  - Эта способность передается по наследству от класса Object.

# Средства синхронизации потоков в Java

- На уровне объекта можно синхронизировать
  - Метод. При вызове такого метода блокируется объект, для которого вызван данный метод

```
public void synchronized f() {
    ...
}
```

  - Участок кода.

```
synchronized(ref) {   // ref – ссылка на блокируемый объект
    ...
}
```

- Когда один из потоков входит в критический участок (synchronized метод или блок), связанный с блокировкой какого-то объекта, то остальные потоки не могут войти во все **критические участки**, связанные с блокировкой того же объекта, пока захвативший объект поток не выйдет из критического участка

# Пример синхронизации участков кода

```
public class MyObject{
    // В данном классе объявлены 2 синхронизированных метода f() и g()
    private String name;
    public MyObject(String name){ this.name = name; }

    public synchronized void f(){
      System.out.println("f() started for object" + name);
      try{
            Thread.sleep(1000);
      }catch(InterruptedException e){
            throw new RuntimeException(e);
      }
      System.out.println("f() finished for object" + name);
    }

    public synchronized void g(){
      System.out.println("g() started for object" + name);
      try{
            Thread.sleep(1000);
      }catch(InterruptedException e){
            throw new RuntimeException(e);
      }
      System.out.println("g() finished for object" + name);
    }
}
```

# Пример синхронизации участков кода

```
// Поток, в котором вызывается метод f()
public class Thread1 extends Thread{
      private MyObject myObj;
      public Thread1(MyObject myObj){ this.myObj = myObj; }
      public void run(){
        myObj.f();
      }
}

// Поток, в котором вызывается метод g()
public class Thread2 extends Thread{
      private MyObject myObj;
      public Thread2(MyObject myObj){ this.myObj = myObj; }
      public void run(){
        myObj.g();
      }
}

// Обратите внимание, что объект MyObject передается данным потокам в качестве
      параметра
```

# Пример синхронизации участков кода

- В данном потоке объявлена критическая секция, связанная с объектом MyObject

```
static class Thread3 extends Thread{
    private MyObject myObj;
    public Thread3(MyObject myObj){ this.myObj = myObj; }
    public void run(){
      synchronized(myObj){
            System.out.println("synchonized section started for object" +
            myObj.name);
            try{
                  Thread.sleep(1000);
            }catch(InterruptedException e){
                  throw new RuntimeException(e);
            }
            System.out.println("synchonized section finished for object" +
            myObj.name);
      }
}
```

# Пример синхронизации участков кода

```
public static void main(String[] args) {
    MyObject myObj = new MyObject("MyObject1");
    // Вызов потоков с синхронизацией по MyObject1
    Thread1 t1 = new Thread1(myObj);
    t1.start();
    Thread2 t2 = new Thread2(myObj);
    t2.start();
    Thread3 t3 = new Thread3(myObj);
    t3.start();

     // Вызов потока с синхронизацией по MyObject2
    MyObject myObj1 = new MyObject("MyObject2");
    Thread1 t4 = new Thread1(myObj1);
    t4.start();
}
```

# Пример синхронизации участков кода

- Результаты запуска программы

```
f() started for object MyObject1
f() started for object MyObject2
f() finished for object MyObject1
g() started for object MyObject1
f() finished for object MyObject2
g() finished for object MyObject1
synchonized section started for object MyObject1
synchonized section finished for object MyObject1
```

- Как видно, все синхронизированные участки кода по объекту MyObject1, выполнены последовательно.

- Участок, синхронизированный по MyObject2 – параллельно с остальными участками

# Синхронизация статических участков

- В качестве синхронизирующего объекта может выступать сам класс
- В этом случае последовательно выполняются все участки кода, синхронизированные по этому классу

```
static class MyClass{
    public synchronized static void f(){
      System.out.println("f() started");
      try{
            Thread.sleep(1000);
      }catch(InterruptedException e){
            throw new RuntimeException(e);
      }
      System.out.println("f() finished");
    }
}

static class ThreadS extends Thread{
    public void run(){
      MyClass.f();
    }
}
```

# Пример синхронизации статических участков

```
public static void main(String[] args) {

    ThreadS ts = new ThreadS();
    ts.start();

    synchronized(MyClass.class){
      System.out.println("We are in synchronized section");
    }
}
```

- Результат работы:
  f() started
  f() finished
  We are in synchronized section

- Как видно, участок, синхронизированный по MyClass.class ожидает, пока не выполнится метод f()

# Методы wait(), notify(), notifyAll()

- Эти методы применяются для обеспечения взаимодействия между потоками при работе с объектом
- Они определены в классе Object, а значит, могут быть вызваны для всех объектов
  - **wait()** – блокирует выполнение потока, из которого он вызван, и одновременно разблокирует объект, для которого он вызван
  - **notify()** - пробуждает один из потоков, которые вызвали wait() на том же самом мониторе
  - **notifyAll()** – пробуждает все потоки. Первым будет выполняться поток с самым высоким приоритетом
- Эти методы следует вызывать только когда объект служит монитором, т.е. из синхронизированного кода

# Применение wait () и notifyAll()

- Типичный код для перевода потока в режим ожидания:

```
synchronized void do() {
    while (условие ожидания)
      wait();
    // Сделать нечто, когда условие ожидания снимется
}
```

  - Оператор while нельзя заменять оператором if, поскольку поток может просыпаться от уведомлений, не связанных с условием его ожидания.

- Типичный код для вывода потоков из режима ожидания:

```
synchronized void changeCondition() {
    // Изменить условие ожидания
    notifyAll(); // Послать уведомление всем ожидающим потокам
}
```

# Взаимодействие потоков на примере

- **Задача.** Есть производитель и потребитель данных - целых чисел. Имеется склад данных, способный хранить одно число, принимать и выдавать число со склада. Необходимо так организовать работу системы, чтобы склад не переполнялся и чтобы потребитель не получал одно и то же число дважды
- Программа состоит из 3-х классов:
  1) Producer - поток-производитель данных;
  2) Consumer - поток-потребитель данных;
  3) Store - склад

```
class Producer implements Runnable {
    Store store; // Ссылка на объект-склад
    Producer(Store store) {   // Конструктор запускает поток-производитель
        this.store = store;
        (new Thread(this)).start();
    }
    public void run() {
        for (int i = 0; ; i++)
                store.put(i); // Производитель отсылает числа а склад
    }
}
```

# Взаимодействие потоков на примере

```
class Consumer implements Runnable {
    Store store; // Ссылка на объект-склад

    // Конструктор запускает поток-потребитель
    Consumer(Store store) {
      this.store = store;
      (new Thread(this)).start();
    }

    // Потребитель получает числа со склада
    public void run() {
      for (;;)
          store.get();
    }
}
```

# Взаимодействие потоков на примере

```
public class Store {
        private int n; // Хранимое на складе число
        private boolean hasData; // Признак наличия данных на складе
        synchronized int get() {
          while (!hasData) {            // Если склад пуст, поток пришедший за данными, должен ждать
                    try {
                         wait();
                    } catch (InterruptedException ex) {}
          }
           // Если число есть на складе, оно выдается
          hasData = false;
          notifyAll();
          System.out.println("Выдано со склада " + n);
          return n;
        }
        synchronized void put(int n) {
          while (hasData) {             // Если склад полон, поток принесший данные, должен ждать
                    try {
                              wait();
                    } catch (InterruptedException ex) {}
          }
           // Если склад пуст, число принимается на склад
          this.n = n;
          hasData = true;
          notifyAll();
          System.out.println("Поступило на склад " + n);
        }
}
```

# Взаимодействие потоков на примере

- В методе main() главный поток запускает два дочерних и тут же прекращается:

```
public static void main(String[] args) {
    Store store = new Store();
    Producer p = new Producer(store);
    Consumer c = new Consumer(store);
}
```

- В результате получим следующий вывод:

```
Поступило на склад 0
Выдано со склада 0
Поступило на склад 1
Выдано со склада 1
Поступило на склад 2
Выдано со склада 2
...
```

- Нетрудно убедиться, что если не вызывать методы wait() и notifyAll(), работа программы становится хаотической

# Взаимная блокировка потоков (deadlock)

- В простейшем случае взаимная блокировка возникает, когда два потока используют два синхронизированных объекта
- Пусть первый поток вводит монитор в объект A, а второй поток вводит монитор в объект B.
- После этого первый поток пытается ввести монитор в объект B, но переходит в ожидание, т.к. объект B занят вторым потоком. Второй поток не выполняется, он ждет освобождения объекта A.
- Получается, что оба потока не могут продолжать выполнение – возникает ситуация, которую называют DeadLock

# Пример взаимной блокировки потоков

```
public class DeadLocker extends Thread{
    Object a, b;
    public DeadLocker (Object a, Object b, String name) {
      super(name);
      this.a = a;
      this.b = b;
    }
    public void run() {
      // ввод монитора a
      synchronized (a) {
            System.out.println(getName() + ": step A");
            // передача процессора другому потоку
            sleep(0);
            // ввод монитора b
            synchronized (b) {
                  System.out.println(getName() + ": step B");
            };
      };
    }
}
```

# Пример взаимной блокировки потоков

- Для демонстрации взаимной влокировки используем следующий код:

```
// Объекты a и b, которые послужат мониторами
Object a = new Object(),
Object b = new Object();
// Первый поток получает объекты a и b
new DeadLocker(a, b, "First").start();
// Второй поток получает те же объекты в другом порядке
new DeadLocker (b, a, "Second").start();
```

- Этот код напечатает:

  First: step A

  Second: step A

  после чего потоки заблокируются

- Если вызов метода sleep(0) заменить на вызов метода a.wait(), взаимной блокировки не произойдет !!!

# Завершение потока

- Поток продолжает действовать (метод потока isAlive() возвращает **true**), пока не произойдет одно из 3-х событий
  - метод run() естественно завершится
  - работа метода run() будет прервана
  - будет вызван метод потока destroy()

- Вызов метода destroy() немедленно разрушает поток.
  - Все блокировки, сделанные потоком остаются, поэтому другие потоки могут легко попасть в состояние бесконечного ожидания.

- Многими операционными системами метод destroy() не поддерживается, и его применение вызывает исключение java.lang.NoSuchMethodError в вызывающем потоке.
  - Применять для завершения потока метод destroy() нельзя !!!

# Пример корректного завершения потока

- Корректного прекращения работы потока можно добиться, поместив в цикл метода run() **проверку флага завершения**:

```
public class MyThread extends Thread {
    // Флаг завершения и метод для его установки
    private boolean stopFlag = false;
    public void stopIt() {
       stopFlag = true;
    }
    // Поток, который печатает свое имя каждые полсекунды
    public void run() {
       while (!stopFlag) {
              System.out.println(getName());
              try {
                     sleep(500);
              }
              catch (InterruptedException ex) {}
       }
    }
}
```

- Такой подход не учитывает того, что в момент желаемой остановки поток  MyThread может выполнять длительный метод sleep() или wait()

# Прерывание потоков

- Механизм прерывания в классе Thread поддерживается методами:
  - interrupt() – устанавливает состояние потока в значение “прерван”
  - isInterrupted() – проверяет, прерван ли поток
  - interrupted() – то же самое, но еще и сбрасывает его

- **Замечание.** В Java 1 для приостановки, перезапуска и остановки потока использовали методы suspend(), resume() и stop().
  - Эти методы не снимают блокировки, установленные потоком, и поэтому **неприменимы !!!**

# Пример завершения потока с interrupt()

```
public class MyThread extends Thread{
    public void run() {
     while (!interrupted()) {
            System.out.println(getName());
            try {
                sleep(500);
            }
            catch (InterruptedException ex) {
                interrupt();
            }
     }
    }
}
```

- Повторный вызов метода interrupt() в блоке catch необходим потому, что метод sleep(), выбросив исключение, так же очищает состояние  потока "прерван", как и метод interrupted()

# Ожидание завершения потока

- Если главный поток решил дожидаться завершения работы дочернего потока, он может сделать это, организовав цикл с проверкой условия isAlive() для объекта дочернего потока.
  - Чтобы ожидание не было активным, в цикле можно выполнять метод sleep()

```
public static void main(String[] args) throws Exception{
    MyThread thread = new MyThread();
    thread.start();
    while (thread.isAlive())
     Thread.sleep(100);
    System.out.println("Конец");
}
```

- Другой вариант приостановить один поток до завершения другого – с использованием метода **join()**

# Модификатор volatile

- Обращение к любым данным, допускающим изменение несколькими потоками, должно выполняться в синхронизированном коде
- Если один поток пишет, а остальные только читают, синхронизация не обязательна, но возникает опасность чтения не из оригинала, а из копии переменной, которую может создать компилятор при оптимизации кода

**Пример.** Код, который периодически обновляет значение на экране:

```
currentValue = 5;
for (;;) {
    display.showValue(currentValue);
    Thread.sleep(1000);
}
```

  - Компилятор сочтет, что раз переменная не меняется внутри цикла, ее можно заменить константой, равной ее первоначальному значению.

- Чтобы избежать подмены оригинала копиями за счет оптимизации кода, код надо синхронизировать или объявить переменную как volatile.
  - volatile int currentValue;