СБЕРБАНК ТЕХНОЛОГИИ

MULTITHREADING
вводная лекция

- **Общие сведения и принципы функционирования**
- **Способы создания потоков**
- **Приоритезация потоков**
- **Daemon-потоки**

В следующей лекции: управление потоками,

организация взаимодействия потоков

ОПРЕДЕЛИМСЯ С ПОНЯТИЯМИ

СБЕРБАНК ТЕХНОЛОГИИ

- **Процессор / вычислитель**
- **Процесс**
- **Thread (Нить/Поток)**

СБЕРБАНК ТЕХНОЛОГИИ

Процесс

- совокупность кода и данных, выполняющихся в ОС
- изолированы друг от друга
- индивидуальное «виртуальное адресное пространство»

Поток

- принадлежит процессу
- имеет общее с ним и другими потоками «виртуальное адресное пространство»
- может быть порожден другими потоками, может сам порождать потоки

ПОСЛЕДОВАТЕЛЬНОЕ ВЫПОЛНЕНИЕ + / -

СБЕРБАНК ТЕХНОЛОГИИ

Плюсы:

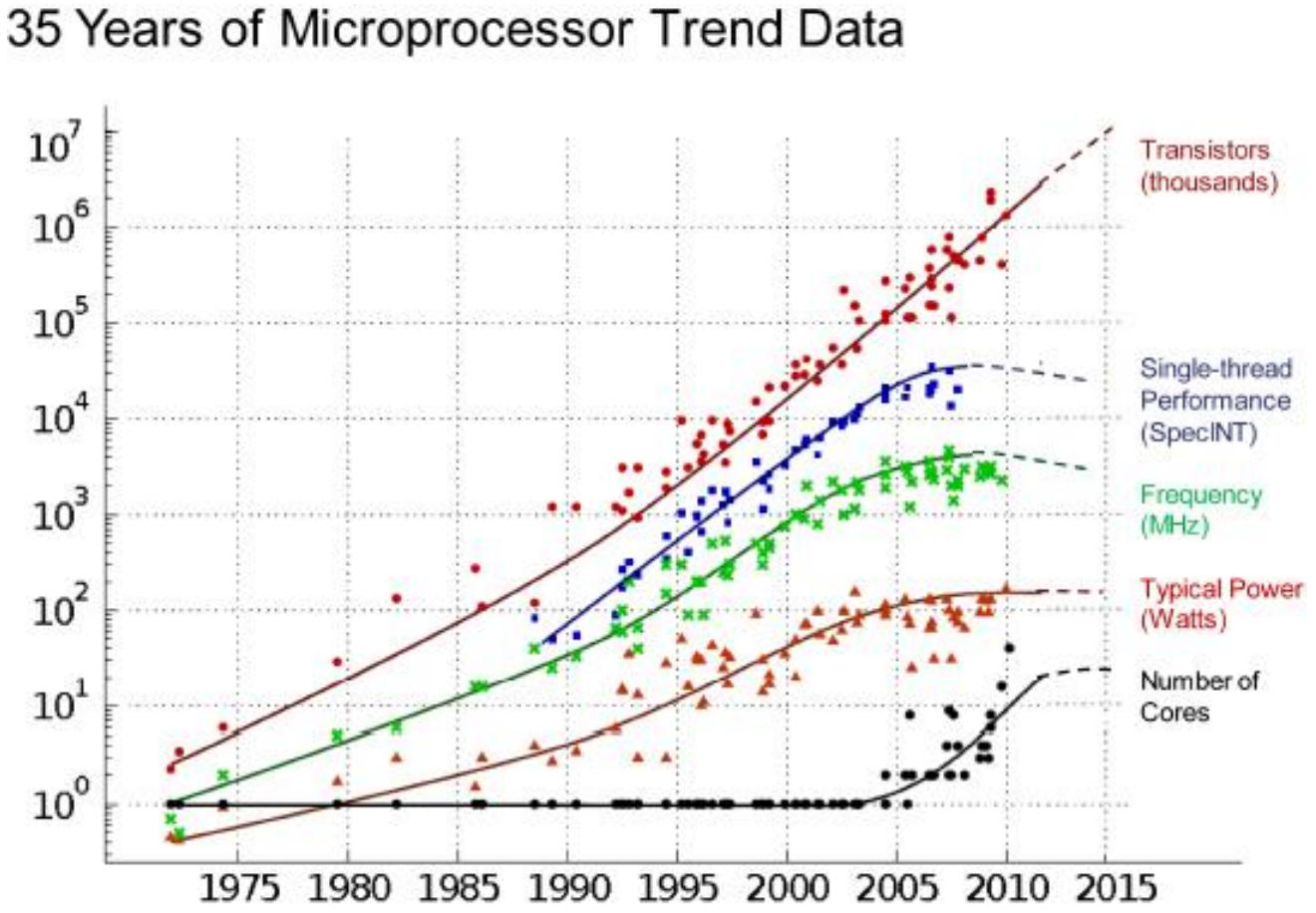
- Простота реализации
- Надежность
- Простота отладки

Минусы:

- Неприменимость в некоторых случаях (GUI, web-сервера, …)
- Ненужные простои (синхронное выполнение)
- Неоптимальное использование ресурсов системы

ТРЕНДЫ РАЗВИТИЯ МИКРОПРОЦЕССОРОВ

СБЕРБАНК ТЕХНОЛОГИИ



МНОГОПОТОЧНОЕ ВЫПОЛНЕНИЕ + / -

СБЕРБАНК ТЕХНОЛОГИИ

Плюсы:

- Улучшенная реакция приложения
- Эффективное использование ресурсов
- Повышение производительности

Минусы:

- Сложность разработки (высокий уровень входа, выше вероятность ошибки)
- Отладка затруднена
- Малоэффективна на однопроцессорных системах*
- Специфические проблемы (race condition, dead/live locks, …)

ЗАКОН АМДАЛА

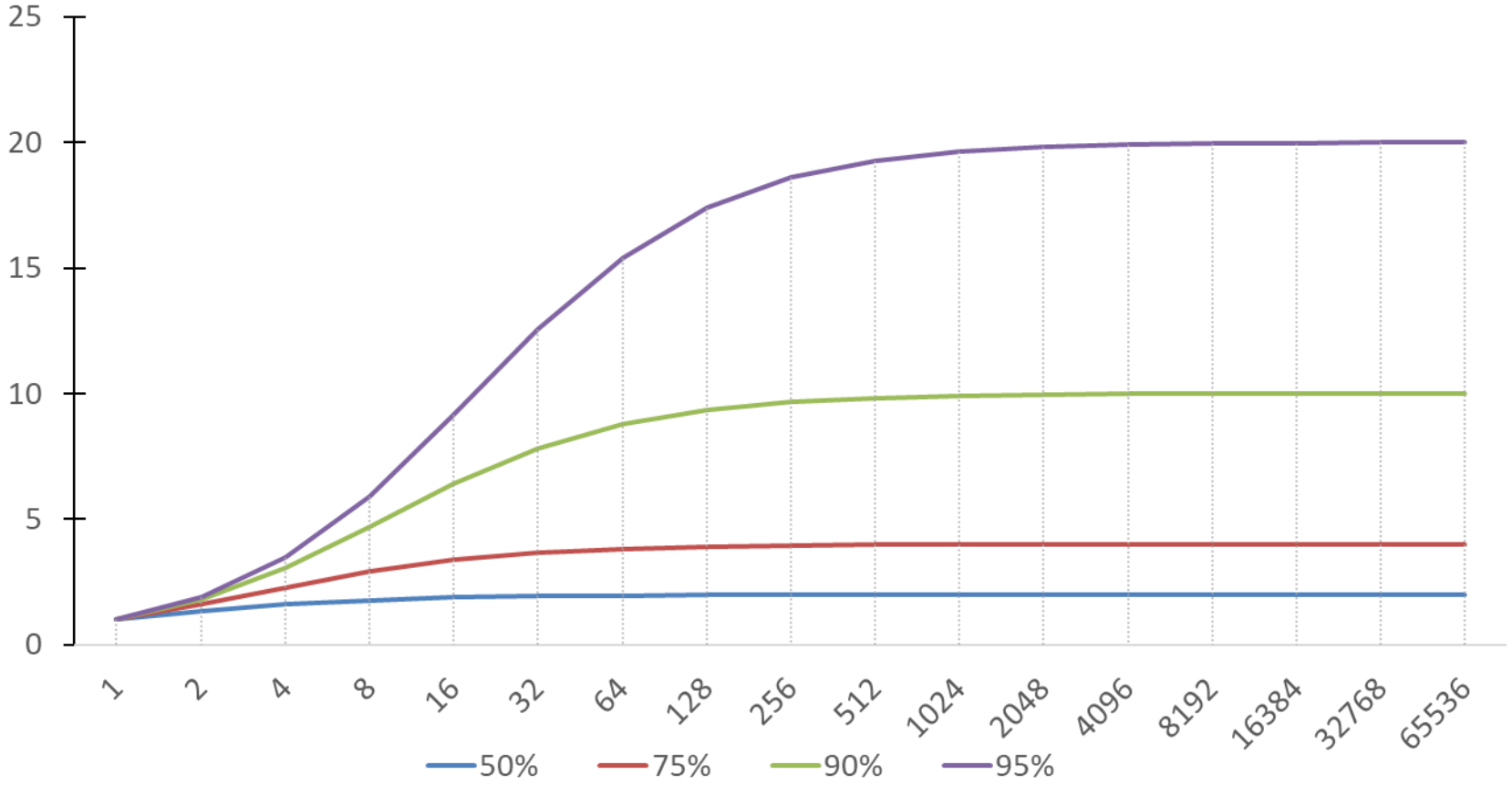
СБЕРБАНК ТЕХНОЛОГИИ

В случае, когда задача разделяется на несколько частей, суммарное время её выполнения на параллельной системе не может быть меньше времени выполнения самого длинного фрагмента.

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

a – доля последовательных вычислений
P – кол-во процессоров
S - ускорение

ЗАКОН АМДАЛА



СБЕРБАНК ТЕХНОЛОГИИ

Пересел на многопоточность...

Ожидание

Реальность

СБЕРБАНК ТЕХНОЛОГИИ

Neil J. Gunther's USL – Универсальный закон масштабируемости.

$$S = \frac{p}{1 + \alpha(p-1) + \beta p(p-1)}$$

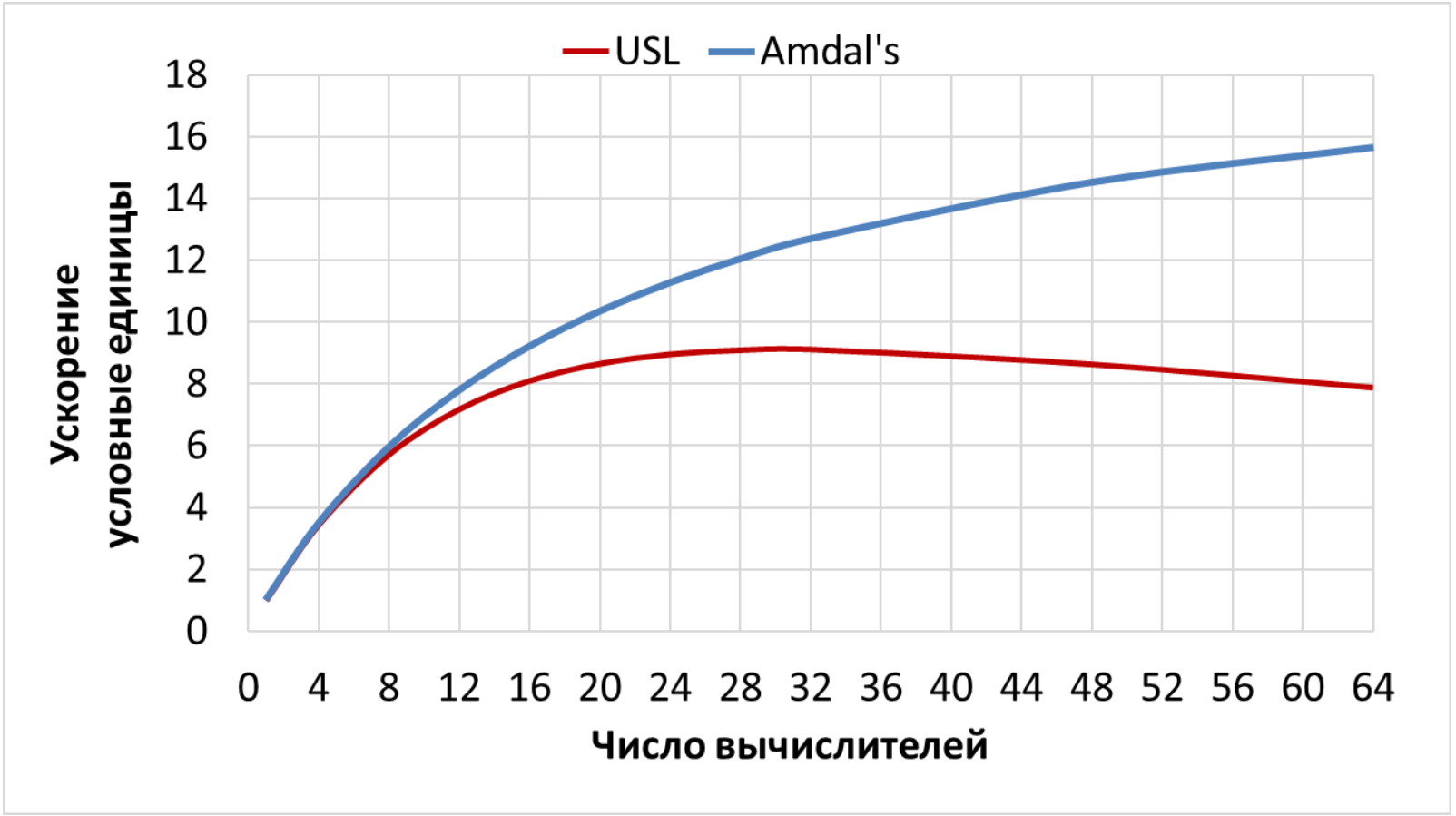
concurrency — contention — coherence penalty

p — ускорение конкретной части
α — оставшаяся часть
β — коэффициент

https://cdn2.hubspot.net/hubfs/498921/eBooks/scalability_new.pdf
http://www.perfdynamics.com/Manifesto/USLscalability.html

UNIVERSAL LAW OF COMPUTATIONAL SCALABILITY

СБЕРБАНК ТЕХНОЛОГИИ



ХВАТИТ ФОРМУЛ!

СБЕРБАНК ТЕХНОЛОГИИ

Java с первых версий имеет встроенную поддержку построения многопоточного *конкурентного* кода. Придерживается традиционного подхода – создание потоков в рамках одного процесса.

- **Задача (task)** – некоторая работа которая может быть выполнена.

- **Поток (thread)** – механизм который может выполнить задачу.

Определить неделимую задачу для выполнения можно с помощью реализации интерфейса Runnable:

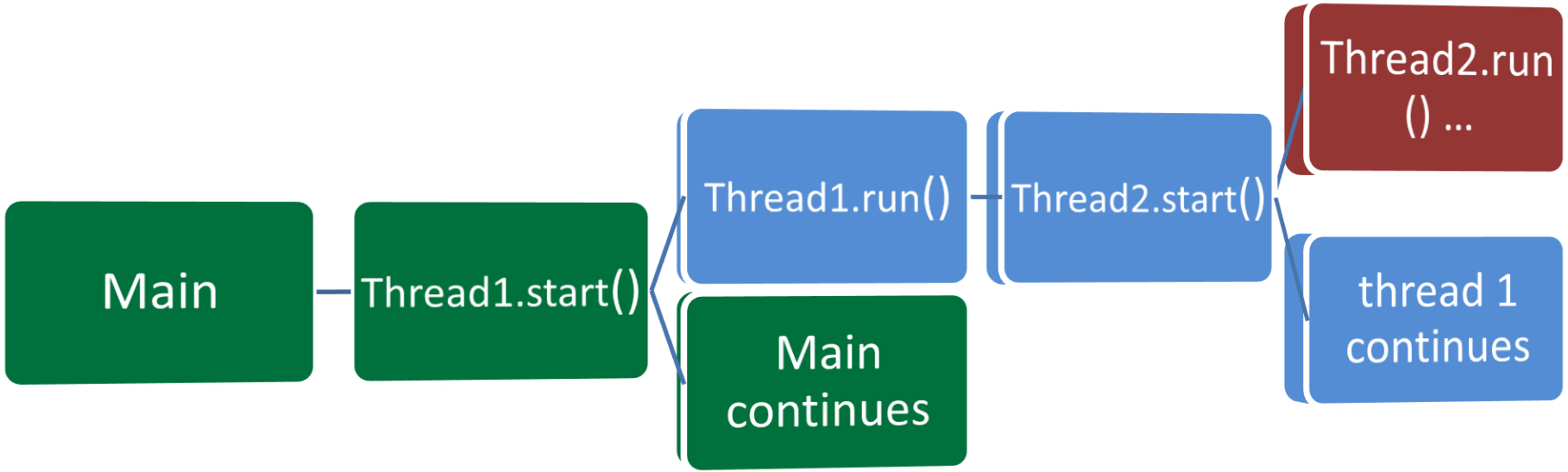
```
public class SomeTask implements Runnable {
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.print(format("#%d(%s)", i,
            Thread.currentThread().getName()));
        }
    }
}
```

Для запуска задачи в отдельном потоке можно использовать класс Thread:

```
public static void main(String[] args) {
    Thread t = new Thread(new SomeTask());
    t.start();
}
```

СОЗДАНИЕ ПОТОКА

СБЕРБАНК ТЕХНОЛОГИИ



Следовательно можно попробовать создать несколько параллельных потоков

```
public static void main(String[] args) {
    for (int i = 0; i < 10; ++i) {
        new Thread(new SomeTask()).start();
    }
    System.out.println("Waiting end of some task.");
}
```

Какой будет вывод программы?

ПОРЯДОК ЗАПУСКА. ОЖИДАНИЕ VS РЕАЛЬНОСТЬ

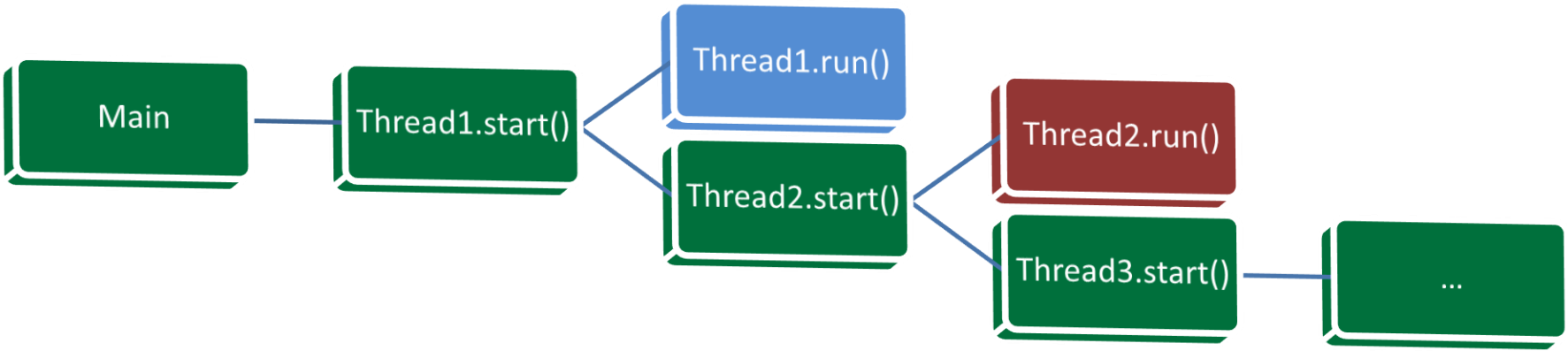
СБЕРБАНК ТЕХНОЛОГИИ

Вывод программы:

```
Waiting end of some task.
#0(Thread-4)#1(Thread-4)#2(Thread-4)#0(Thread-5)#0(Thread-6)#0(Thread-0)
#1(Thread-6)#0(Thread-8)#1(Thread-8)#2(Thread-8)#0(Thread-7)#1(Thread-7)
#2(Thread-7)#2(Thread-6)#0(Thread-9)#1(Thread-9)#2(Thread-9)#0(Thread-1)
#1(Thread-1)#2(Thread-1)#1(Thread-0)#2(Thread-0)#1(Thread-5)#0(Thread-2)
#2(Thread-5)#1(Thread-2)#2(Thread-2)#0(Thread-3)#1(Thread-3)#2(Thread-3)
Process finished with exit code 0
```

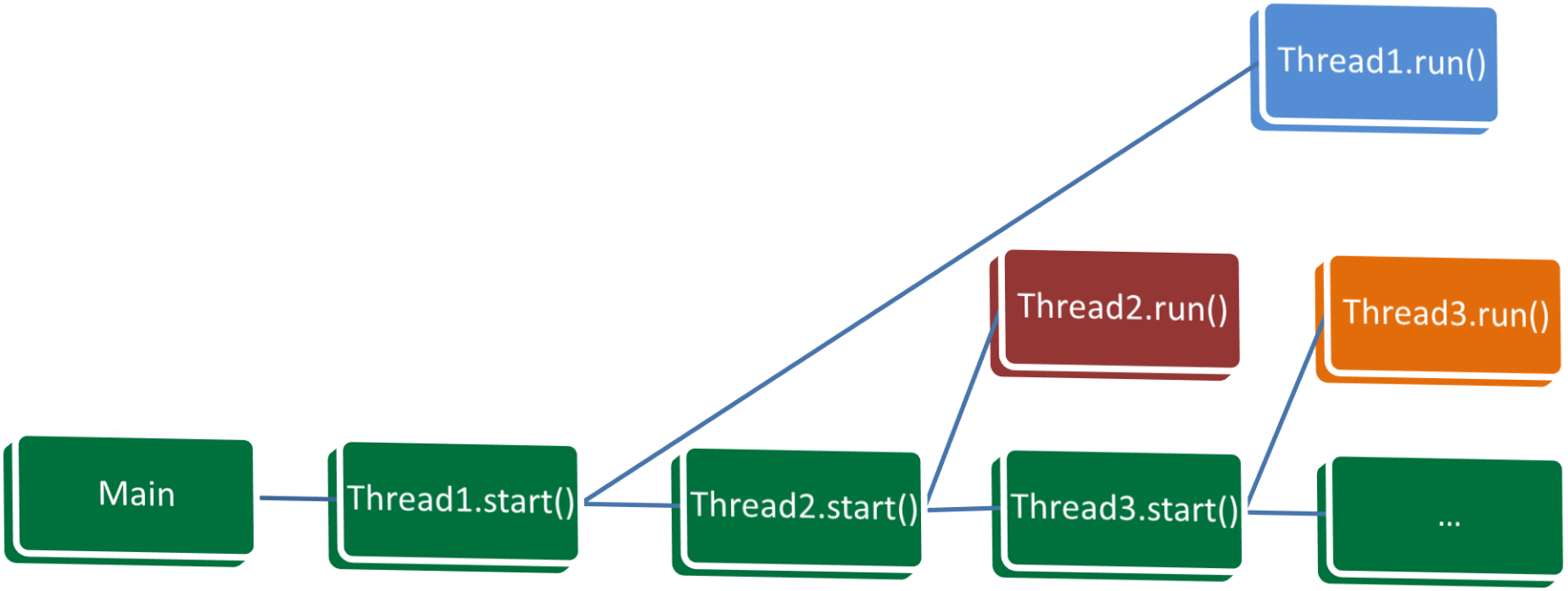
ПОРЯДОК ЗАПУСКА. ОЖИДАНИЕ

СБЕРБАНК ТЕХНОЛОГИИ



ПОРЯДОК ЗАПУСКА. РЕАЛЬНОСТЬ

СБЕРБАНК ТЕХНОЛОГИИ



ВЫВОД

СБЕРБАНК ТЕХНОЛОГИИ

- переключение между потоками контролируется **планировщиком ОС**
- на многопроцессорной машине планировщик распределит **потоки по процессорам**
- алгоритм планировщика **не детерминирован** – следовательно вывод предыдущей программы так же будет отличатся от запуска к запуску
- каждый объект типа **Thread** регистрирует себя в определённом месте и garbage collector не может удалить этот объект до тех пор пока не завершиться выполнение функции **run()**

СБЕРБАНК ТЕХНОЛОГИИ

Базовые способы создани и запуска потоков

- implements Runnable
- extends Thread

СБЕРБАНК ТЕХНОЛОГИИ

Реализовать Runnable:

```
public class RunnableImpl implements Runnable {
    @Override
    public void run() {
        // some task code here
    }

    public static void main(String[] args) {
        new Thread(new RunnableImpl()).start();
    }
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Унаследовать класс Thread:

```
public class SimpleThread extends Thread {

    @Override
    public void run() {
        //some task code here
    }

    public static void main(String[] args) {
        new SimpleThread().start();
    }
}
```

Через анонимный класс:

```
public class RunnableImpl {
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                //some task code here
            }
        }).start();
    }
}
```

Замечание:

Хорошей практикой является отделение потока от бизнес логики.

Т.е. класс Runnable не должен содержать бизнес логики, а должен быть просто инструментом для её запуска в отдельном потоке.

ВАРИАНТЫ СОЗДАНИЯ ПОТОКОВ

СБЕРБАНК ТЕХНОЛОГИИ

- Создавать потоки «руками» можно только в простых примерах и тестах
- Более правильным решением является использование ThreadPool, Executors и т.п. (пакет concurrent)
- Создавать и запускать поток в конструкторе нельзя – опасно

СБЕРБАНК ТЕХНОЛОГИИ

Какие бы вы выделили этапы жизненного цикла потоков?

СОСТОЯНИЯ ПОТОКА

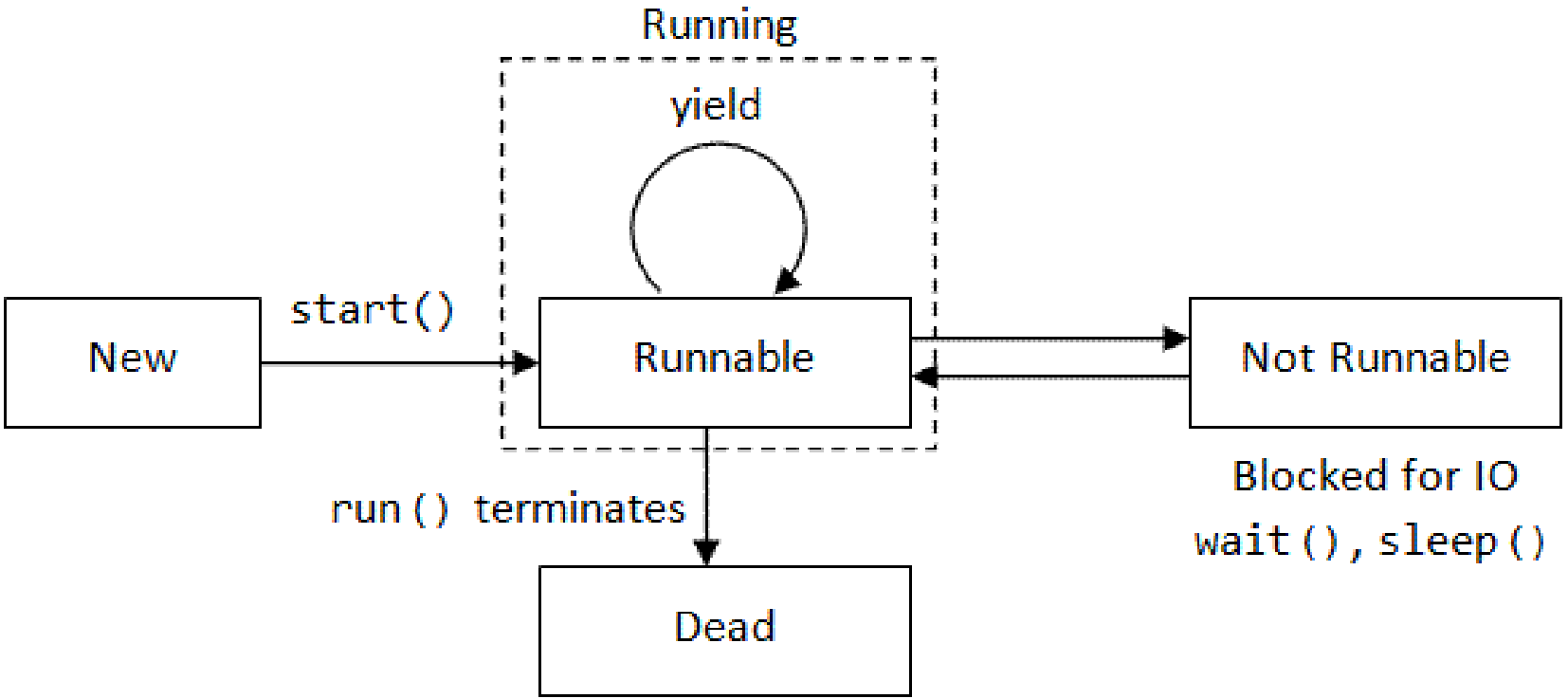
СБЕРБАНК ТЕХНОЛОГИИ

Любой поток может находится в одном из 4 логических состояний:

1. **NEW** – состояние в момент создания, когда система выделяет ему ресурсы, а планировщик готовит его к планированию
2. **RUNNABLE** – состояние, когда поток может выполнять полезную работу, планировщик его планирует и выделяет кванты времени CPU
3. **BLOCKED / WAITING / TIMED_WAITING** – поток блокирован, ожидает выполнение другого потока, ожидает по времени. Планировщик НЕ планирует его и не выделяет CPU, до возвращения в run
4. **TERMINATED (dead)** – планировщик не планирует, задача завершена и НЕ может больше выполняться

ЖИЗНЕННЫЙ ЦИКЛ ПОТОКА

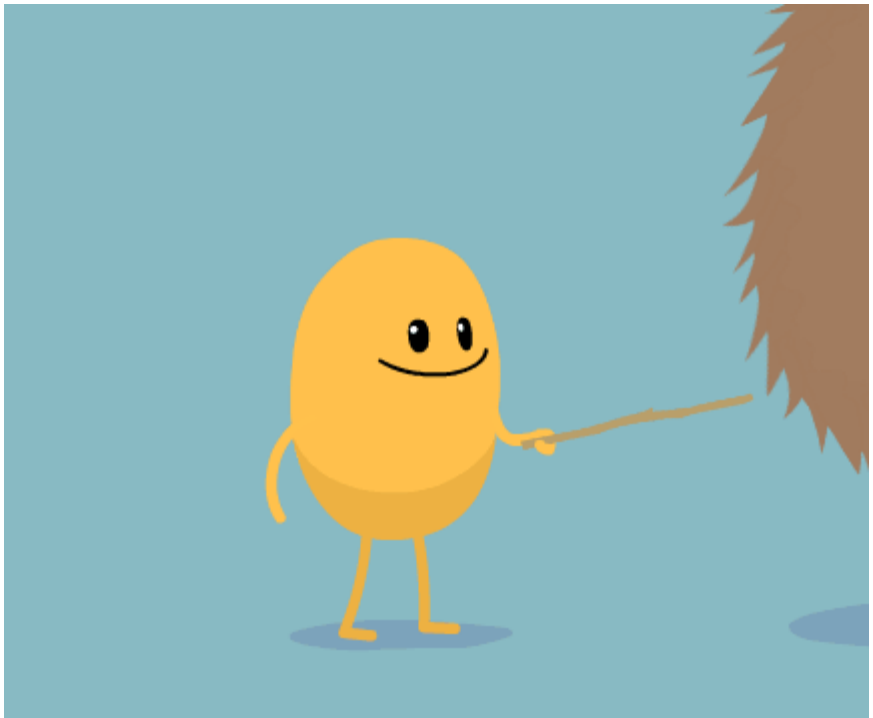
СБЕРБАНК ТЕХНОЛОГИИ



СБЕРБАНК ТЕХНОЛОГИИ

```
Thread t = Thread.currentThread()

t.getState()
```



Заставить поток «заснуть» на определённое время можно с помощью Thread.sleep():

```
Thread.sleep(1000);
```

В sleep состоянии поток перестаёт планироваться ОС.

СБЕРБАНК ТЕХНОЛОГИИ

Более удобная форма:

```
TimeUnit.SECONDS.sleep(5);
TimeUnit.MILLISECONDS.sleep(5);
TimeUnit.MICROSECONDS.sleep(5);
TimeUnit.MINUTES.sleep(5);
```

Есть возможность из потока дать «подсказку» планировщику о том что наш поток сделал достаточно и готов уступить квант времени другим потокам (используется редко).

```
public class Yield implements Runnable {
    public void run() {
        while(true) {
            Task t = getNewTask();
            t.execute();
            Thread.yield();
        }
    }
}
```

ПРИОРИТЕТЫ ПОТОКОВ - ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

Есть возможность задать приоритет потоку через метод setPriority.

```
public class PriorityTest implements Runnable{
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(format("#%s(%d)",
            Thread.currentThread().getName(),
            Thread.currentThread().getPriority()));
            Thread.yield();
        }
    }
}
```

СБЕРБАНК ТЕХНОЛОГИИ

Запустим:

```
public static void main(String[] args) {
    for (int i = 0; i < 4; i++) {
        Thread t = new Thread(new PriorityTest());
        t.setPriority(i % 2 == 0
            ? Thread.MAX_PRIORITY
            : Thread.MIN_PRIORITY);
        t.start();
    }
}
```

Какой будет вывод у программы?

ПРИОРИТЕТЫ ПОТОКОВ

СБЕРБАНК ТЕХНОЛОГИИ

В идеальном мире:

#Thread-2(10)
#Thread-2(10)
#Thread-0(10)
#Thread-0(10)
#Thread-0(10)
#Thread-2(10)
#Thread-1(1)
#Thread-3(1)
#Thread-1(1)
#Thread-3(1)
#Thread-1(1)
#Thread-3(1)

В реальном мире:

#Thread-1(1)
#Thread-3(1)
#Thread-0(10)
#Thread-2(10)
#Thread-1(1)
#Thread-0(10)
#Thread-2(10)
#Thread-3(1)
#Thread-1(1)
#Thread-2(10)
#Thread-0(10)
#Thread-3(1)

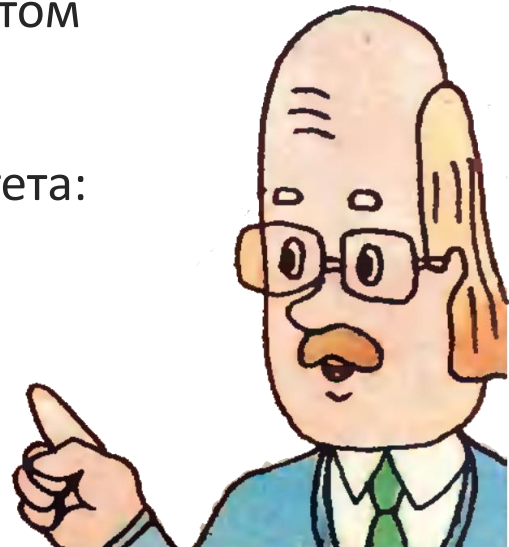
ПРИОРИТЕТЫ ПОТОКОВ

СБЕРБАНК ТЕХНОЛОГИИ

ПРИОРИТЕТЫ ПОТОКОВ

СБЕРБАНК ТЕХНОЛОГИИ

- Приоритет является мерой **важности потока для планировщика**.
- Планировщик будет **стараться** давать больший квант времени на исполнения потока с **более высоким приоритетом**.
- Задать приоритет можно с помощью метода **setPriority**().
- Так же это не значит что потоки с более низким приоритетом вообще не будут планироваться.
- Обычно стараются использовать 3 из 10 уровней приоритета: **MAX_PRIORITY**, **NORM_PRIORITY** и **MIN_PRIORITY**
- Обычно манипулирование приоритетом потока является **ошибочной практикой**.



ПОТОКИ ДЕМОНЫ

СБЕРБАНК ТЕХНОЛОГИИ

Daemon threads предназначены для выполнения минорных задач до тех пор пока программа не закончит выполнение.

Программа закончит выполнение только тогда когда все **НЕ демон** потоки завершат свою работу. Таким образом, завершая выполнение, программа убивает все демон потоки и завершается.

ПОТОКИ ДЕМОНЫ - ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

```
public class DaemonExample implements Runnable {
    public void run() {
        try {
            // invoke long task logic (> one second)
        } finally {
            System.out.println("Thread has done.");
        }
    }
}
```

ПОТОКИ ДЕМОНЫ - ПРИМЕР

СБЕРБАНК ТЕХНОЛОГИИ

```
public static void main(String[] args)
      throws InterruptedException {
    Thread t = new Thread(new DaemonExample());
    t.setDaemon(true);
    t.start();
    System.out.println("Daemon thread has been started.");
}
```

Что выведет программа?

Daemon thread has been started.

Process finished with exit code 0

Код из finally блока не вызвался.

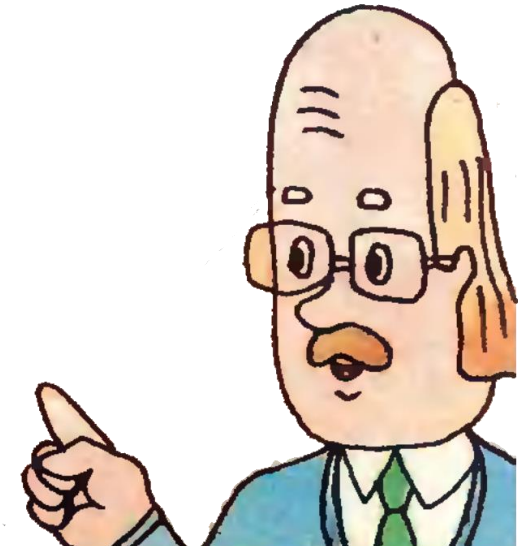
ПОТОКИ ДЕМОНЫ

СБЕРБАНК ТЕХНОЛОГИИ

- чтобы поток сделать демоном, нужно на нём перед стартом вызвать метод setDaemon(**true**)
- **finally** блок не будет вызван, т.к. поток прерывается «грубо», без освобождения занимаемых ресурсов

Так же:

- узнать является ли поток демоном можно вызвав метод isDaemon() на потоке



ОЖИДАНИЕ ЗАВЕРШЕНИЯ ПОТОКА

СБЕРБАНК ТЕХНОЛОГИИ

Любой поток может дождаться завершения работы другого потока с помощью метода класса Thread - join().

```
Thread t = new Thread(new JoinTask());
t.start();
t.join();
```

ОЖИДАНИЕ ЗАВЕРШЕНИЯ ПОТОКА

СБЕРБАНК ТЕХНОЛОГИИ

- ожидающий поток **блокируется** и не планируется пока ожидаемый не завершит свою работу
- **join** может быть вызван с аргументом задающим кол-во мс которое необходимо ожидать
- состояние потока можно проверить с помощью метода **isAlive()**

СБЕРБАНК ТЕХНОЛОГИИ

Какой вывод будет у программы?

```
public class ThrowExceptionSimpleCase implements
Runnable {
    public void run() {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        new Thread(new ThrowExceptionSimpleCase())
        .start();
    }
}
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ В ПОТОКЕ

СБЕРБАНК ТЕХНОЛОГИИ

```
Exception in thread "Thread-0" java.lang.RuntimeException
        at ThrowExceptionSimpleCase.run(ThrowExceptionSimpleCase.java:3)
        at java.lang.Thread.run(Thread.java:745)

Process finished with exit code 0
```

Если исключение будет выброшено из функции run(), оно запишется на консоль, поток завершит работу.

СБЕРБАНК ТЕХНОЛОГИИ

Попробуем поймать исключение:

```
public static void main(String[] args) {
   try {
      new Thread(new ThrowExceptionSimpleCase())
.start();
   } catch (RuntimeException ex) {
      System.out.println("Exception has been catch");
   }
}
```

ОБРАБОТКА ИСКЛЮЧЕНИЙ В ПОТОКЕ

СБЕРБАНК ТЕХНОЛОГИИ

Нельзя перехватить исключение из контекста другого потока.

```
Exception in thread "Thread-0" java.lang.RuntimeException
        at ThrowExceptionSimpleCase.run(ThrowExceptionSimpleCase.java:3)
        at java.lang.Thread.run(Thread.java:745)
Process finished with exit code 0
```

Метод **Thread.UncaughtExceptionHandler()** позволяет задавать обработчик uncaught (не пойманных) исключений потоку.

```
public static void main(String[] args) {
    Thread t = new Thread(new
ThrowExceptionSimpleCase());
    t.setUncaughtExceptionHandler((t1, e) ->
            System.out.printf("Exception %s has been
catch from thread %s",
      e, t1.getName()));
}
```

Вывод программы:

```
Exception java.lang.RuntimeException has been catch from thread Thread-0
Process finished with exit code 0
```

Метод **Thread.setDefaultUncaughtExceptionHandler()** позволяет задавать обработчик uncaught исключений всем потокам по-умолчанию.

Следующие методы класса Thread устарели и являются опасными, т.к. не освобождают занятые ими ресурсы.

```
public final void stop()
public final synchronized void stop(Throwable obj)
public void destroy()
public final void suspend()
public final void resume()
```

Подробнее:

https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html

Цитата про функциональность группы потоков:

"Thread groups are best viewed as an unsuccessful experiment, and you may simply ignore their existence."

Joshua Bloch

СПИСОК ЛИТЕРАТУРЫ

СБЕРБАНК ТЕХНОЛОГИИ

- “Thinking in Java” Bruce Eckel
- “Java Concurrency In Practice”
- “Java SE 8 Programmer II Study Guide” by Jeanne Boyarsky & Scott Selikoff
- “SCJP Sun Certified Programmer for Java 7 Study Guide”