Chapter 8. 多线程 `Multithreading`

> Java is a multi-threaded programming language

1. Process and thread
   - A process is an execution of a program and a thread is a single execution of work within the process.
   - A process can contain multiple threads.
   - A thread is also known as a lightweight process.
2. Thread Lifecycle
   - 新建 `New`
   - 就绪 `Runnable`
   - 运行 `Running`
   - 阻塞 `Blocked`
     - 等待阻塞 `Waiting`
     - 同步阻塞 `Blocked on synchronization`
     - 其他阻塞
       - I/O
       - Sleep
       - Join
   - 死亡 `Dead`

3. Implementation
   - Extend `Thread` class

```java
public class MT1 extends Thread {

    public static void main(String[] args) {
        MT1 mt1 = new MT1();
        mt1.start();
        System.out.println("test...");
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}
```

- Implement `Runnable` interface

```
public class MT2 implements Runnable {
    public static void main(String[] args) {
        MT2 mt2 = new MT2();
        Thread thread = new Thread(mt2);
        thread.start();
        System.out.println("test...");
    }

    @Override
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}
```

4. `join`

> Waits for this thread to die.

```
public class MT3 implements Runnable {

    public static void main(String[] args) {

        Thread thread = new Thread(new MT3());
        thread.setName("thread");
        thread.start();

        try {
            thread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("test...");
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName() + " i
s running...");
            try {
                Thread.sleep(1000 * 3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
```

```
    }
}
```

```
public class MT4 implements Runnable {

    public static void main(String[] args) {

        Thread thread1 = new Thread(new MT4());
        thread1.setName("thread 1");
        Thread thread2 = new Thread(new MT4());
        thread2.setName("thread 2");
        Thread thread3 = new Thread(new MT4());
        thread3.setName("thread 3");

        thread1.start();
        thread2.start();

        try {
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        thread3.start();

        System.out.println("test...");
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName() + " i
s running...");
            try {
                Thread.sleep(1000 * 1);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

5. `Thread.yield` `[ji:ld]`

> A hint to the scheduler that the current thread is willing to yield its current use of a processor. The scheduler is free to ignore this hint.

```
public class MT5 implements Runnable {

    public static void main(String[] args) {
        Thread thread1 = new Thread(new MT5());
        thread1.setName("thread 1");
        Thread thread2 = new Thread(new MT5());
        thread2.setName("thread 2");

        thread1.start();
        thread2.start();

        System.out.println("test...");
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            System.out.println(i + ": " + Thread.currentThread().getN
ame() + " is running...");
            if (i % 10 == 0) {
                Thread.yield();
            }
        }
    }
}
```

6. `setPriority`

> Java Thread priority has no effect

- `MIN_PRIORITY` 1
- `MAX_PRIORITY` 10
- `NORMAL_PRIORITY` 5

```
public class MT6 implements Runnable {
    @Override
    public void run() {
        Thread thread = Thread.currentThread();
        System.out.println(thread.getName() + ", " + thread.getPriori
ty());
    }

    public static void main(String[] args) {
        Thread thread1 = new Thread(new MT6(), "thread1");
        Thread thread2 = new Thread(new MT6(), "thread2");
        Thread thread3 = new Thread(new MT6(), "thread3");

        thread1.setPriority(Thread.MIN_PRIORITY);
        thread2.setPriority(Thread.NORM_PRIORITY);
```

```
        thread3.setPriority(Thread.MAX_PRIORITY);

        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

7. Synchronization
    - synchronization method
    - synchronization block
    - `synchronized` 作用域
        - 对象 / 实例 范围

            > 一个线程只能访问一个对象的 synchronized 方法，但其他线程可以访问另一个对象的同一方法

        - 类范围

            > 一个线程只能访问一个类的一个 synchronized static 方法，对这个类的所有对象都适用

```
public class Synchronization {
    public static void main(String[] args) {
        Food water = new Food("water");
//        Food fish = new Food("fish");
//        Food bone = new Food("bone");

        Cat cat = new Cat("kitty", water);
        Dog dog = new Dog("tiger", water);

        cat.start();
        dog.start();
    }
}

class Food {
    private String name;

    Food(String name) {
        this.name = name;
    }

    synchronized void eat1() {
        System.out.println(Thread.currentThread().getName() + " is ea
ting " + name);
        try {
            Thread.sleep(1000 * 5);
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
    }

    void eat2() {
        synchronized (this) {
            System.out.println(Thread.currentThread().getName() + " i
s eating " + name);
            try {
                Thread.sleep(1000 * 5);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }

    synchronized static void eat3() {
        System.out.println(Thread.currentThread().getName() + " is ea
ting...");
        try {
            Thread.sleep(1000 * 5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Cat extends Thread {
    private Food food;

    Cat(String name, Food food) {
        super(name);
        this.food = food;
    }

    @Override
    public void run() {
        food.eat1();
//        food.eat2();
//        food.eat3();
    }
}

class Dog extends Thread {
    private Food food;

    Dog(String name, Food food) {
        super(name);
        this.food = food;
    }

    @Override
```

```
    public void run() {
        food.eat1();
//        food.eat2();
//        food.eat3();
    }
}
```

8. `wait` `notify` `notifyAll`

> 来自 `Object` 类，线程间通讯的方式

```
public class OutputThread implements Runnable {
    private int num;
    private final Object lock;

    private OutputThread(int num, Object lock) {
        this.num = num;
        this.lock = lock;
    }

    public void run() {
        try {
            while (true) {
                synchronized (lock) {
                    System.out.println(num);
                    lock.notify();
                    lock.wait();
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        final Object lock = new Object();

        Thread thread1 = new Thread(new OutputThread(1, lock));
        Thread thread2 = new Thread(new OutputThread(2, lock));

        thread1.start();
        thread2.start();
    }
}
```

```
public class ObjectTest {
public static void main(String[] args) throws InterruptedException {
    Object o = new Object();
    Object lock = new Object();
```

```
//        o.wait();
//        o.notify();

//        synchronized (lock) {
//            o.notify();
//            System.out.println("test...");
//            o.wait();
//        }
    }
}
```

- `wait`

  ```
  public final void wait() throws InterruptedException
  ```

  - 调用后，当前线程释放锁
  - 当前线程阻塞
  - 等待其他线程调用 `notify` 或 `notifyAll` 唤醒
  - 被唤醒后，重新竞争锁
  - 调用 `wait` 时，线程必须获得对象级别锁，即，在同步方法或同步块中
  - 如果调用时没有锁，抛 `IllegalMonitorStateException` 运行时异常

- `notify`

  ```
  public final native void notify()
  ```

  - 唤醒正在 wait 的线程，如有多个，挑选**一个**
  - 通知后，当前线程不会马上释放锁，wait 线程不会马上获得锁，需要等待当前线程退出同步区
  - 被唤醒的线程获得锁并执行完成，如果没有继续 notify，其他 wait 线程继续阻塞，等待被唤醒
  - 调用 `notify` 时，线程必须获得对象级别锁，即，在同步方法或同步块中
  - 如果调用时没有锁，抛 `IllegalMonitorStateException` 运行时异常

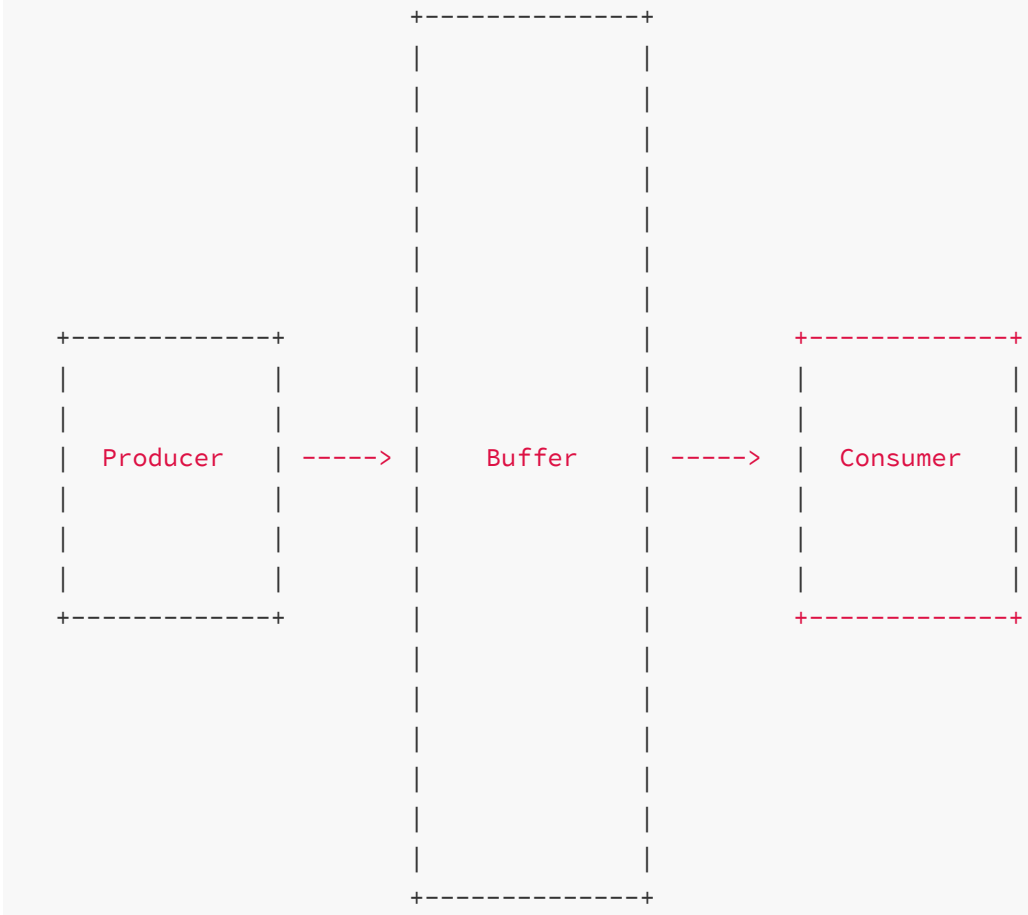
- `notifyAll`

  ```
  public final native void notifyAll()
  ```

  - 唤醒**全部**正在 wait 的线程
  - 当前线程退出同步区时，所有被唤醒线程竞争锁
  - 得到锁的线程执行完成同步区，其他线程继续竞争锁，直到全部执行完成
  - 调用 `notifyAll` 时，线程必须获得对象级别锁，即，在同步方法或同步块中
  - 如果调用时没有锁，抛 `IllegalMonitorStateException` 运行时异常

9. 生产者与消费者问题

> Producer-consumer problem
>
> 保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。


Producer
Buffer
Consumer