姓名:张煜；学号2208300002:

# 线程

## 需求分析：

什么是线程？

答：

线程是进程中的一个执行单元，负责当前进程中程序的执行，一个进程中至少 有一个线程，一个进程中也可以有多个线程，此时这个应用程序就可以称之为多线程程序。

# 时间片

## 需求分析：

CPU中的时间片，有什么作用？

答：

1. **公平调度**：通过为每个线程分配相同的时间片，可以确保所有活动的线程都有机会获得CPU时间。这有助于实现公平性，防止某些线程长时间占用CPU资源而其他线程得不到执行的机会。
2. **并发执行**：尽管物理CPU核心在同一时刻只能执行一个线程，但是通过快速切换线程并在它们之间分配时间片，操作系统可以让用户感觉多个线程似乎同时在运行。这就是所谓的并发执行。
3. **响应性**：时间片机制使得系统能够对新的请求或事件做出及时响应。如果某个线程在其时间片内没有完成执行，它将被暂停，并且CPU将被分配给另一个等待的线程。这样可以避免单个线程占用CPU过长时间而导致系统响应迟缓。

# 进程和线程

## 需求分析：

进程和线程的关系是什么？

答：

**进程与线程的关系**

* **包含关系**：一个进程可以拥有一个或多个线程；而一个线程一定属于一个进程。
* **资源拥有方式**：进程之间有独立的地址空间和其他资源；而同一个进程内的线程共享这些资源。
* **切换开销**：由于进程间的隔离性，从一个进程切换到另一个进程需要较大的开销；线程间的切换开销较小，因为它们共享相同的上下文。
* **创建和销毁**：创建一个新的进程涉及更多资源的分配，因而比创建一个新线程更耗时；线程的创建和销毁相对简单快速。
* **并行执行能力**：在多处理器或多核心系统中，来自不同进程的不同线程可以真正并行执行（即同时执行），而同一个进程内的线程只能并发执行（即看起来同时执行，实际上是快速切换执行）。

# 线程分类

## 需求分析：

线程都有哪些分类？

答：

1. **按状态分类**：
   * **NEW**：线程新建但尚未开始执行。
   * **RUNNABLE**：线程正在运行或者准备好运行。
   * **BLOCKED**：线程正在等待监视器锁。
   * **WAITING**：线程处于等待状态，等待另一个线程执行特定的动作，比如调用Object.notify()或者Thread.join()。
   * **TIMED\_WAITING**：线程处于等待状态，并且有一个指定的等待时间。
   * **TERMINATED**：线程已经终止。
2. **按角色分类**：
   * **普通线程**：执行应用程序的任务。
   * **守护线程（Daemon Thread）**：提供系统服务，如垃圾回收。当仅剩下守护线程时，Java虚拟机将会退出。
3. **按执行方式分类**：
   * **并发执行（Concurrent Execution）**：多个线程在一个时间段内交替执行。在一个单核CPU上，线程会交替使用CPU，而在多核CPU上，多个线程可能同时在不同的核心上执行。
   * **并行执行（Parallel Execution）**：多个线程在同一时刻执行。在多核或多处理器环境下，多个线程可以同时运行于不同的CPU核心上。

# 并发和并行

## 需求分析：

描述什么是并发与并行？

答：

* 并发：指两个或多个事件在同一个时间段内发生
  + 线程的并发执行，是指在一个时间段内（微观），俩个或多个线程，使用同

一个CPU交替运行。

* 并行：指两个或多个事件在同一时刻发生（同时发生）
  + 线程的并行执行，是指在同一时刻，俩个或多个线程，各自使用一个CPU同 时运行。

# 多线程

## 需求分析：

多线程的好处是什么？

答：

1. **资源利用率提升**：多线程可以更好地利用多核处理器的计算能力。在多核架构中，多个线程可以同时执行，从而提高硬件资源的利用率。
2. **响应速度提高**：多线程可以改善程序的响应性，特别是对于需要处理大量输入输出（I/O）操作的应用程序来说。一个线程在等待I/O操作完成时，不会阻止其他线程继续执行，这样可以避免整个程序的停滞不前。
3. **并发执行**：多线程允许程序在宏观上并发执行多个任务，使得应用程序可以在等待一个任务完成的同时执行另一个任务。例如，一个线程可以负责下载数据，而另一个线程则可以处理用户界面的更新。
4. **可管理性增强**：多线程编程可以使复杂的任务分解成多个独立的小任务，每个任务可以由单独的线程来处理，这样可以使代码更易于理解和维护。
5. **任务划分**：对于一些需要处理大量数据或进行复杂运算的应用来说，将任务分割成多个部分，每个部分由不同的线程处理，可以显著加快处理速度。
6. **提高吞吐量**：吞吐量是指单位时间内完成的工作量，多线程可以增加系统的吞吐量，尤其是在处理密集型任务时，通过并行处理可以显著提高处理速度。
7. **提升任务效率**：多线程可以使得用户交互更加流畅，即使后台正在进行大量的计算，也不会影响到前台的响应速度，从而提升任务效率

# 线程启动

## 需求分析：

请描述start方法和run方法的区别？

答：

1. **启动线程**：
   * start()方法是真正用来启动一个新线程的方法。当调用一个Thread对象的start()方法时，Java虚拟机会创建一个新的线程，并调用该线程的run()方法。
   * 直接调用run()方法并不会启动一个新的线程，而只是作为当前线程的一个普通方法来执行。
2. **执行线程任务**：
   * run()方法包含了线程要执行的具体任务逻辑。当一个线程被启动后，它的run()方法会被调用，从而开始执行线程的任务。
   * 如果直接调用run()方法，则它只是当前线程的一个方法调用，不会导致新的线程被创建或执行。
3. **线程状态**：
   * 调用start()方法会使线程进入可运行状态（RUNNABLE），此时线程可以被调度来执行。
   * 直接调用run()方法不会改变线程的状态，它只是作为当前线程的函数调用。
4. **并发执行**：
   * 当调用start()方法后，run()方法将在一个新的线程中执行，可以与其他线程并发执行。
   * 直接调用run()方法的结果是在当前线程中执行run()方法体中的代码，与调用其他普通方法无异。

# 程序分析

## 需求分析：

分析代码，程序的输出结果是什么

public class Test08 {

public static void main(String[] args) { Thread t = new Thread() {

public void run() {

String name = Thread.currentThread().getName(); System.out.println(name);

}

};

t.start();

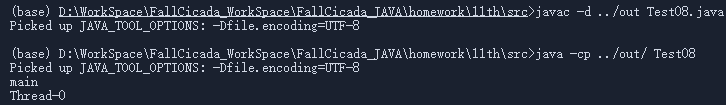
t.run();

}

}

答：

1. **创建线程对象**：创建了一个匿名内部类的线程对象t，该对象重写了run()方法，用于打印当前线程的名字。
2. **启动线程**：调用t.start()启动了线程t，这会导致在新线程中执行t.run()。打印新启动的线程签名---🡪 “Thread-0”
3. **直接调用run()**：紧接着，在主线程中直接调用了t.run()。打印主线程签名--🡪“main”

运行结果：  


# 打印奇偶数

## 需求分析：

请编写一个Java程序，创建两个线程，一个打印50以内的奇数，另一个打印50以内的偶数

答：

代码：

**/\*\***

**\* 请编写一个Java程序，创建两个线程，一个打印50以内的奇数，另一个打印50以内的偶数**

**\*/**

public class Test9 {

    public static void main(String[] args) {

        Thread oddThread = new Thread(new OddThread());

        Thread evenThread = new Thread(new EvenThread());

        oddThread.start();

        evenThread.start();

    }

}

class OddThread implements Runnable {

    @Override

    public void run() {

        for (int i = 1; i < 50; i += 2) {

            System.out.println("Odd: " + i);

        }

    }

}

class EvenThread implements Runnable {

    @Override

    public void run() {

        for (int i = 0; i < 50; i += 2) {

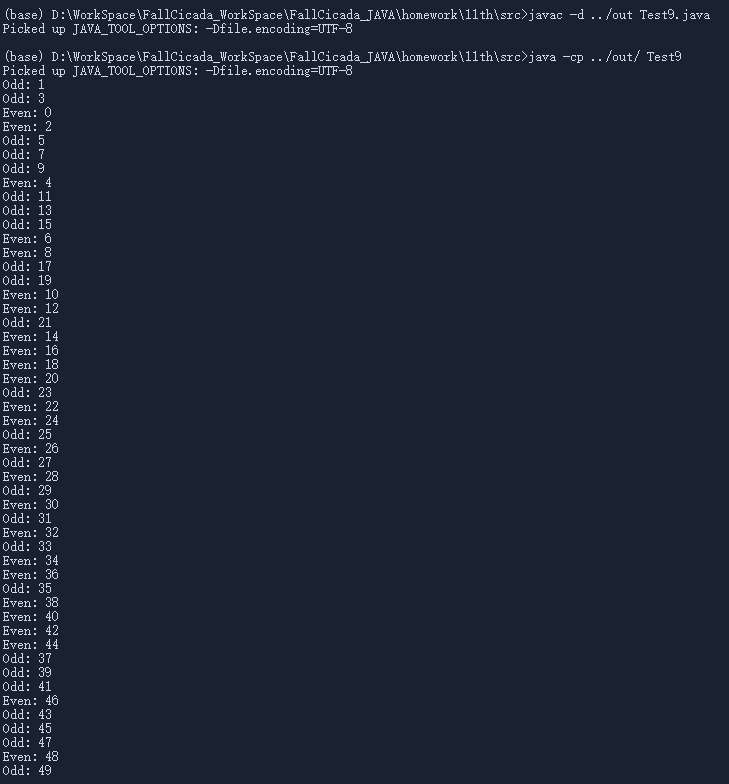
            System.out.println("Even: " + i);

        }

    }

}

运行结果：



# 线程优先级

## 需求分析：

编写程序，实现创建3个线程。

* 1. 分别输出1~10之间的数字及对应的输出数字线程名
  2. 线程优先级分别为最高、普通、最低
  3. 每个线程输出一次后，休眠一秒钟
  4. 思考是否每次都是优先级最高的线程先打印输出？

答：

代码：

public class Test10 {

    public static void main(String[] args) {

**// 创建高优先级线程，并命名为 "HighPriorityThread"**

        Thread highPriorityThread = new Thread(new HighPriorityThread(), "HighPriorityThread");

**// 创建普通优先级线程，并命名为 "NormalPriorityThread"**

        Thread normalPriorityThread = new Thread(new NormalPriorityThread(), "NormalPriorityThread");

**// 创建低优先级线程，并命名为 "LowPriorityThread"**

        Thread lowPriorityThread = new Thread(new LowPriorityThread(), "LowPriorityThread");

**// 设置高优先级线程的优先级为最大值**

        highPriorityThread.setPriority(Thread.MAX\_PRIORITY);

**// 设置普通优先级线程的优先级为默认值**

        normalPriorityThread.setPriority(Thread.NORM\_PRIORITY);

**// 设置低优先级线程的优先级为最小值**

        lowPriorityThread.setPriority(Thread.MIN\_PRIORITY);

**// 输出线程的优先级**

        System.out.println(highPriorityThread.getName() + " Priority: " + highPriorityThread.getPriority());

        System.out.println(normalPriorityThread.getName() + " Priority: " + normalPriorityThread.getPriority());

        System.out.println(lowPriorityThread.getName() + " Priority: " + lowPriorityThread.getPriority());

**// 启动高优先级线程**

        highPriorityThread.start();

**// 启动普通优先级线程**

        normalPriorityThread.start();

**// 启动低优先级线程**

        lowPriorityThread.start();

    }

}

**// 高优先级线程类，实现 Runnable 接口**

class HighPriorityThread implements Runnable {

    @Override

    public void run() {

**// 循环输出1到10之间的数字及线程名**

        for (int i = 1; i <= 10; i++) {

            System.out.println(Thread.currentThread().getName() + ": " + i);

            try {

**// 每次输出后休眠一秒钟**

                Thread.sleep(1000);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

**// 普通优先级线程类，实现 Runnable 接口**

class NormalPriorityThread implements Runnable {

    @Override

    public void run() {

**// 循环输出1到10之间的数字及线程名**

        for (int i = 1; i <= 10; i++) {

            System.out.println(Thread.currentThread().getName() + ": " + i);

            try {

**// 每次输出后休眠一秒钟**

                Thread.sleep(1000);

            } catch (InterruptedException e) {

                e.printStackTrace();

            }

        }

    }

}

**// 低优先级线程类，实现 Runnable 接口**

class LowPriorityThread implements Runnable {

    @Override

    public void run() {

**// 循环输出1到10之间的数字及线程名**

        for (int i = 1; i <= 10; i++) {

            System.out.println(Thread.currentThread().getName() + ": " + i);

            try {

**// 每次输出后休眠一秒钟**

                Thread.sleep(1000);

            } catch (InterruptedException e) {

                e.printStackTrace();

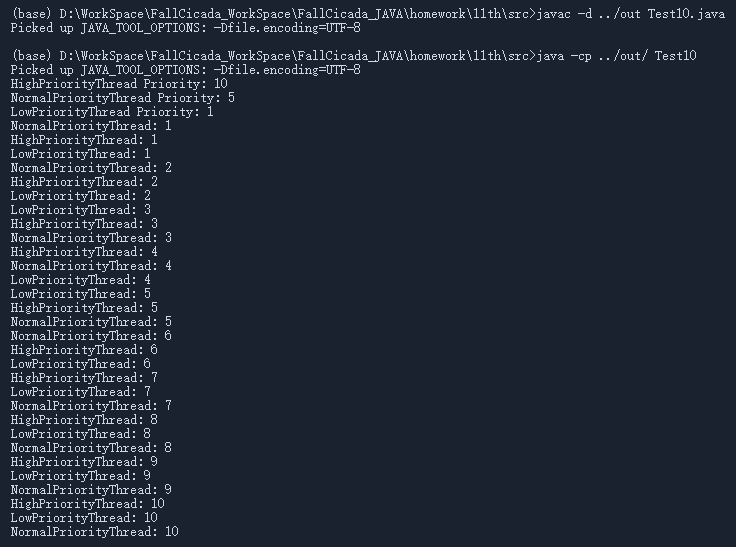
            }

        }

    }

}

运行结果：



思考题：

三个线程分别被设置为不同的优先级：高优先级（MAX\_PRIORITY）、普通优先级（NORM\_PRIORITY）和低优先级（MIN\_PRIORITY）。每个线程都会循环输出1到10的数字，并在每次输出后休眠1秒。而每次运行不一定是按照优先级的次序来运行的

原因：

1. **抢占式调度**：Java虚拟机（JVM）采用的是抢占式的调度策略，这意味着优先级较高的线程有更高的概率获得执行权，但这并不意味着它可以完全排除低优先级线程的执行。实际上，操作系统如何调度线程还取决于很多因素，包括但不限于当前系统负载、其他线程的行为以及操作系统自身的调度策略。
2. **休眠与阻塞**：由于每个线程在输出之后都会调用Thread.sleep(1000);，这意味着每个线程都会暂停执行1秒。在这个期间，其他线程有机会被调度执行。

# 龟兔赛跑

## 需求分析：

编写程序，使用多线程实现龟兔赛跑游戏。

* 1. 可以使用随机数取得0~1之间的随机数模拟比赛进程
  2. 如果随机数在0~0.3之间代表兔子开始跑,每次跑2米。如果随机数在0.3~1之间代表乌龟开始跑，每次跑1米
  3. 总距离为100米，先跑完100米者为胜利者，输出赛跑过程及谁取得冠军

答：

代码：

import java.util.Random;

public class Test11 {

    public static void main(String[] args) {

**// 创建 Race 对象**

        Race race = new Race();

**// 创建兔子线程，并命名为 "Rabbit"**

        Thread rabbitThread = new Thread(race.new Rabbit(), "Rabbit");

**// 创建乌龟线程，并命名为 "Turtle"**

        Thread turtleThread = new Thread(race.new Turtle(), "Turtle");

**// 启动兔子线程**

        rabbitThread.start();

**// 启动乌龟线程**

        turtleThread.start();

    }

}

class Race {

**// 定义总距离为100米**

    private static final int TOTAL\_DISTANCE = 100;

**// 记录兔子跑的距离**

    private int rabbitDistance = 0;

**// 记录乌龟跑的距离**

    private int turtleDistance = 0;

**// 标记比赛是否结束**

    private boolean raceOver = false;

**// 兔子线程类，实现 Runnable 接口**

    class Rabbit implements Runnable {

        @Override

        public void run() {

            Random random = new Random();

**// 当比赛未结束时，循环执行**

            while (!raceOver) {

**// 生成一个0到1之间的随机数**

                double chance = random.nextDouble();

**// 如果随机数在0到0.3之间，兔子跑2米**

                if (chance <= 0.3) {

                    rabbitDistance += 2;

                    System.out.println(Thread.currentThread().getName() + " ran 2 meters, total: " + rabbitDistance + " meters");

                }

**// 检查是否有胜利者**

                gameOver();

                try {

**// 线程休眠100毫秒**

                    Thread.sleep(100);

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }

    }

**// 乌龟线程类，实现 Runnable 接口**

    class Turtle implements Runnable {

        @Override

        public void run() {

            Random random = new Random();

**// 当比赛未结束时，循环执行**

            while (!raceOver) {

**// 生成一个0到1之间的随机数**

                double chance = random.nextDouble();

**// 如果随机数在0.3到1之间，乌龟跑1米**

                if (chance > 0.3) {

                    turtleDistance += 1;

                    System.out.println(Thread.currentThread().getName() + " ran 1 meter, total: " + turtleDistance + " meters");

                }

**// 检查是否有胜利者**

                gameOver();

                try {

**// 线程休眠100毫秒**

                    Thread.sleep(100);

                } catch (InterruptedException e) {

                    e.printStackTrace();

                }

            }

        }

    }

**// 检查是否有胜利者的方法**

    private synchronized void gameOver() {

**// 如果兔子跑完100米且比赛未结束，兔子获胜**

        if (rabbitDistance >= TOTAL\_DISTANCE && !raceOver) {

            raceOver = true;

            System.out.println("Rabbit wins the race!");

**// 如果乌龟跑完100米且比赛未结束，乌龟获胜**

        } else if (turtleDistance >= TOTAL\_DISTANCE && !raceOver) {

            raceOver = true;

            System.out.println("Turtle wins the race!");

        }

    }

}

运行结果：  
(base) D:\WorkSpace\FallCicada\_WorkSpace\FallCicada\_JAVA\homework\11th\src>javac -d ../out Test11.java

Picked up JAVA\_TOOL\_OPTIONS: -Dfile.encoding=UTF-8

(base) D:\WorkSpace\FallCicada\_WorkSpace\FallCicada\_JAVA\homework\11th\src>java -cp ../out/ Test11

Picked up JAVA\_TOOL\_OPTIONS: -Dfile.encoding=UTF-8

Turtle 跑了 1 m, 总奔跑路程: 1 m

Turtle 跑了 1 m, 总奔跑路程: 2 m

Rabbit 跑了 2 m, 总奔跑路程: 2 m

Rabbit 跑了 2 m, 总奔跑路程: 4 m

Turtle 跑了 1 m, 总奔跑路程: 3 m

Rabbit 跑了 2 m, 总奔跑路程: 6 m

Turtle 跑了 1 m, 总奔跑路程: 4 m

Turtle 跑了 1 m, 总奔跑路程: 5 m

Turtle 跑了 1 m, 总奔跑路程: 6 m

Turtle 跑了 1 m, 总奔跑路程: 7 m

Turtle 跑了 1 m, 总奔跑路程: 8 m

Turtle 跑了 1 m, 总奔跑路程: 9 m

Turtle 跑了 1 m, 总奔跑路程: 10 m

Rabbit 跑了 2 m, 总奔跑路程: 8 m

Turtle 跑了 1 m, 总奔跑路程: 11 m

Turtle 跑了 1 m, 总奔跑路程: 12 m

Turtle 跑了 1 m, 总奔跑路程: 13 m

Rabbit 跑了 2 m, 总奔跑路程: 10 m

Turtle 跑了 1 m, 总奔跑路程: 14 m

Turtle 跑了 1 m, 总奔跑路程: 15 m

Turtle 跑了 1 m, 总奔跑路程: 16 m

Rabbit 跑了 2 m, 总奔跑路程: 12 m

Turtle 跑了 1 m, 总奔跑路程: 17 m

Rabbit 跑了 2 m, 总奔跑路程: 14 m

Turtle 跑了 1 m, 总奔跑路程: 18 m

Turtle 跑了 1 m, 总奔跑路程: 19 m

Turtle 跑了 1 m, 总奔跑路程: 20 m

Turtle 跑了 1 m, 总奔跑路程: 21 m

Turtle 跑了 1 m, 总奔跑路程: 22 m

Turtle 跑了 1 m, 总奔跑路程: 23 m

Turtle 跑了 1 m, 总奔跑路程: 24 m

Rabbit 跑了 2 m, 总奔跑路程: 16 m

Rabbit 跑了 2 m, 总奔跑路程: 18 m

Turtle 跑了 1 m, 总奔跑路程: 25 m

Rabbit 跑了 2 m, 总奔跑路程: 20 m

Turtle 跑了 1 m, 总奔跑路程: 26 m

Turtle 跑了 1 m, 总奔跑路程: 27 m

Turtle 跑了 1 m, 总奔跑路程: 28 m

Rabbit 跑了 2 m, 总奔跑路程: 22 m

Turtle 跑了 1 m, 总奔跑路程: 29 m

Rabbit 跑了 2 m, 总奔跑路程: 24 m

Turtle 跑了 1 m, 总奔跑路程: 30 m

Turtle 跑了 1 m, 总奔跑路程: 31 m

Turtle 跑了 1 m, 总奔跑路程: 32 m

Turtle 跑了 1 m, 总奔跑路程: 33 m

Turtle 跑了 1 m, 总奔跑路程: 34 m

Turtle 跑了 1 m, 总奔跑路程: 35 m

Turtle 跑了 1 m, 总奔跑路程: 36 m

Rabbit 跑了 2 m, 总奔跑路程: 26 m

Turtle 跑了 1 m, 总奔跑路程: 37 m

Turtle 跑了 1 m, 总奔跑路程: 38 m

Turtle 跑了 1 m, 总奔跑路程: 39 m

Turtle 跑了 1 m, 总奔跑路程: 40 m

Turtle 跑了 1 m, 总奔跑路程: 41 m

Turtle 跑了 1 m, 总奔跑路程: 42 m

Turtle 跑了 1 m, 总奔跑路程: 43 m

Turtle 跑了 1 m, 总奔跑路程: 44 m

Rabbit 跑了 2 m, 总奔跑路程: 28 m

Rabbit 跑了 2 m, 总奔跑路程: 30 m

Turtle 跑了 1 m, 总奔跑路程: 45 m

Rabbit 跑了 2 m, 总奔跑路程: 32 m

Turtle 跑了 1 m, 总奔跑路程: 46 m

Turtle 跑了 1 m, 总奔跑路程: 47 m

Turtle 跑了 1 m, 总奔跑路程: 48 m

Turtle 跑了 1 m, 总奔跑路程: 49 m

Turtle 跑了 1 m, 总奔跑路程: 50 m

Turtle 跑了 1 m, 总奔跑路程: 51 m

Turtle 跑了 1 m, 总奔跑路程: 52 m

Turtle 跑了 1 m, 总奔跑路程: 53 m

Rabbit 跑了 2 m, 总奔跑路程: 34 m

Turtle 跑了 1 m, 总奔跑路程: 54 m

Rabbit 跑了 2 m, 总奔跑路程: 36 m

Turtle 跑了 1 m, 总奔跑路程: 55 m

Turtle 跑了 1 m, 总奔跑路程: 56 m

Turtle 跑了 1 m, 总奔跑路程: 57 m

Rabbit 跑了 2 m, 总奔跑路程: 38 m

Turtle 跑了 1 m, 总奔跑路程: 58 m

Turtle 跑了 1 m, 总奔跑路程: 59 m

Turtle 跑了 1 m, 总奔跑路程: 60 m

Turtle 跑了 1 m, 总奔跑路程: 61 m

Turtle 跑了 1 m, 总奔跑路程: 62 m

Turtle 跑了 1 m, 总奔跑路程: 63 m

Turtle 跑了 1 m, 总奔跑路程: 64 m

Turtle 跑了 1 m, 总奔跑路程: 65 m

Rabbit 跑了 2 m, 总奔跑路程: 40 m

Turtle 跑了 1 m, 总奔跑路程: 66 m

Rabbit 跑了 2 m, 总奔跑路程: 42 m

Turtle 跑了 1 m, 总奔跑路程: 67 m

Turtle 跑了 1 m, 总奔跑路程: 68 m

Turtle 跑了 1 m, 总奔跑路程: 69 m

Turtle 跑了 1 m, 总奔跑路程: 70 m

Rabbit 跑了 2 m, 总奔跑路程: 44 m

Turtle 跑了 1 m, 总奔跑路程: 71 m

Rabbit 跑了 2 m, 总奔跑路程: 46 m

Rabbit 跑了 2 m, 总奔跑路程: 48 m

Turtle 跑了 1 m, 总奔跑路程: 72 m

Turtle 跑了 1 m, 总奔跑路程: 73 m

Turtle 跑了 1 m, 总奔跑路程: 74 m

Turtle 跑了 1 m, 总奔跑路程: 75 m

Turtle 跑了 1 m, 总奔跑路程: 76 m

Turtle 跑了 1 m, 总奔跑路程: 77 m

Rabbit 跑了 2 m, 总奔跑路程: 50 m

Turtle 跑了 1 m, 总奔跑路程: 78 m

Rabbit 跑了 2 m, 总奔跑路程: 52 m

Turtle 跑了 1 m, 总奔跑路程: 79 m

Turtle 跑了 1 m, 总奔跑路程: 80 m

Turtle 跑了 1 m, 总奔跑路程: 81 m

Rabbit 跑了 2 m, 总奔跑路程: 54 m

Turtle 跑了 1 m, 总奔跑路程: 82 m

Rabbit 跑了 2 m, 总奔跑路程: 56 m

Turtle 跑了 1 m, 总奔跑路程: 83 m

Rabbit 跑了 2 m, 总奔跑路程: 58 m

Rabbit 跑了 2 m, 总奔跑路程: 60 m

Turtle 跑了 1 m, 总奔跑路程: 84 m

Turtle 跑了 1 m, 总奔跑路程: 85 m

Rabbit 跑了 2 m, 总奔跑路程: 62 m

Rabbit 跑了 2 m, 总奔跑路程: 64 m

Turtle 跑了 1 m, 总奔跑路程: 86 m

Rabbit 跑了 2 m, 总奔跑路程: 66 m

Turtle 跑了 1 m, 总奔跑路程: 87 m

Rabbit 跑了 2 m, 总奔跑路程: 68 m

Rabbit 跑了 2 m, 总奔跑路程: 70 m

Turtle 跑了 1 m, 总奔跑路程: 88 m

Turtle 跑了 1 m, 总奔跑路程: 89 m

Turtle 跑了 1 m, 总奔跑路程: 90 m

Rabbit 跑了 2 m, 总奔跑路程: 72 m

Rabbit 跑了 2 m, 总奔跑路程: 74 m

Turtle 跑了 1 m, 总奔跑路程: 91 m

Turtle 跑了 1 m, 总奔跑路程: 92 m

Turtle 跑了 1 m, 总奔跑路程: 93 m

Turtle 跑了 1 m, 总奔跑路程: 94 m

Rabbit 跑了 2 m, 总奔跑路程: 76 m

Turtle 跑了 1 m, 总奔跑路程: 95 m

Turtle 跑了 1 m, 总奔跑路程: 96 m

Rabbit 跑了 2 m, 总奔跑路程: 78 m

Turtle 跑了 1 m, 总奔跑路程: 97 m

Rabbit 跑了 2 m, 总奔跑路程: 80 m

Turtle 跑了 1 m, 总奔跑路程: 98 m

Rabbit 跑了 2 m, 总奔跑路程: 82 m

Turtle 跑了 1 m, 总奔跑路程: 99 m

Turtle 跑了 1 m, 总奔跑路程: 100 m

Turtle wins the race!

# 线程状态

## 需求分析：

线程的状态有哪些？

答：

1. **NEW**(新建)
2. **RUNNABLE(**可运行)
3. **BLOCKED**(锁阻塞)
4. **WAITING(**无限等待)
5. **TIMED\_WAITING**(即使等待)
6. **TERMINATED** (被终止)

# 获取线程状态

## 需求分析：

编写程序，实现创建一个线程，并调用方法实现输出这个线程状态所经历的NEW、RUNNABLE、

TIMED\_WAITING和TERMINATED状态。

答:

代码:

public class Test13 {

    public static void main(String[] args) {

**// 创建线程对象**

        Thread thread = new Thread(new MyThread());

**// 输出线程状态：NEW**

        System.out.println("Thread state: " + thread.getState());

**// 启动线程**

        thread.start();

**// 输出线程状态：RUNNABLE**

        System.out.println("Thread state: " + thread.getState());

        try {

**// 让主线程休眠500毫秒，确保子线程进入TIMED\_WAITING状态**

            Thread.sleep(500);

**// 输出线程状态：TIMED\_WAITING**

            System.out.println("Thread state: " + thread.getState());

**// 等待子线程结束**

            thread.join();

**// 输出线程状态：TERMINATED**

            System.out.println("Thread state: " + thread.getState());

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

}

class MyThread implements Runnable {

    @Override

    public void run() {

        try {

**// 让线程休眠1000毫秒，模拟TIMED\_WAITING状态**

            Thread.sleep(1000);

        } catch (InterruptedException e) {

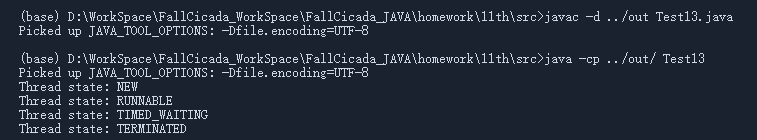
            e.printStackTrace();

        }

    }

}

运行结果:



# 画出线程状态

## 需求分析：

画出线程不同的状态及状态转化关系

