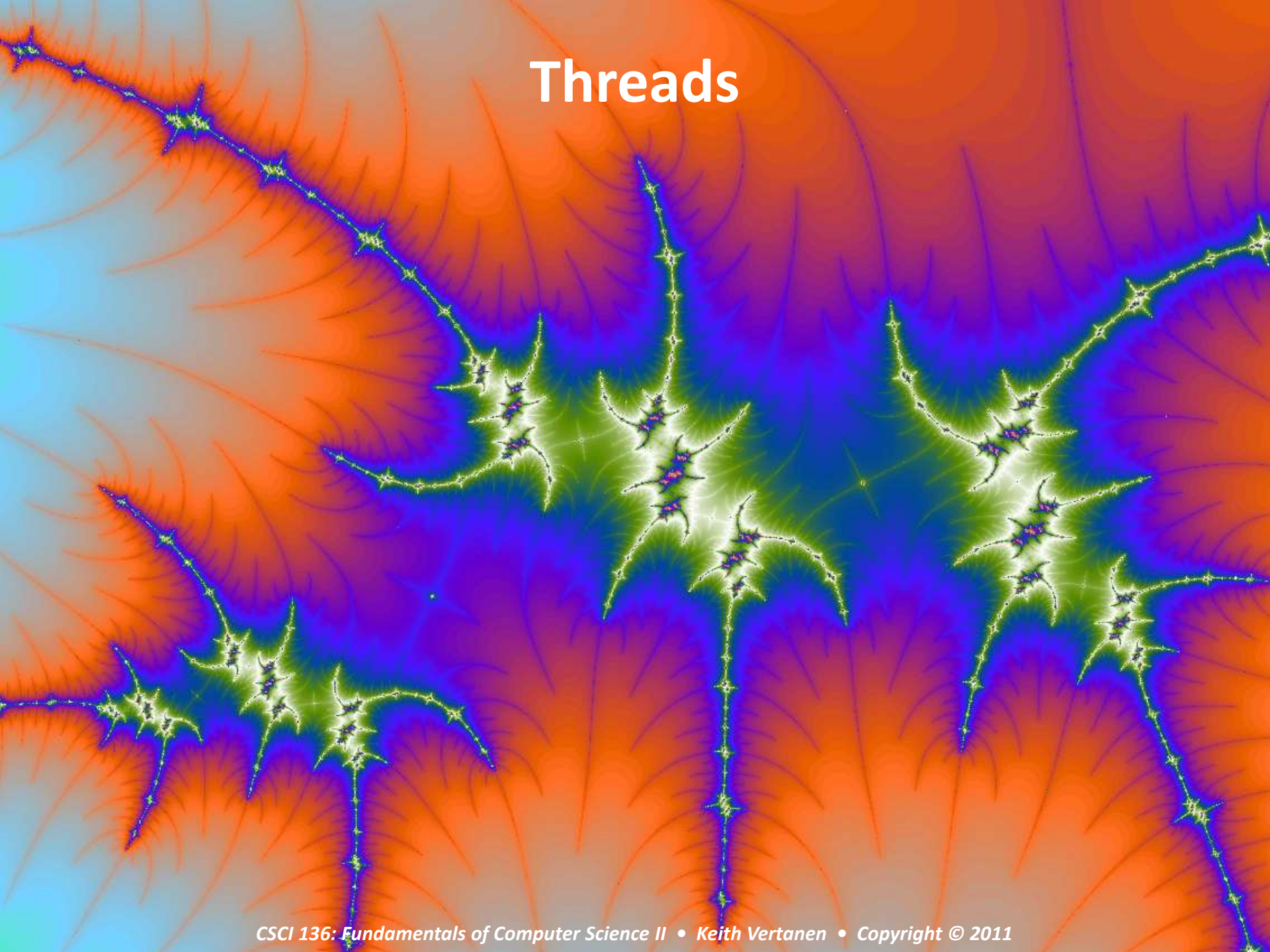
Threads



Overview

- Single-threaded programs
  - What your computer is really up to
  - So many cores, so little utilization…
- Multi-threaded programs
  - Multiple simultaneous paths of execution
    - Seemingly at once (single core)
    - Actually at the same time (multiple cores)
- Threads in Java
  - Creating and starting
  - Unpredictability
  - Sleeping
  - Debugging

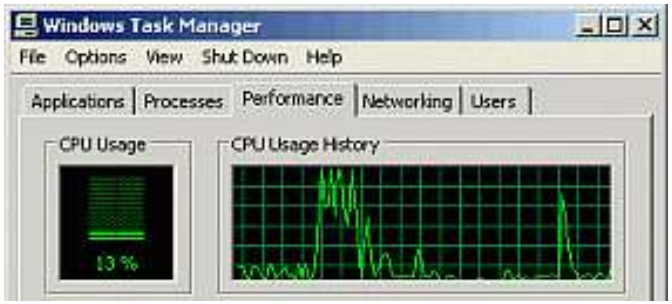
A single-threaded program

```
public class Animal
{
   protected String image;
   protected String audio;

   public Animal(String image,
                 String audio)
   {
9  6  3 this.image = image;
A  7  4 this.audio = audio;
   }

   public void show()
   {
   H StdDraw.picture(0.5,
                     0.5,
                     image);
   I StdAudio.play(audio);
   }
}
```

Animal.java

```
public static void main(String [] args)
{
1  HashMap<String, Animal> map =
      new HashMap<String, Animal>();

2  map.put("dog", new Animal("dog.jpg", "dog.wav"));
5  map.put("cat", new Animal("cat.jpg", "cat.wav"));
8  map.put("cow", new Animal("cow.jpg", "cow.wav"));

B  while (true)
   {
   C StdDraw.clear();
   D String name = StdIn.readLine();
   E Animal animal =
        map.get(name.toLowerCase().trim());
   F if (animal != null)
     G animal.show();
   J StdDraw.show(100);
   }
}
```

AnimalMap.java



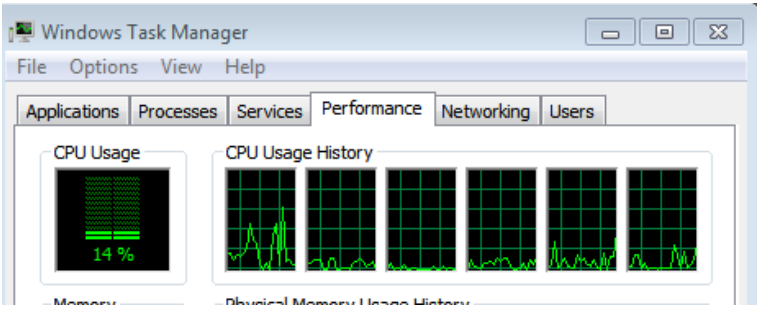
A single-threaded program

```
public class Animal
{
   protected String image;
   protected String audio;

   public Animal(String image,
                 String audio)
   {
9 6 3 this.image = image;
A 7 4 this.audio = audio;
   }

   public void show()
   {
   H StdDraw.picture(0.5,
                     0.5,
                     image);
   I StdAudio.play(audio);
   }
}
```

Animal.java

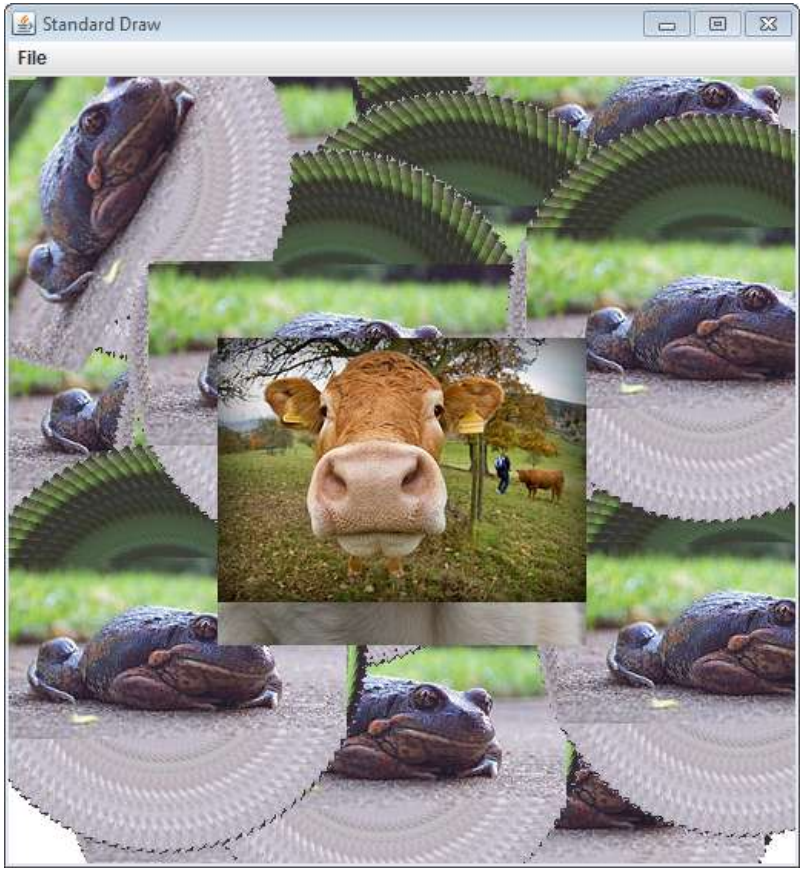
```
public static void main(String [] args)
{
1  HashMap<String, Animal> map =
      new HashMap<String, Animal>();

2  map.put("dog", new Animal("dog.jpg", "dog.wav"));
5  map.put("cat", new Animal("cat.jpg", "cat.wav"));
8  map.put("cow", new Animal("cow.jpg", "cow.wav"));

B  while (true)
   {
   C StdDraw.clear();
   D String name = StdIn.readLine();
   E Animal animal =
        map.get(name.toLowerCase().trim());
   F if (animal != null)
     G animal.show();
   J StdDraw.show(100);
   }
}
```

AnimalMap.java



Multi-threaded animals

- New "magic" program: AnimalMapDeluxe
- Random spinning frogs!
  - Appear every second
- User can still request:
  - "dog", "cat", "cow"
  - Even while frog is spinning



Creating and starting a thread

- Thread
  - A separate path of execution
  - The name of a class in the Java API
  - main() program creates an object of type Thread
- Creating and starting a thread:

```
Thread thread = new Thread();
thread.start();
```

Simple, but doesn't actually do anything. Thread is born, thread dies, end of story.

Making work for a thread

- Thread constructor can take an object
  - Object must implement Runnable
  - An interface with a single method: run()
    - run() may do work forever (e.g. random spinning frogs)
    - run() may do something and exit (e.g. print a message)
    - run() can call other methods, create objects, whatever

```
public class BlastOff implements Runnable
{
   public void run()
   {
      for (int i = 10; i > 0; i--)
         System.out.print(i + " ");
      System.out.println("BLAST OFF!");
   }
}
```

Possible code execution order #1

```
public class BlastOff implements Runnable
{
   public void run()
   {
9 7 5 for (int i = 10; i > 0; i--)
… 8 6    System.out.print(i + " ");
      System.out.println("BLAST OFF!");
   }
}
```

```
public class Launch
{
   public static void main(String [] args)
   {
   1 System.out.println("prepare for launch");
   2 Thread thread = new Thread(new BlastOff());
   3 thread.start();
   4 System.out.println("done with launch");
   }
}
```

```
% java Launch
prepare for launch
done with launch
10 9 8 7 6 5 4 3 2 1 BLAST OFF!
```

Possible code execution order #2

```
public class BlastOff implements Runnable
{
   public void run()
   {
9 7 4 for (int i = 10; i > 0; i--)
… 8 6    System.out.print(i + " ");
      System.out.println("BLAST OFF!");
   }
}
```

```
public class Launch
{
   public static void main(String [] args)
   {
   1 System.out.println("prepare for launch");
   2 Thread thread = new Thread(new BlastOff());
   3 thread.start();
   5 System.out.println("done with launch");
   }
}
```

```
% java Launch
prepare for launch
done with launch
10 9 8 7 6 5 4 3 2 1 BLAST OFF!
```

Possible code execution order #3

```
public class BlastOff implements Runnable
{
   public void run()
   {
9 7 4 for (int i = 10; i > 0; i--)
… 8 5    System.out.print(i + " ");
      System.out.println("BLAST OFF!");
   }
}
```

```
public class Launch
{
   public static void main(String [] args)
   {
   1 System.out.println("prepare for launch");
   2 Thread thread = new Thread(new BlastOff());
   3 thread.start();
   6 System.out.println("done with launch");
   }
}
```

```
% java Launch
prepare for launch
10 done with launch
9 8 7 6 5 4 3 2 1 BLAST OFF!
```

Possible code execution order #4

```
public class BlastOff implements Runnable
{
   public void run()
   {
9 6 4 for (int i = 10; i > 0; i--)
… 7 5    System.out.print(i + " ");
      System.out.println("BLAST OFF!");
   }
}
```

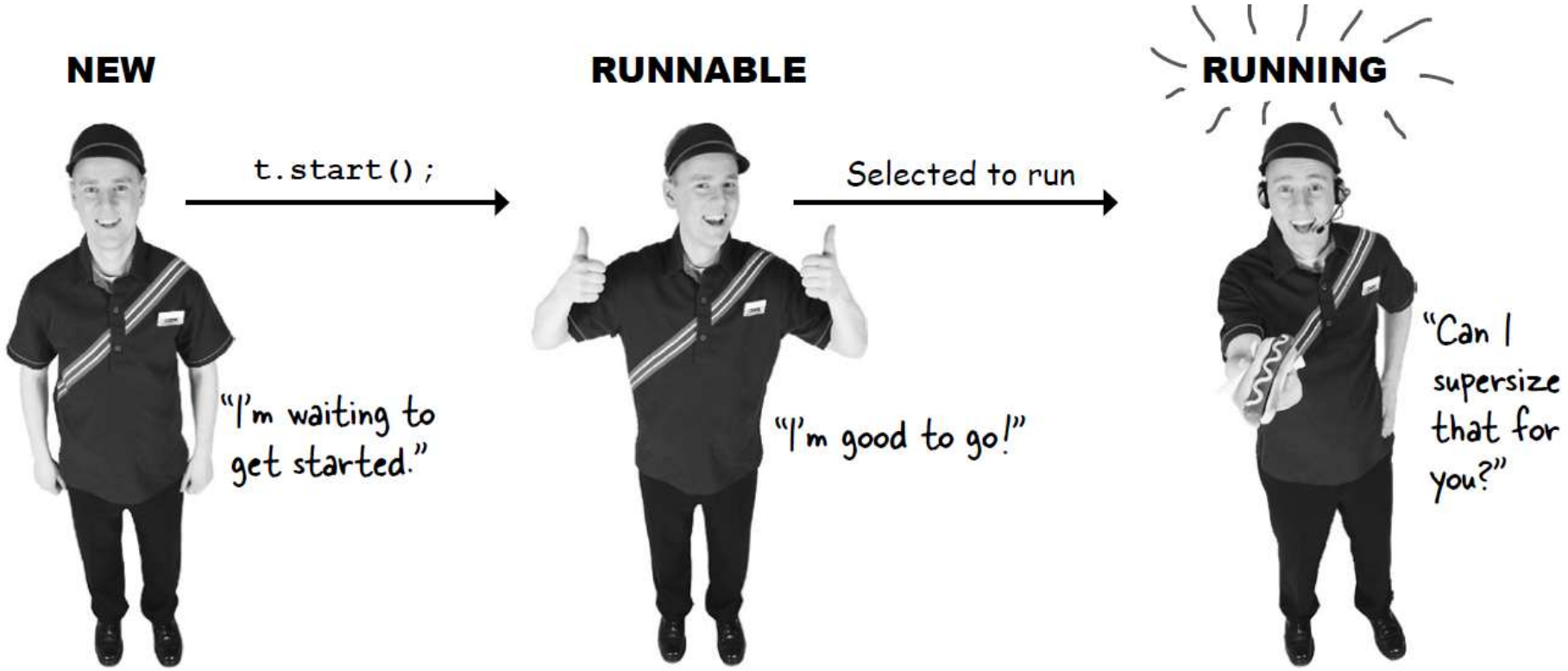
Lots more possible orders!

Depends on ***thread scheduler***

```
public class Launch
{
   public static void main(String [] args)
   {
   1 System.out.println("prepare for launch");
   2 Thread thread = new Thread(new BlastOff());
   3 thread.start();
   8 System.out.println("done with launch");
   }
}
```

```
% java Launch
prepare for launch
10 9 done with launch
8 7 6 5 4 3 2 1 BLAST OFF!
```

Thread states: startup




```
Thread t = new Thread(r);
```

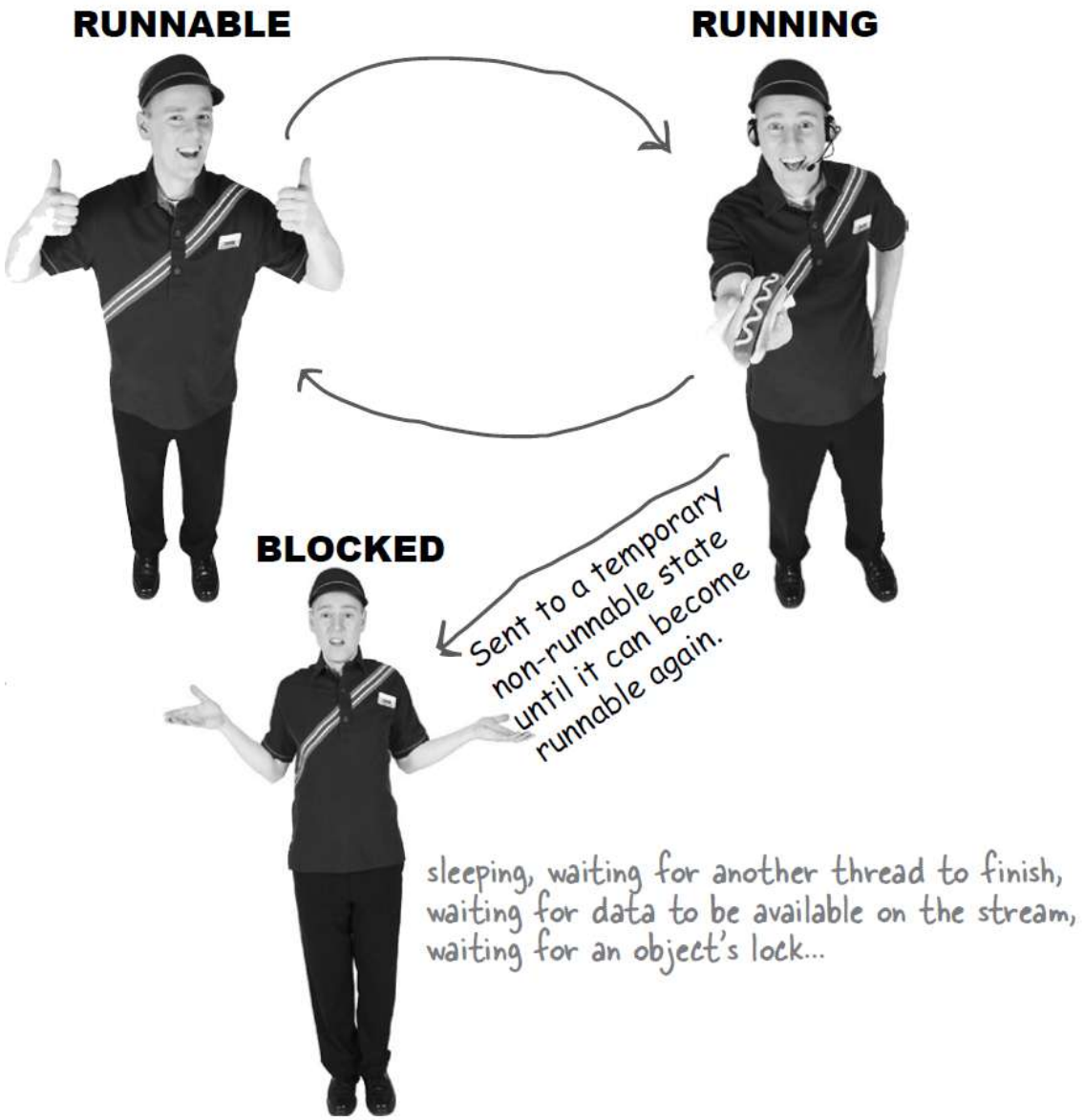
Just a normal object on the heap

```
t.start();
```

Waiting to be selected by the ***thread scheduler***

Actually running on the CPU

Thread states



Launching multiple threads

- Main program can launch > 1 threads
  - Split work into multiple parts

```
public class MultiLaunch
{
   public static void main(String [] args)
   {
      final int N = Integer.parseInt(args[0]);
      System.out.println("prepare for multi launch");
      Thread [] threads = new Thread[N];
      for (int i = 0; i < N; i++)
      {
         threads[i] = new Thread(new BlastOff());
         threads[i].start();
      }
      System.out.println("done launching");
   }
}
```

```
% java MultiLaunch 3
prepare for multi launch
10 10 10 done launching
9 9 9 8 8 7 8 6 7 5 7 4 6 3 6 2 5 1 5 BLAST OFF!
4 4 3 3 2 2 1 BLAST OFF!
1 BLAST OFF!
```

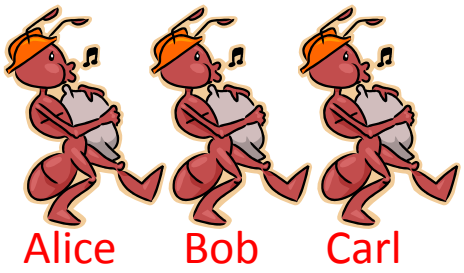
Sleeping

- Making a thread take a nap
  - Specify the nap time in milliseconds
    - Guaranteed to sleep at least this long (maybe longer though).
  - Allows other threads to enter running state
  - Polite behavior when you've got nothing to do
  - `Thread.sleep(ms)`, but must catch an exception

```
while (!StdDraw.hasNextKeyTyped())
{
   // Burns 100% CPU to do nothing!
}
char ch = StdDraw.nextKeyTyped();
```

```
while (!StdDraw.hasNextKeyTyped())
{
   try
   {
      Thread.sleep(10);
   }
   catch (InterruptedException e)
   {
      e.printStackTrace();
   }
}
char ch = StdDraw.nextKeyTyped();
```

Naming threads



- Threads can be given a name
  - Helpful for debugging
  - Second parameter to `Thread()` constructor

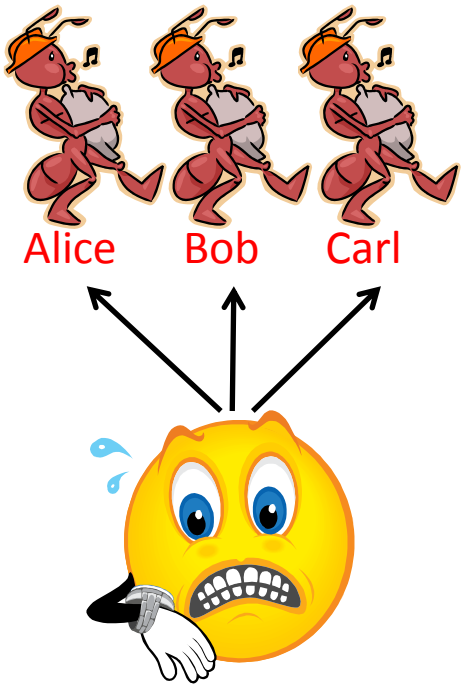
```
public class MultiLaunchSleep
{
   public static void main(String [] args)
   {
      final int N = Integer.parseInt(args[0]);
      System.out.println("prepare for multi launch");
      Thread [] threads = new Thread[N];
      for (int i = 0; i < N; i++)
      {
         threads[i] = new Thread(new BlastOff(), "B" + i);
         threads[i].start();
      }
      System.out.println("done launching");
   }
}
```

A sleepy named worker

```
public class BlastOffSleep implements Runnable
{
   public void run()
   {
      String myName = Thread.currentThread().getName();
      for (int i = 10; i > 0; i--)
      {
         System.out.print(i + "(" + myName + ") ");
         try
         {
            Thread.sleep(1000);
         }
         catch (InterruptedException e)
         {
            e.printStackTrace();
```

```
% java MultiLaunchSleep 3
prepare for multi launch
10(B0) done launching
10(B2) 10(B1) 9(B0) 9(B1) 9(B2) 8(B0) 8(B1) 8(B2) 7(B0) 7(B1) 7(B2)
6(B1) 6(B0) 6(B2) 5(B0) 5(B2) 5(B1) 4(B0) 4(B1) 4(B2) 3(B0) 3(B1)
3(B2) 2(B0) 2(B1) 2(B2) 1(B0) 1(B1) 1(B2) BLAST OFF! (B0)
BLAST OFF! (B1)
BLAST OFF! (B2)
```

Other important Thread tricks



- Main thread can wait for workers
  - Might want to merge the results or such
  - Call `join()` on the thread object

- Passing data to/from a worker
  - `Thread()` constructor is passed a Runnable object
  - You can add any instance variables / methods you want
    - Input: send to constructor
    - Output: make some get data method
  - But you must keep track of the object passed to `Thread()`

Parallel Fibonacci calculator

```
public class FibWorker implements Runnable
{
   private int  num    = 0;
   private long result = 0;

   public FibWorker(int num)
   {
      this.num = num;
   }
   public long getResult()
   {
      return result;
   }
   public void run()
   {
      result = fib(num);
   }
   private long fib(int n)
   {
      if (n == 0) return 0;
      if (n == 1) return 1;
      return fib(n-1) + fib(n-2);
   }
}
```

Get the input, what we'll calculate once somebody says go!

Somebody asking for what we calculated. Should only be called after the thread is known to be done.

Method that runs when somebody call `.start()` on a Thread that has been passed this object.

Parallel Fibonacci calculator

```
public class FibLauncher
{
   public static void main(String [] args)
   {
      Thread    [] threads = new Thread[args.length];
      FibWorker [] workers = new FibWorker[args.length];
      for (int i = 0; i < args.length; i++)
      {
         workers[i] = new FibWorker(Integer.parseInt(args[i]));
         threads[i] = new Thread(workers[i]);
         threads[i].start();
      }
      try
      {
         for (int i = 0; i < args.length; i++)
         {
             threads[i].join();
             System.out.print("fib(" + args[i] + ") = ");
             System.out.println(workers[i].getResult());
         }
      }
      catch (InterruptedException e) { e.printStackTrace(); }
   }
}
```

We must keep track of two parallel arrays, one for threads and one for worker objects.

Setup worker with its job.

Once a thread is done, we know it has a good output value.

Programming activity

- Build a class that implements a worker thread
  - Draw something in unit box
  - Sleep
  - Change something about the drawing
  - Repeat forever
  - Don't worry about erasing (don't call `StdDraw.clear()`)
- Email me your completed class
  - I'll integrate into my ThreadZoo program

Summary

- Java Thread
  - Multiple simultaneous paths of execution
  - May be really simultaneous (multiple cores) or simply seem that way (single core)
- Important Thread skills
  - Implementing a worker class
  - Starting a thread
  - Making a thread sleep
  - Waiting for a thread to finish
  - Getting input to a thread
  - Getting output from a thread