Threads

Thread: Individual and separate unit of execution that is part of a process

Can run separated from other Threads

Multiple threads can work together to accomplish a common goal

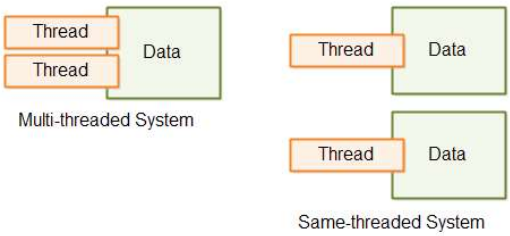
```
public class Printer {
   private String toSay;
   public Printer(String st) {
      toSay = st;
   }
   public void start() { // Weird, I know...
      run();
   }
   public void run() {
      try {
         for (int i = 1; i <= 20; i++) {
            System.out.println(this + " " + toSay);
            Thread.sleep(1000);
         }
      } catch(InterruptedException e) {
         System.out.println(e);
      }
   }
}
```

```java
public class PrinterMain {
    public static void main(String [] args) {
        Printer thr1 = new Printer("Hello");
        Printer thr2 = new Printer("There");
        thr1.start();
        thr2.start();
    }
}
```

```java
public class Printer extends Thread {
    private String toSay;
    public Printer(String st) {
        toSay = st;
    }
    public void run() {
        try {
            for (int i = 1; i <= 20; i++) {
                System.out.println(this + " " + toSay);
                sleep(1000);
            }
        } catch(InterruptedException e) {
            System.out.println(e);
        }
    }
}
```

Threads

Can operate on private or *shared* data




```
public class Counter {
   private int count;
   public Counter() { count = 0; }
   public void increment() { count++; }
   public int getCounter() { return count; }
}

public class CounterThread extends Thread {
   private Counter c;

   public CounterThread(Counter c) {
      this.c =c;
   }
   public void run() {
      c.increment();
   }
}
```

```java
public class SharedCounter {
   public static void main(String [] args) {
      Counter c = new Counter();
      Thread t1 = new CounterThread(c);
      Thread t2 = new CounterThread(c);
      t1.start();
      t2.start();
      try {
         t1.join(); // Wait for Thread to finish
         t2.join();
      } catch (InterruptedException e) {
         System.out.println(e.getMessage());
      }
      System.out.println(c.getCounter());
   }
}
```

```java
public class SharedCounter2 {
   public static void main(String[] args) {
      Thread[] many = new Thread[5000]; // No problem!
      Counter c = new Counter();

      for (int i = 0; i < many.length; i++) {
         many[i] = new CounterThread(c);
      }
      for (int i = 0; i < many.length; i++) {
         many[i].start();
      }
      try {
         for (int i = 0; i < many.length; i++) {
            many[i].join();
         }
      } catch (InterruptedException e) {
         System.out.println(e.getMessage());
      }
      System.out.println(c.getCounter());
   }
}
```

Race Conditions

A race condition (…) is the behavior of (…) software (…) where the output is dependent on the sequence or timing of other uncontrollable events.

It becomes a bug when events do not happen in the order the programmer intended. [Wikipedia]

```
public class BankAccount {
   private int balance;
   public BankAccount() {
      balance = 0;
   }
   public int getBalance() {
      return balance;
   }
   public void deposit(int amount) {
      balance += amount;
   }
   public void withdraw(int amount) {
      balance -= amount;
   }
}
```

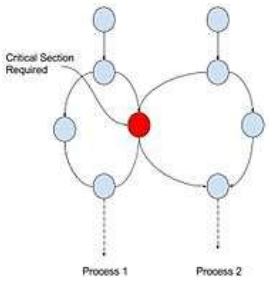
```
public class AccountThread extends Thread {
   private BankAccount acct;

   public AccountThread(BankAccount acct) {
      this.acct = acct;
   }

   public void run() {
      for (int i = 1; i <= 2000; i++) {
         acct.deposit(i);
         acct.withdraw(i); // Should balance
      }
   }
}
```

```
public class UpdateMany {
      public static void main(String[] args) {
         Thread[] many = new Thread[5000];
         BankAccount a = new BankAccount();
         for (int i = 0; i < many.length; i++) {
            many[i] = new AccountThread(a);
         }
         for (int i = 0; i < many.length; i++) {
            many[i].start();
         }
         try {
            for (int i = 0; i < many.length; i++) {
               many[i].join();
            }
         } catch (InterruptedException e) {
            System.out.println(e.getMessage());
         }
         System.out.println(a.getBalance());
      }
   }
```

A **Critical Section** is the part of a program that accesses shared resources.

Only when a process is in its Critical Section can it be in a position to disrupt other processes.

We can avoid race conditions by making sure that no two processes enter their Critical Sections at the same time.



Java has a couple of ways to protect Critical Sections.

Easiest: Mark a method as a Critical Section:

```
public class BankAccount {
    private int balance;
    public BankAccount() {
        balance = 0;
    }
    public int getBalance() {
        return balance;
    }
    public synchronized void deposit(int amount) {
        balance += amount;
    }
    public synchronized void withdraw(int amount) {
        balance -= amount;
    }
}
```

```java
public class BankAccount {
    private int balance;
    public BankAccount() {
        balance = 0;
    }
    public int getBalance() {
        return balance;
    }
    public synchronized void deposit(int amount) {
        balance += amount;
    }
    public synchronized void withdraw(int amount) {
        balance -= amount;
    }
}
```