# 3. Java Nuggets in a Nutshell

- Sequential Java Programming
- Shared Memory Multi-Threading in Java

# Help Yourself

## Java Language and Standard Development Kit (SDK)

- Read (and download) the Java documentation:
  `http://java.sun.com/javase/6/docs`
  - Language:
    `http://java.sun.com/javase/6/docs/technotes/guides/language`
  - Application Programming Interface (API):
    `http://java.sun.com/javase/6/docs/api`

- *INF422 is not a Java programming or software engineering course: we use Java because it is portable, reasonably clean, pedagogical, productive to use, and free software*

# 3. Java Nuggets in a Nutshell

- Sequential Java Programming
- Shared Memory Multi-Threading in Java

# Java Programming Language

- Imperative language
  - Primitive (scalar), array and `Object` types
  - Control flow: `if`, `while`, `for`...
  - Functional abstraction: *methods*
  - Modular composition: *separate compilation* `.java`→`.class`, `package`
- Object-oriented language
  - Goal: *code reuse and modularization* (complementary to functions alone)
  - Classes, objects, generic types
  - Inheritance, interfaces

# Java Programming — Why?

- Portable
  *"Write once, run everywhere"* Scott McNeally, Sun's CEO
- Virtual Machine (VM): abstraction of the machine, standardized
- Sandboxing of applications on a given VM
- Productive and safer memory management: concurrent garbage collector
- Encourage good software engineering practices
- Fast and lightweight thanks to Just-in-Time (JIT) compilation
- Free software implementation
- More? see Google's arguments about its Dalvik VM

# Java Programming — Anticipated from INF431

## Course Syllabus

INF431: first chapter on Java programming

http://www.enseignement.polytechnique.fr/informatique/INF422/inf431_chap1.pdf

The Java Trail

http://java.sun.com/docs/books/tutorial/java

## Eclipse Integrated Development Environment (IDE)

Thanks to Julien Cervelle: Eclipse tutorial

http://www.enseignement.polytechnique.fr/profs/informatique/Julien.Cervelle/eclipse

# Java Trail: Class

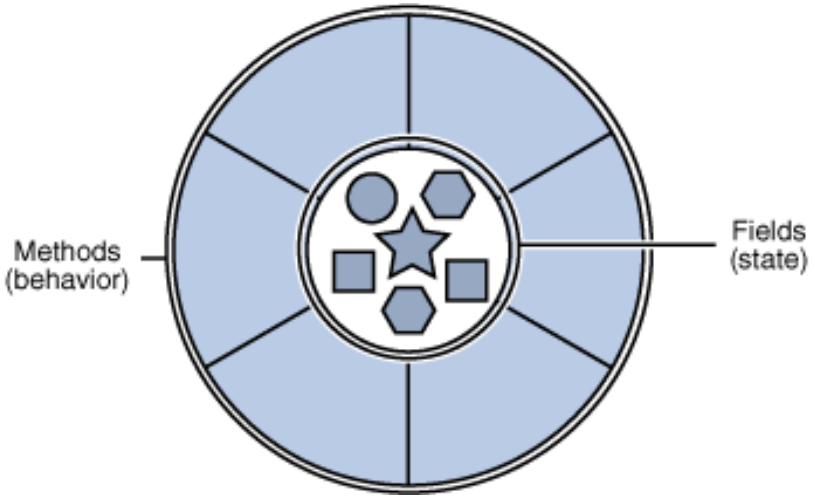
```
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;
    Color color;

    Bicycle(Color c) {
        color = c;
    }

    void changeCadence(int newValue) {
         cadence = newValue;
    }
    void changeGear(int newValue) {
         gear = newValue;
    }
    void speedUp(int increment) {
         speed = speed + increment;
    }
    void applyBrakes(int decrement) {
         speed = speed - decrement;
    }
    void printStates() {
         System.out.println("cadence:" + cadence + " speed:" + speed + " gear:" + gear);
    }

}
```



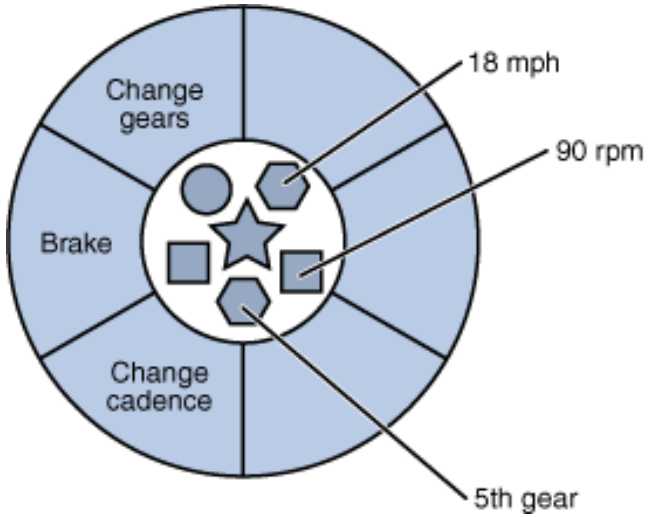
# Java Trail: Objects

```
class BicycleDemo {
     public static void main(String[] args) {

          // Create two different Bicycle objects
          Bicycle bike1 = new Bicycle(Color.BLUE);
          Bicycle bike2 = new Bicycle(Color.RED);

          // Invoke methods on those objects
          bike1.changeCadence(50);
          bike1.speedUp(10);
          bike1.changeGear(2);
          bike1.printStates();

          bike2.changeCadence(50);
          bike2.speedUp(10);
          bike2.changeGear(2);
          bike2.changeCadence(40);
          bike2.speedUp(10);
          bike2.changeGear(3);
          bike2.printStates();
     }
}
```



# Java Trail: Inheritance

- Main purpose: modify or extend the fundamental behavior of a class without copying the code of the *base* (a.k.a. *super*) class

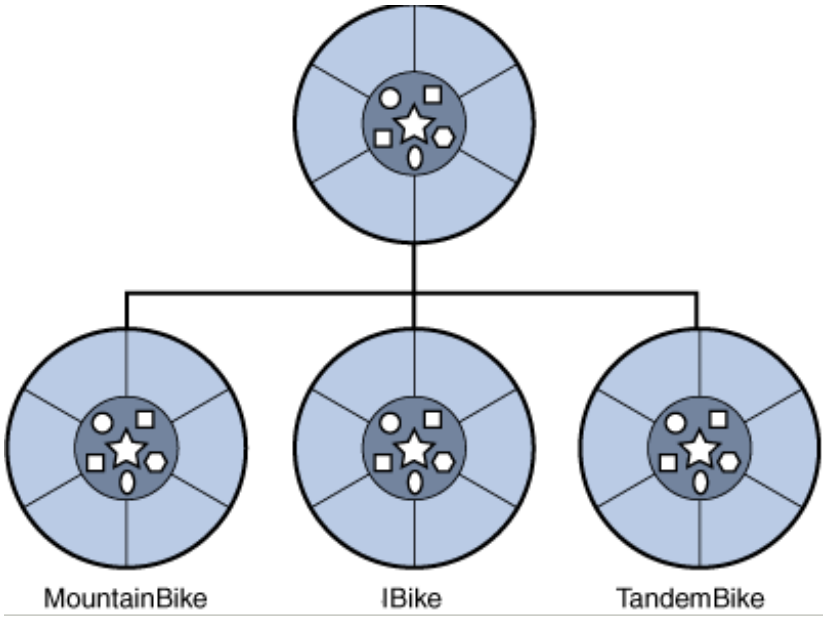
```
class IBike extends Bicycle {

    // New fields defining an i-bike
    MusicPlayer musicPlayer;

    // Overload a method of the parent class
    void printStates() {
        System.out.println("cadence:" + cadence +
          " speed:" + speed + " gear:" + gear +
          " player:" + musicPlayer.getStatus());
    }

    // New method defining an i-bike
    void playSomeMusic() {
        musicPlayer.turnOn();
        musicPlayer.play(musicPlayer.randomTrack());
    }

    // Specialized constructor for an i-bike
    IBike(Color c, MusicPlayer mp) {
        super(c);
        musicPlayer = mp;
    }
}
```



# Java Trail: Interfaces

- Specify a subset of the methods *to be* implemented by a class
- Main purpose: factor out common properties/actions/capacities of classes

```
interface Engine extends Runnable { // Runnable is a class of the java.lang package

    void turnOn();
    void turnOff();
    void refuel(int liters);

}

class MotorBike extends Bicycle implements Engine {

    // If defined, overload the method in the parent class
    void turnOn() { /* ... */ }
    void turnOff() { /* ... */ }
    void refuel(int liters) { /* ... */ }

    void run() {
        changeGear(0); // Disengage the gear
        turnOn();
        /* ... */
    }

}
```

# Java Trail: Inheritance Pitfalls

## Quiz

- Question: should a `Rectangle` class *extend* a `Square` class or the opposite?

# Java Trail: Inheritance Pitfalls

## Quiz

- Question: should a `Rectangle` class *extend* a `Square` class or the opposite?
- Hint: what about the `resize(double x, double y)` method?

# Java Trail: Inheritance Pitfalls

## Quiz

- Question: should a `Rectangle` class *extend* a `Square` class or the opposite?
- Hint: what about the `resize(double x, double y)` method?
- Answer: neither! Both should more likely *implement* a `Shape` interface

# Java Trail: Inheritance Pitfalls

## Quiz

- Question: should a `Rectangle` class *extend* a `Square` class or the opposite?
- Hint: what about the `resize(double x, double y)` method?
- Answer: neither! Both should more likely *implement* a `Shape` interface
- Alternative: `abstract` classes with `abstract` methods (definition deferred to child classes)

# Java Trail: Exceptions

- Control mechanism to trigger and react to exceptionnal events
  - E.g., errors, termination, transition from a steady-state mode to another
- Richer than flags, `break`, `continue`

- *Throw* an exception in a given method
- *Catch* an exception at an arbitrary depth in the method invocation stack (function call stack)
- An exception is an object: two families depending on which base class they inherit from
  - *`Exception`*: the programmer must explicit any such exception in a method signature, and catch it eventually in a calling method
    Example: *`IOException`*
  - *`RuntimeException`*: no constraint on the programmer, but the exception is free to escape all the way to the virtual machine
    Example: *`NullPointerException`*

# Java Trail: Exceptions

```
class File {
    /* ... */

    public boolean createNewFile() throws IOException {
          /* ... */
          if (/* ... */) {
              throw new IOException("I/O Error");
          }
          /* ... */
     }
}

/* ... */

class MyClass {
    /* ... */

        try {
            /* ... */
            File f = new File();
            boolean b = f.createNewFile();
            /* ... */
        } catch (IOException e) {
            System.err.println(e.toString());
            System.exit(1);
        }
}
```

# Java Trail: Generics (Java 5)

```
class Box<T> {

    private T t; // T stands for "Type"

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }
}
```

- Static type checking
  - Safer than casts:
    `Integer someInteger = (Integer)integerBox.get();`
  - Note: the compiler "erases" generic types in favor of casts (dynamic type checking, enforced by Java standard)

```
class BoxDemo {

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        Integer someInteger = integerBox.get(); // No cast!
        System.out.println(someInteger);
    }
}
```

# Java Trail: Generics (Java 5)

```
class Box<T> {

    /* ... */

    public <U> void inspect(U u) {
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

}
```

- Further extensions possible: bounded type parmeters
  - Subtyping constraint `<U extends ParentClass>`
  - Supertyping constraint `<U super ChildClass>`

```
class BoxDemo {

    public static void main(String[] args) {
        Box<Integer> integerBox = new Box<Integer>();
        integerBox.add(new Integer(10));
        integerBox.inspect("some text");
    }

}
```

# Syntactic Sugar: Loops Over Collections

## Array: Default Syntax

```
for (int i=0; i<a.length; i++)
    System.out.println(a[i]);
```

# Syntactic Sugar: Loops Over Collections

## Array: Default Syntax

```
for (int i=0; i<a.length; i++)
    System.out.println(a[i]);
```

## Nicer

```
for (int x : a)
    System.out.println(x);
```

# Syntactic Sugar: Loops Over Collections

## Collection: Default Syntax

```
void turnAllOff(Collection<Engine> c) {
    for (Iterator<Engine> i = c.iterator(); i.hasNext(); )
        i.next().turnOff();
}
```

# Syntactic Sugar: Loops Over Collections

## Collection: Default Syntax

```
void turnAllOff(Collection<Engine> c) {
    for (Iterator<Engine> i = c.iterator(); i.hasNext(); )
        i.next().turnOff();
}
```

## Much Much Nicer

```
void turnAllOff(Collection<Engine> c) {
    for (Engine e : c)
        e.turnOff();
}
```

# Syntactic Sugar: Automatic (Un)Boxing

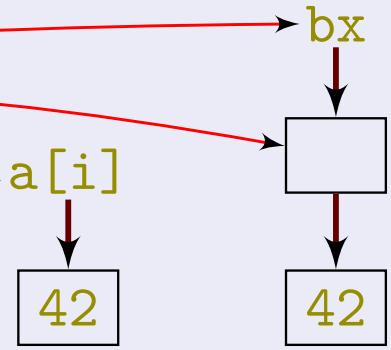
## Converting an Array into a List

```
List<Integer> arrayToList(int[] a) {
    List<Integer> l = new List<Integer>();
    for (int i=0; i<a.length; i++)
        Integer bx = new Integer(a[i]);
        l.set(i, bx);
    return l;
}
```

# Syntactic Sugar: Automatic (Un)Boxing

## Converting an Array into a List

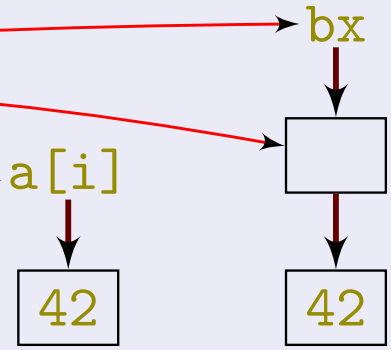
```
List<Integer> arrayToList(int[] a) {
    List<Integer> l = new List<Integer>();
    for (int i=0; i<a.length; i++)
        Integer bx = new Integer(a[i]);
        l.set(i, bx);
    return l;
}
```



# Syntactic Sugar: Automatic (Un)Boxing

## Converting an Array into a List

```
List<Integer> arrayToList(int[] a) {
    List<Integer> l = new List<Integer>();
    for (int i=0; i<a.length; i++)
        Integer bx = new Integer(a[i]);
        l.set(i, bx);
    return l;
}
```




## Much Nicer

```
List<Integer> arrayToList(int[] a) {
    List<Integer> l = new List<Integer>();
    for (int i=0; i<a.length; i++)
        l.set(i, a[i]); // Automatic boxing
    return l;
}
```

# 3. Java Nuggets in a Nutshell

- Sequential Java Programming
- Shared Memory Multi-Threading in Java

# Thread Abstraction

## Thread-Level Concurrency

- Parallelism
  - Take advantage of cache-coherent multiprocessors
  - Moore's law translates into higher needs for thread-level parallelism in applications
- Concurrency
  - Many algorithms can be expressed more naturally with independent computation flows
  - E.g., *graphical user interface* (studied in the labs), web server, etc.
  - Other examples: reactive systems, client-server applications, distributed component engineering (CORBA, *Android Interface Description Language (IDL)*)

# Multi-Threading Concepts

## Program Threads in Shared Memory

Multiple *concurrent* execution contexts of the same program, cooperating over a single, *shared memory space*:

- consistent memory addresses across all threads: "shared address space"
- "shared data" accessed at any address from any thread
- satisfying some *memory consistency* model

## Hardware Threads in Shared Memory

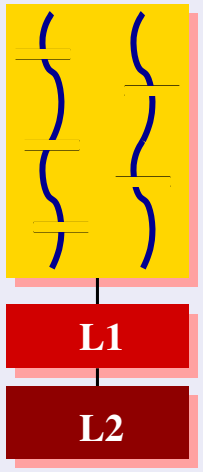
Multiple *concurrent* execution contexts *actively running on the hardware*, cooperating or not, over a shared address space

- Memory consistency model implemented with *cache coherence protocol*
- Hardware implementation differs widely (can also be software)

# Multi-Threaded Architectures

## Simultaneous Multi-Threaded (SMT)

- A.k.a. hyper-threaded (Intel)

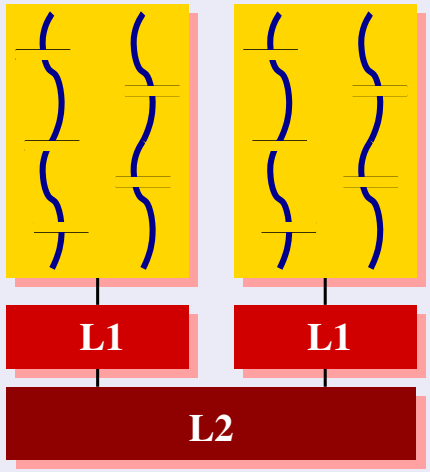



- NVidia's SIMT: combination with Single Instruction Multiple Data (SIMD, vector computing)

# Cache-Coherent Multiprocessor Architectures

## Chip Multi-Processor (CMP)

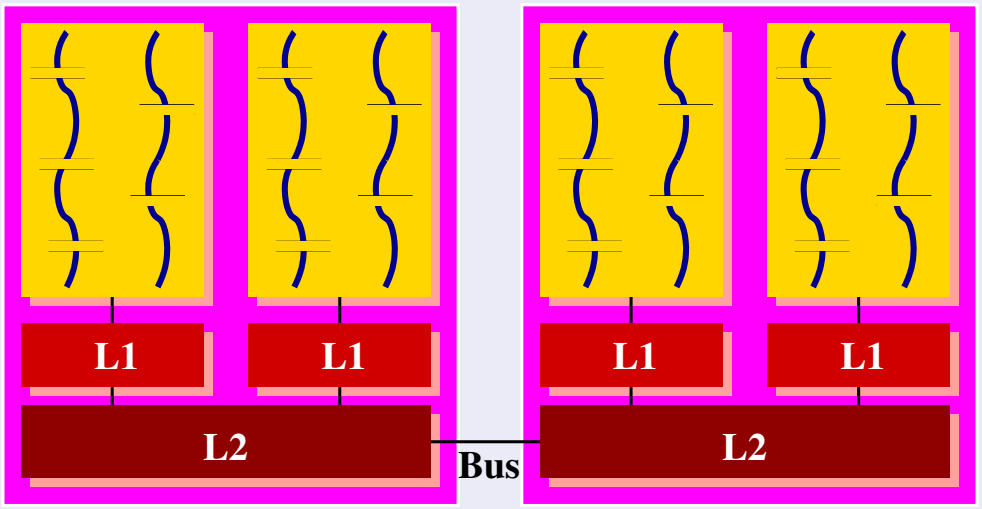
- A.k.a. multicore (Intel)



# Cache-Coherent Multiprocessor Architectures

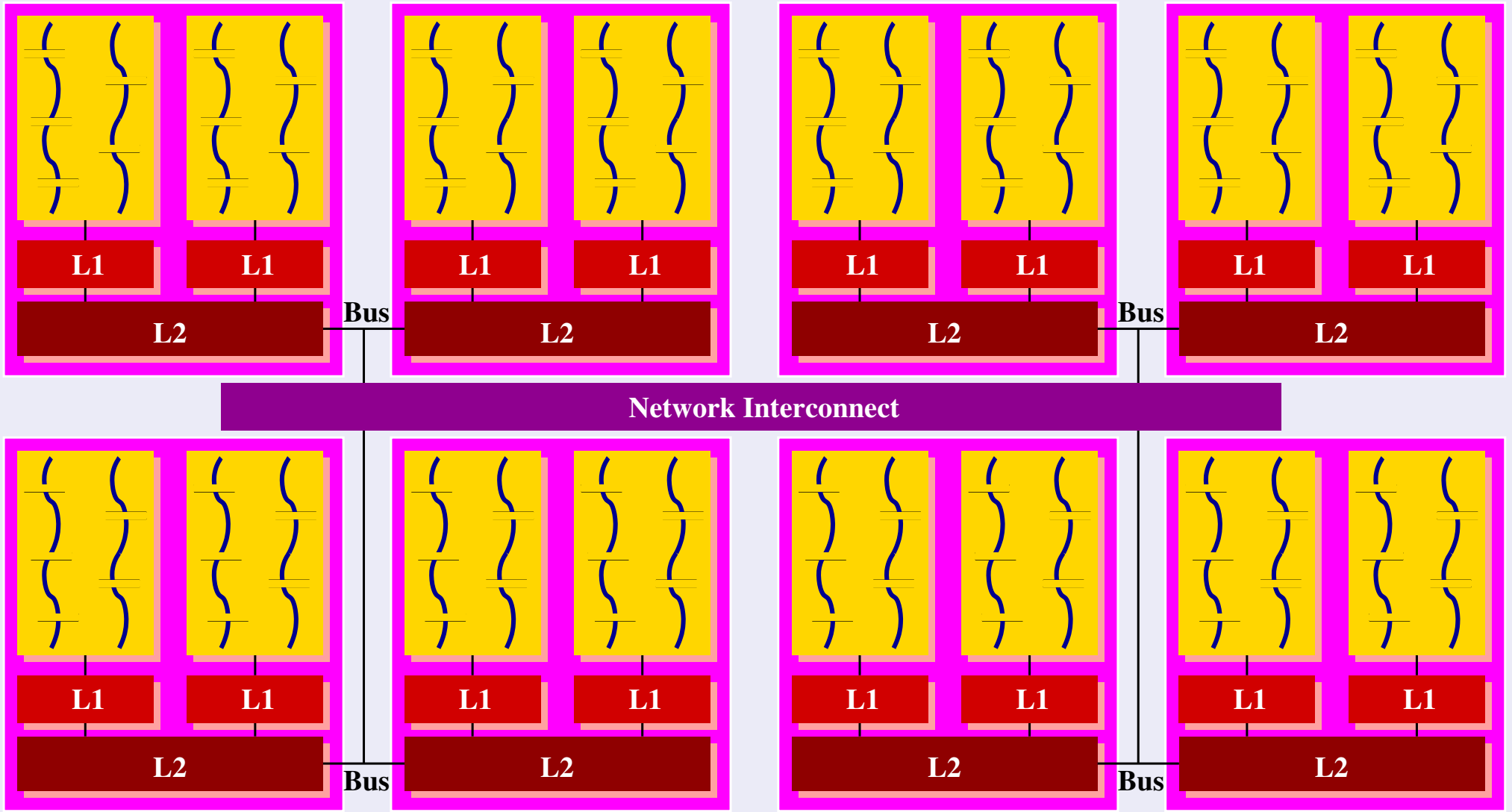
## Symmetric Multi-Processor (SMP)

- Multiple chips interconnected with a bus



# Cache-Coherent Multiprocessor Architectures

## Non-Uniform Memory Architecture (NUMA)



# Multi-Threading Concepts

## Concurrence and Parallelism

*Parallelism* if the steps/instructions executed by different threads occur *simultaneously*

# Multi-Threading Concepts

## Concurrence and Parallelism

*Parallelism* if the steps/instructions executed by different threads occur *simultaneously*

Sounds very natural, no?

# Multi-Threading Concepts

## Concurrence and Parallelism

*Parallelism* if the steps/instructions executed by different threads occur *simultaneously*

Sounds very natural, no?

Not really: if a sequential program (a single thread of execution) computes a function, what does a concurrent program compute?

# Multi-Threading Concepts

## Concurrence and Parallelism

*Parallelism* if the steps/instructions executed by different threads occur *simultaneously*

Sounds very natural, no?

Not really: if a sequential program (a single thread of execution) computes a function, what does a concurrent program compute?

Different answer depending on whether you take the side of the computer architect or the programmer

The telescope's mirror seems to be defective, can we fix it?

... still a hot and difficult research topic today

# Thread Creation

```
class Chronometer extends Thread {
    private int counter = 0;
    private int max;

    Chronometer(int max) { // Constructor to ‘‘pass an argument’’ to the thread
        this.max = max;
    }

    public void run() {
        while (counter < max) { // Iterate max ticks
            try {
                Thread.sleep(1000); // Pause for 1 second
                counter++;
                system.out.println("Tick " + counter + "!");
            } catch (InterruptedException e) { }
        }
    }
}

class BikeRace {
    public static void main(String[] args) {
        Chronometer t = new Chronometer(60); // Set a 1 minute timer
        MotorBike m = new MotorBike(); // MotorBike implements Runnable
        t.start(); // Start the chronometer
        new Thread(m).start(); // Go!
    }
}
```

# Thread Synchronization: Atomic Execution

- Example: concurrently update a given memory location while maintaining *atomicity of a sequence of updates*
- Implementation: a *lock* synchronize the execution of threads entering an atomic sequence of instructions, allowing at most one thread to proceed and delaying the others

```
class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

# Thread Synchronization: Causal Relation

- Goal: implement a *dependence* (causal relation) between threads
  - E.g., pause a consumer thread until the producer has completed its job
- Default Java solution: low-level mechanisms based on `notify()`/`wait()` methods of the `Object` class
- Android: provide a more abstract and more expressive *call-back* mechanism
  - An example will be studied in the labs

# More About Concurrency

- Java is one of the richest mainstream programming language for thread-level concurrency
  - High-level concurrent data-structures
  - High- and low-level synchronization primitives
  - Large efforts to make it as sound and intuitive as possible (but not ideal, and quite empirical)
- More information in the `java.util.concurrent` package and the Java concurrency tutorial:

  http://java.sun.com/docs/books/tutorial/essential/concurrency