Enterprise Programming
(Course Slides)

Module1

Refer for detailed notes :-

https://github.com/sagaruppuluri/EP/blob/main/Module1/README.md

Object Oriented Programming – Core Principles

- **Encapsulation**
- **Inheritance**
- **Abstraction**
- **Polymorphism**
  - **Static**
  - **Dynamic**

Collection – Key Points

List – Duplicates Allowed

Set – No duplicates

Map – Key, Value pair within Map Keys are a Set.

Hash - Hashing, Insertion Order not guaranteed

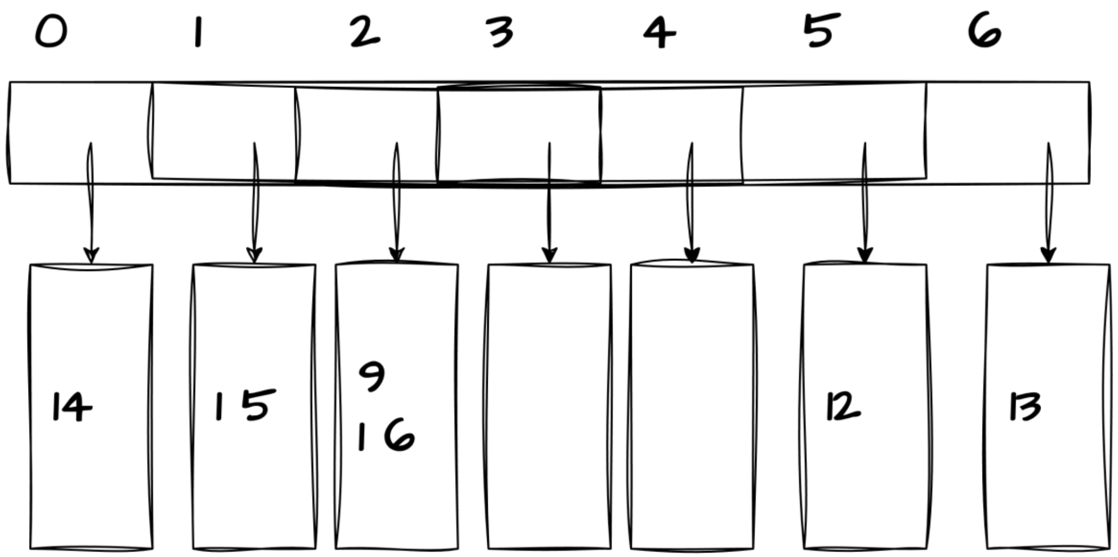
LinkedHash - Insertion Order guaranteed

Tree - Sorted Order

HASHING

Sample hash function => $h(x) = x \bmod 7$

Insertion Order : 9 13 12 15 14 16



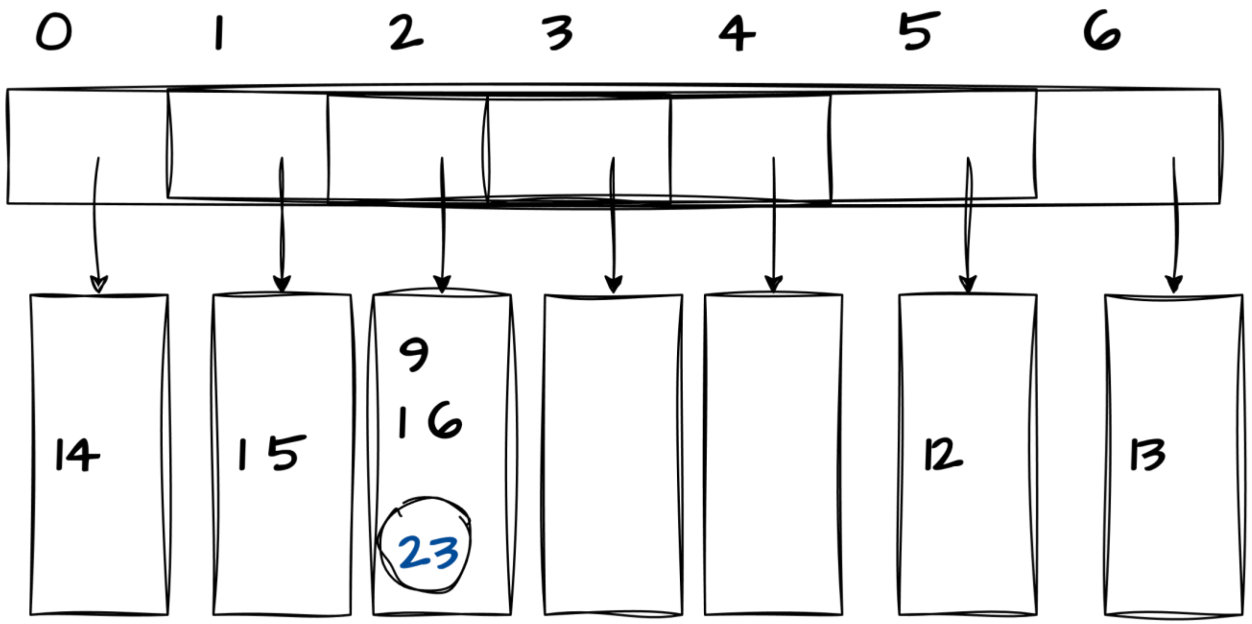
Collect the values : 14 15 9 16 12 13

HENCE : INSERTION ORDER NOT PRESERVED

HASHSET

Sample hash function => $h(x) = x \bmod 7$

Insertion Order : 9 13 12 15 14 16



INSERTING NEW ELEMENT

newVal : 23

↓

hashcode

↓ 2

for each obj in bucket
    if (obj . equals (newVal))
        reject

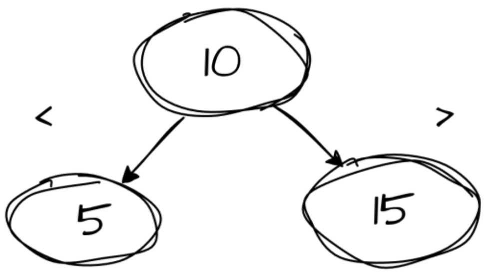
insert newVal to the bucket

```
class Student {

    Attributes: sno, branch, section, name, dob .....

    public int hashcode() {
       use the key attributes to generate hashcode
    }

    public boolean equals(Object other) {
        use the key attributes to check for equality
    }

}
```

Overriding hashcode() and equals() allows Student class objects to work with Hashed collections.

BUT NOT FOR TREE COLLECTIONS AS TREE COLLECTIONS REQUIRE COMPARISON LOGIC AND NOT EQUALITY.

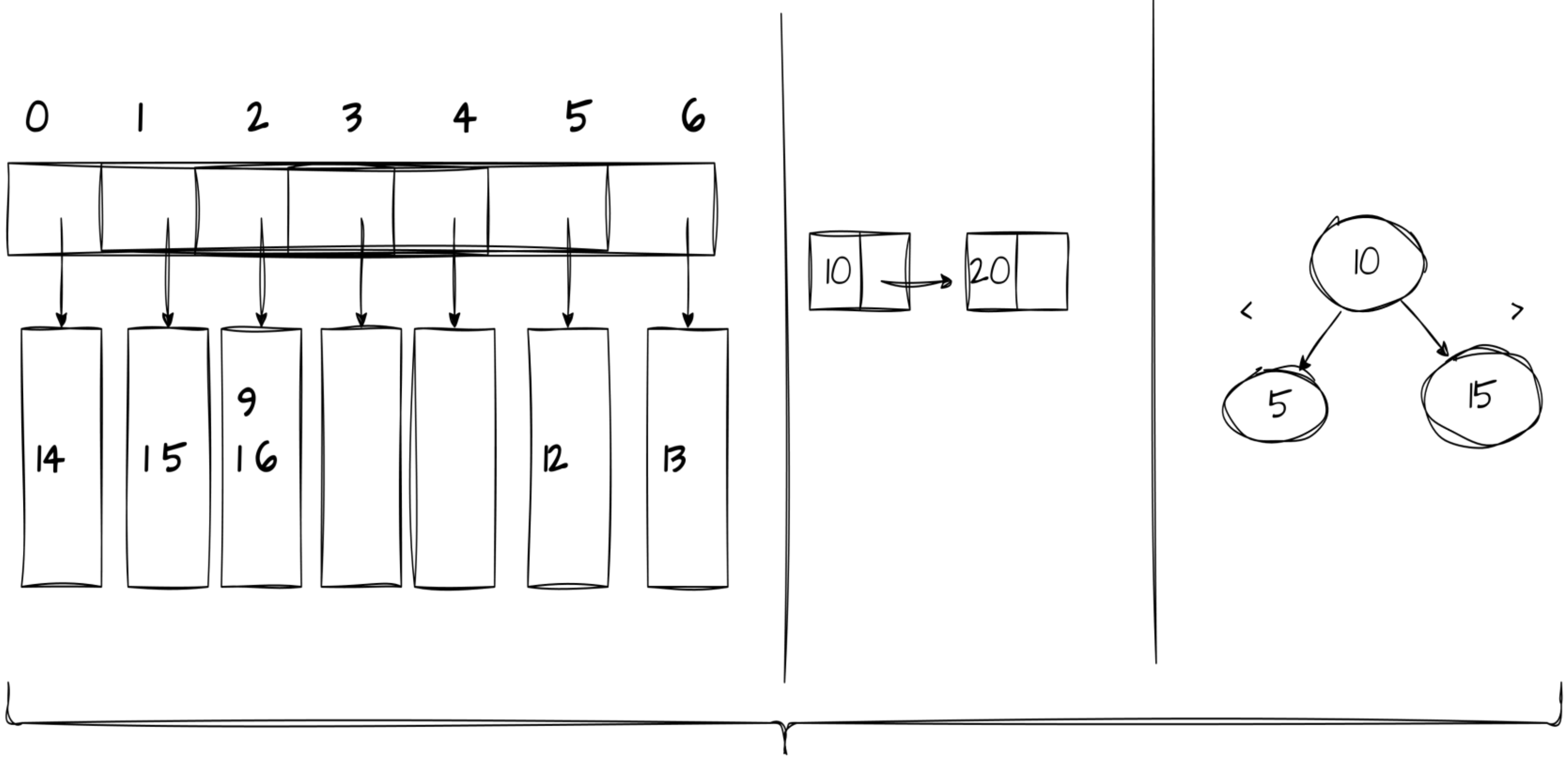
TREE : SORTED




```
class Student implements Comparable {

    Attributes: sno, branch, section, name, dob .....

    public int compareTo(Student other) {
        return -1 if this student is < other student
        return +1 if this student is > other student
        return 0 if equal.
    }
}
```

Or you can provide your own **Comparator**

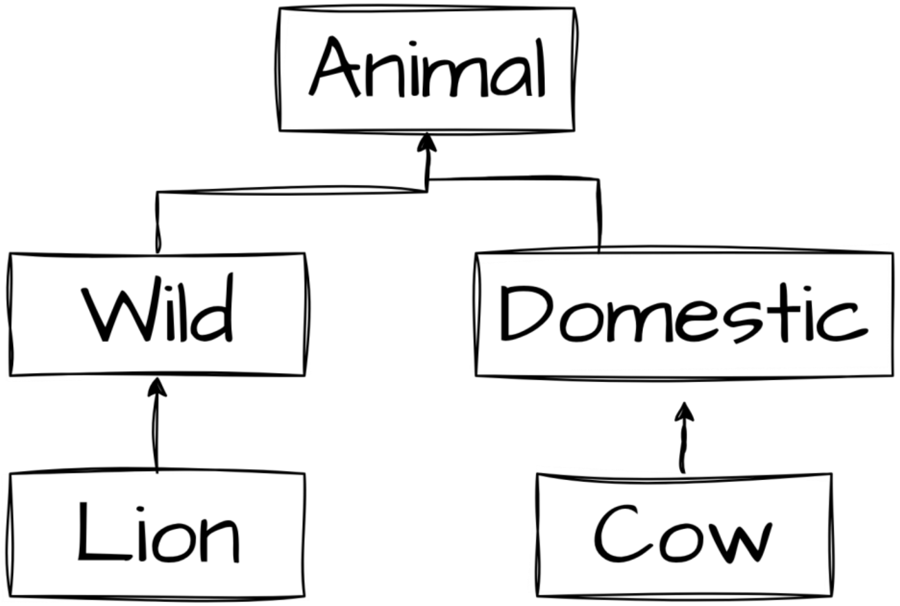
Iterator




```
Iterator  i = col.iterator();

while ( i.hasNext() ) {

   Object obj = i.next();

   ....

}
```

```
// Further simplified using
// for each

for (Object obj : col ) {
   ...
}
```

Type Safety (Problem)



Animal a = ...;
Dog d = (Dog) a;

Can an animal be a Dog?

Compiler: OK (as it may be)

Runtime:

If it is Dog => OK it works

If NOT => ERROR (ClassCastException)

Type Safety (Problem2)

List someList = new LinkedList() ;

someList.add(10);

Double d = (Double) someList.get(0);

Compiler: OK

Runtime : ClassCastException as Integer can be cast to Double

Can we identify these kind of problems during Compilation ???

Type Safety (Solution)

```
List<Integer>  someList = new LinkedList<Integer>() ;

someList.add( 10 );

Double d = (Double) someList.get(0);
```

Compiler: REJECTS as Integer cannot be cast to Double.

Generic Types

```
class Sample<T> {
      T obj;

      void set(T value) {
         obj = value;
       }

       T  get() {
          return obj;
       }
}
```

```
// In this case T is Integer
Sample<Integer> s1 =
                 new Sample<Integer>();
s1.set(10);
Integer a = s1.get();

// In this case T is Double
Sample<Double> s2 =
                New Sample<Double>();
s2.set(10.2);
Double b = s2.get();   // NO TYPECAST NEEDED

// ERROR : during Compilation itself
Integer c = (Integer) s2.get();
```

Lambda – prerequisite

```
// Functional Interface: Interface with
// only one abstract method.

interface  Adder {
    int add(int a, int b);
}
```

```
// Implementation

class MyAdder implements Adder {
    public int add(int a, int b) {
        return   a+b;
    }
}
```

```
Adder obj = new MyAdder();
int x = obj.add(10, 20);
```

Lambda - prerequisite

```java
// Functional Interface: Interface with
// only one abstract method.

interface  Adder {
    int add(int a, int b);
}
```

// **Anonymous** implementation

```java
Adder obj = new Adder() {
    public int add(int a, int b) {
        return a + b;
    }
};

int x = obj.add(10, 20);
```

Lambda

```
// Functional Interface: Interface with
// only one abstract method.

interface  Adder {
    int add(int a, int b);
}
```

```
// Lambda: removes the verbosity
// simple and clean
// implementation of the abstract
// function

Adder obj = (a, b) -> a + b;

int x = obj.add(10, 20);
```

Streams

```
// Find the sum of the squares of all
// the even numbers in the list

int a[] = {10, 11, 12, 13, 14};
int s = 0;
for (int value : a ) {
  if (value % 2 == 0 ) {
      s +=  value * value;
  }
}

// print s.
```

Traditional imperative style

Functional Style using streams:

```
int a[] = {10, 11, 12, 13, 14};
int s = Arrays.stream(a)
              .filter( x -> x % 2 == 0)
              .map( x -> x * x )
              .reduce(0,  Integer::sum);
// print s
```

Streams

Functional programming emphasizes **declarative** operations (what to do) rather than **imperative** operations (how to do it).

Source: source of the stream

Intermediate operations: filter, map etc

Terminal operations: reduce, foreach, collect, sum etc.

Functional Style using streams:

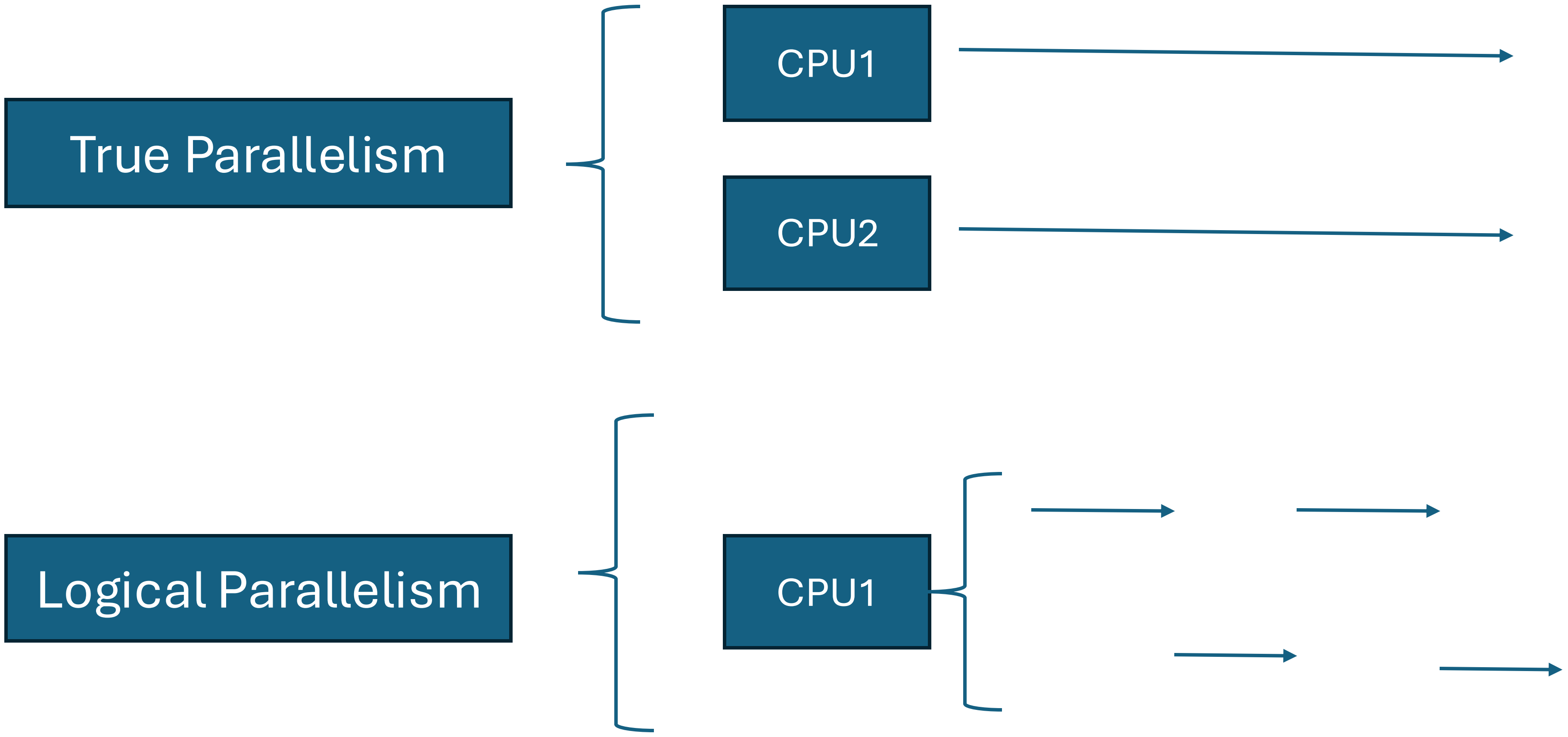
```
int a[] = {10, 11, 12, 13, 14};
int s = Arrays.stream(a)
                .filter( x -> x % 2 == 0)
                .map( x -> x * x )
                .reduce(0,  Integer::sum);
// print s
```

Streams

Example 2: To collect the squared even numbers as a list

```
int a[] = {10, 11, 12, 13, 14};

List<Integer> target = Arrays.stream(a)
                             .filter( x -> x % 2 == 0)
                             .map( x -> x * x )
                             .boxed()
                             .collect( Collectors.toList() );
// print s
```

Threads



Serial Execution

```
class Task {

    public void run() {
       for (int i = 1; i <= 1000; i++ )
           System.out.print("T");
    }
}
```

```
class Main {

   public static void main(String [] a) {

      Task t = new Task();
      t .run();

      for (int i = 1; i <= 1000; i++ )
         System.out.print("M");
   }
}
```

Output: TTTTTT...MMMMM.....

Parallel – Using Threads
Model 1

```
class Task extends Thread {

    public void run() {
  for (int i = 1; i <= 1000; i++ )
          System.out.print("T");
  }
}
```

```
class Main {

   public static void main(String [] a) {

      Task t = new Task();
      t .run();
      t .start();

      for (int i = 1; i <= 1000; i++ )
         System.out.print("M");
   }
}
```

Output: TTMMMTTMMTTMM…..

Note: If you are not seeing this kind of output, increase the number from 1000 to a bigger one.

Parallel – Using Threads Model 2

```
class Task implements Runnable {

    public void run() {
  for (int i = 1; i <= 1000; i++ )
          System.out.print("T");
  }
}
```

Output: TTMMTTMMTTMM.....

```
class Main {

   public static void main(String [] a) {

      Task tsk = new Task();
      Thread thr  = new Thread( tsk );
      thr.start();

      for (int i = 1; i <= 1000; i++ )
         System.out.print("M");
   }
}
```

Thread synchronization:

Need : -

Multiple threads modifying the state of the same object in parallel.

Solutions :-

- Synchronized methods
- Synchronized blocks
- Reentrant Lock (out of scope)

Thread safe code/ Re-entrant code :- code which is safe from concurrency issues.

Synchronized method

Locks the object for the duration of the function (withdraw)

```
class Account {
    ....
    ....
    public synchronized void withdraw( double amount) {
        ... // perform account update
    }
    ....
    ....
}
```

Synchronized block

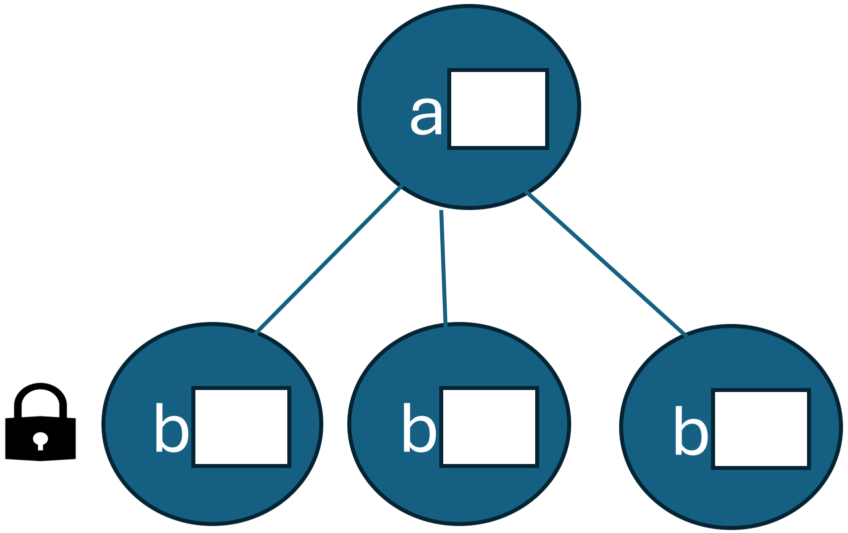
Locks the object for the duration of this block.

```
class Account {
    ….
    public void withdraw( double amount) {
        ….
        synchronized(this) {
            … // perform account update
        }
        ….
    }
    ….
}
```

Synchronizing class members

```
class Sample {

	static int a = 0;
	int b = 0;

	public void incr() {
		synchronized(this) {
			a++;  // NOT THREADSAFE
			b++;
		}
	}
}
```

Not Thread safe :-   a is a class member, not an instance member, here only instance (this) is locked here

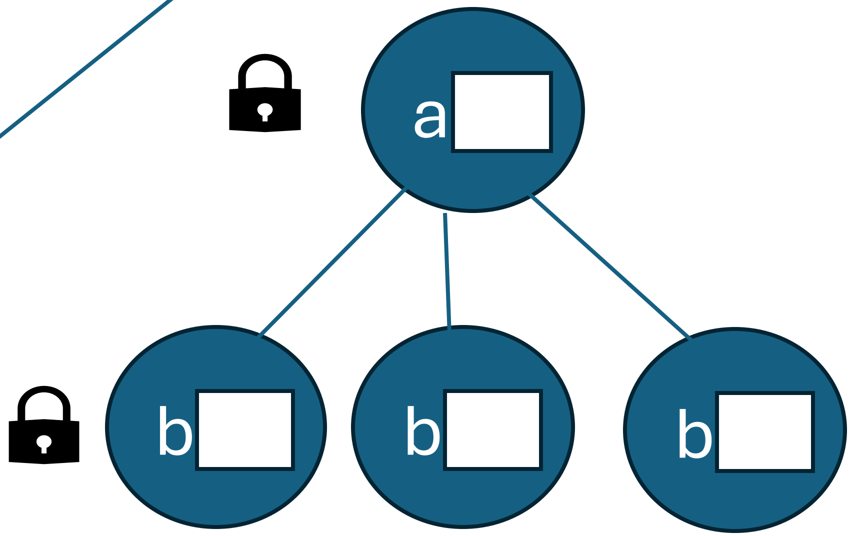


Synchronized class members

```
class Sample {
        static int a = 0;
        int b = 0;

       public void incr() {
              synchronized(Sample.class) {
                synchronized(this) {
                       a++;
                       b++;
                }
              }
       }
}
```

Locks Class object

Locks the object



SOLID Principles

Five Design Principles for Better Object-Oriented Code

S - Single Responsibility

O - Open/Closed

L - Liskov Substitution

I - Interface Segregation

D - Dependency Inversion

Single Responsibility Principle (SRP)

A class should have only one reason to change

Key Point: Each class should have one job or responsibility

Bad

```
class AccountService {

    void openAccount(Account a) {
        // save to database
        // send email
    }
    // save to db logic
    //  send EMAIL logic
}
```

Multiple responsibilities handled by single class such as saving to DB, sending notification etc.

Good

```
class AccountService {

    AccountRepository repo;
    NotificationService notifier;

    void openAccount(Account a) {
        repo.save(a);
        notifier.send(notification);
    }

}
```

AccountService : only workflow,
AccountRepository: saving to DB,
NotificationService: notifications.

Open/Closed Principle (OCP)

Open for extension, closed for modification

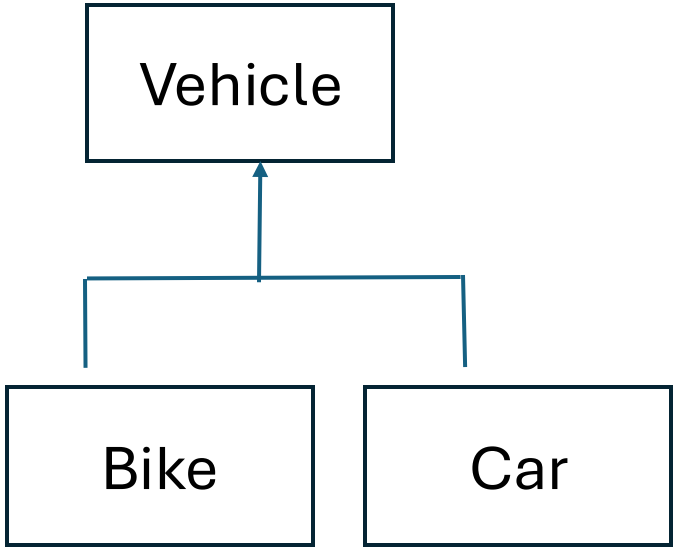
Key Point: Extend behavior without modifying existing code

```
Arrays.sort(T[] a, Comparator<? super T> c) {

    // sort logic is fixed and not modified.
    // comparison extendible through Comparator.

}
```

Liskov Substitution Principle (LSP)

Derived classes must be substitutable for their base classes

Key Point: Subclasses should enhance, not break parent behavior




```
class VechicleDemo {

  static void testDrive(Vehicle vehicle) {

      // Expects this to work reliably for all Vehicles
      vehicle.start();

      vehicle.stop();
  }
}
```

Interface Segregation Principle (ISP)

Clients shouldn't depend on unused methods

Instead of one large, "fat" interface, create multiple small, specific interfaces

```
interface Printable { void print(); }

interface Scannable { void scan(); }

class Printer implements Printable { }

class Scanner implements Scannable { }

class MultiFunctionPrinter implements Printable, Scannable { }
```

Dependency Inversion Principle (DIP)

Depend on abstractions, not concrete classes

BAD

MyMessenger → TCPProtocolHandler

(direct concrete dependency - VIOLATION)

GOOD

MyMessenger → ProtocolHandler

^

TCPProtocolHandler,
UDPProtocolHandler

MyMessenger depends on the ProtocolHandler abstraction, not on concrete implementations.