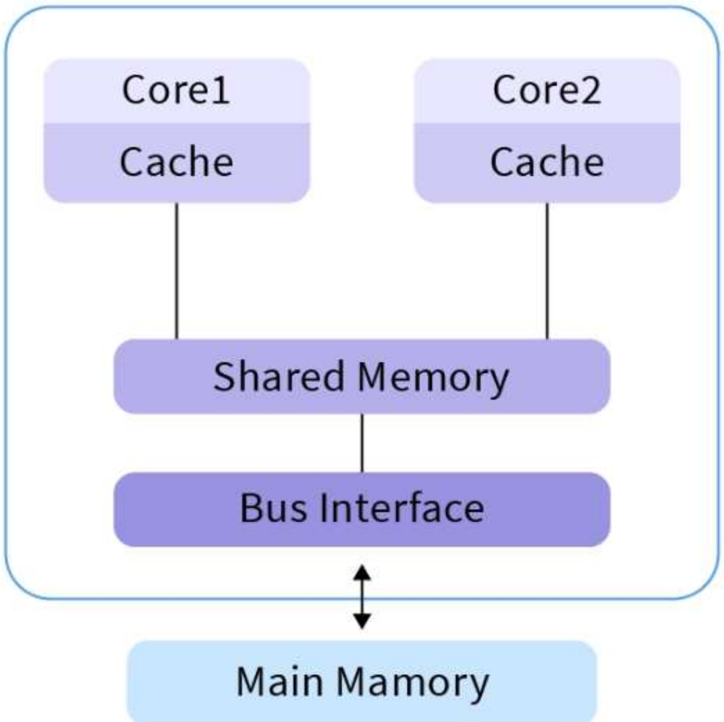
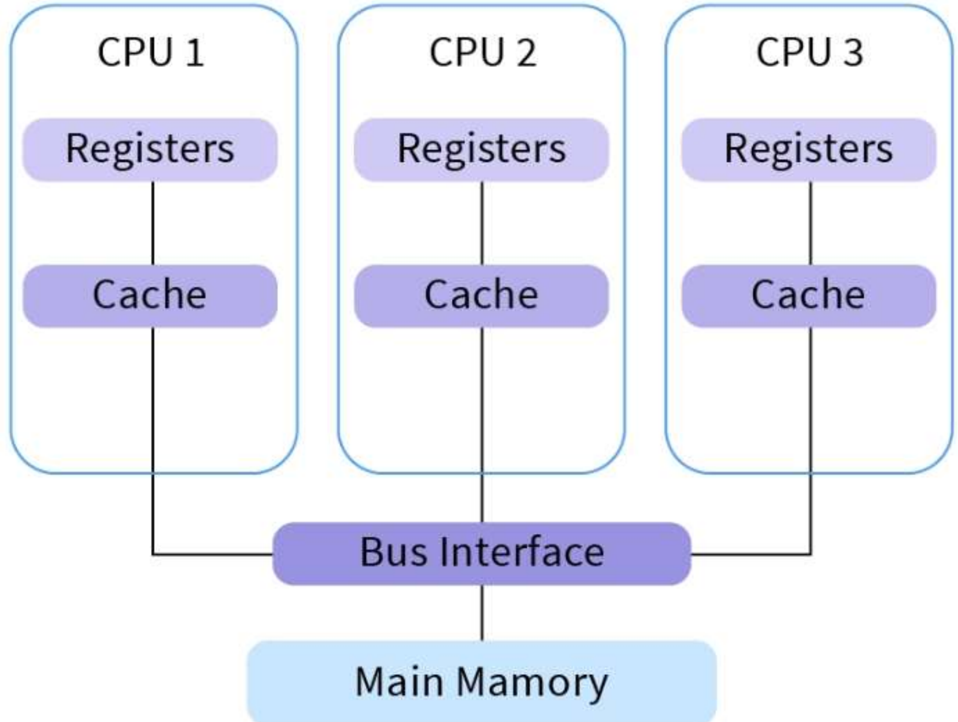
Department of COMPUTER SCIENCE AND ENGINEERING

# Multiprocessor Vs Multicore




Multicore processor




Multiprocessor

SC 57209254974 | P U-8822-2018 | VIDWAN 279509 | in drpsaikiran | 9490856188 | psaikiran@pvpsiddhartha.ac.in | Professor, Department of CSE, Prasad V Potluri Siddhartha Institute of Technology | Dr. P Sai Kiran

# Multiprocessor Vs Multicore

| Multicore | Multiprocessor |
| --- | --- |
| A multicore processor has a single processor with multiple cores that read and execute instructions. | A Multiprocessor has two or more processors that allow simultaneous processing of programs. |
| Multicore executes a single program faster. | Multiprocessor executes multiple programs faster. |
| Multicore processors are not as reliable as multiprocessors. | Multiprocessors are more reliable as multiple processors are available and failure of one processor will not affect the others. |
| Multicore has a simple configuration. | Multiprocessor requires complex configuration. |
| Multicore has less traffic. | Multiprocessor has more traffic. |
| Multicore is cheap as there is a single processor. | Multiprocessor is expensive compared to multicore. |

# Threads

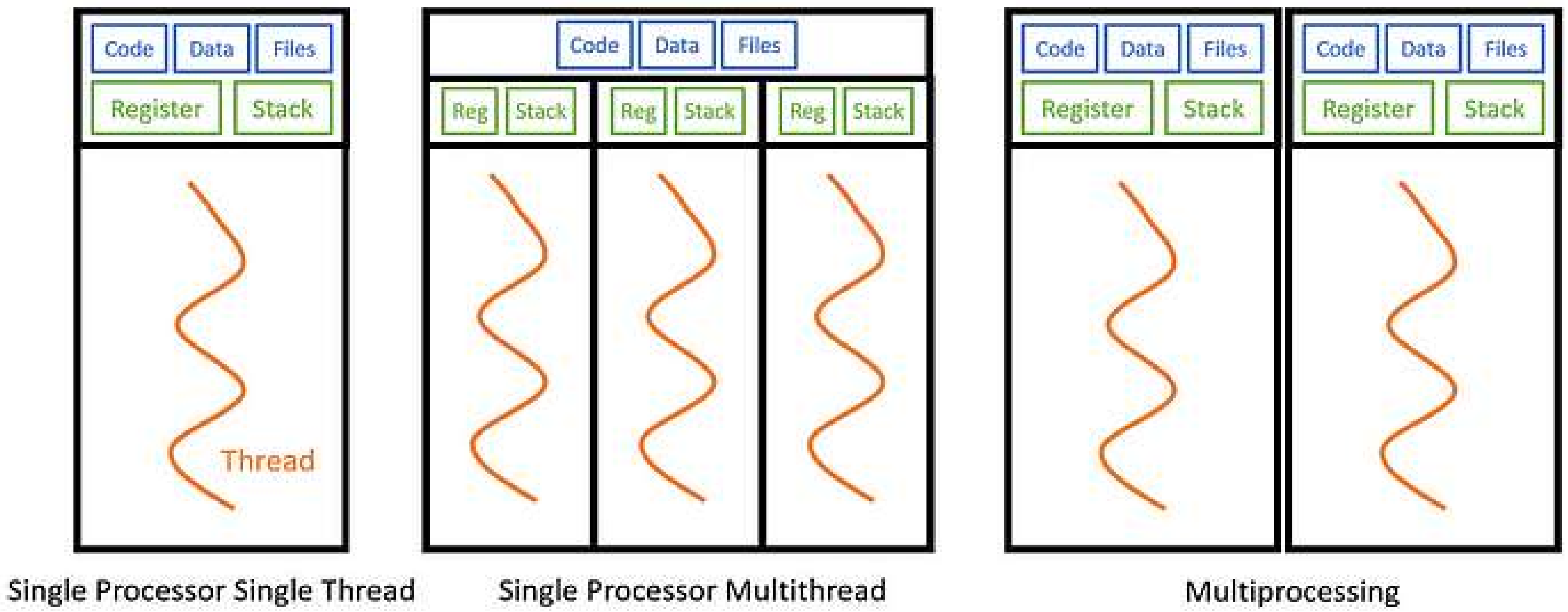
- **A process** is an instance of a program running on a computer. Each process has its memory space, system resources, and execution context.
  - web browser or a word processor.
  - Independent Projects in Different Departments.

- **A thread** is a smaller unit of execution within a process. Multiple threads within the process share the same memory space and resources, allowing for more efficient communication and data sharing. Threads are often referred to as "lightweight processes."
  - Team Members Working on a Single Project.

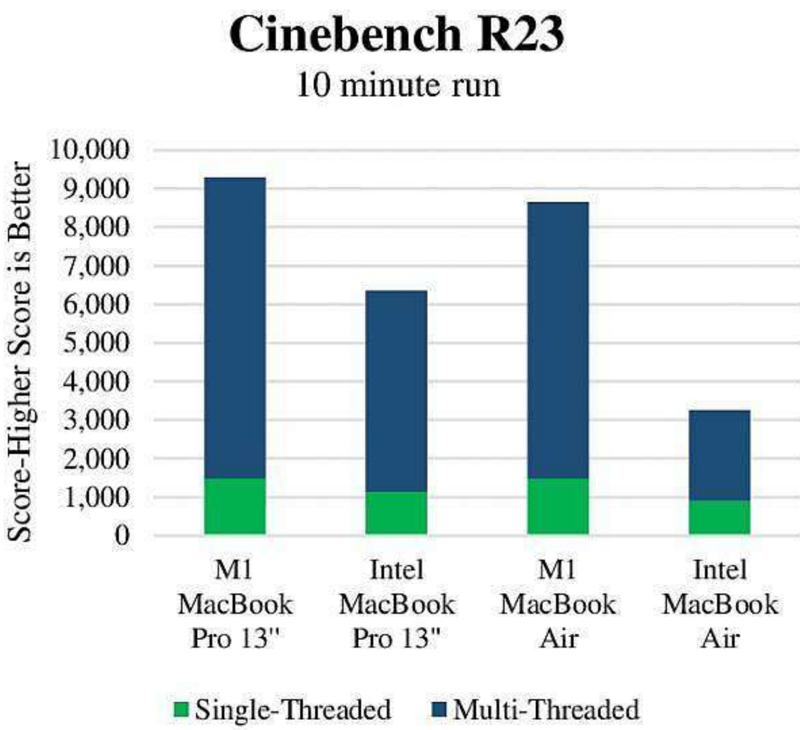
# Threads

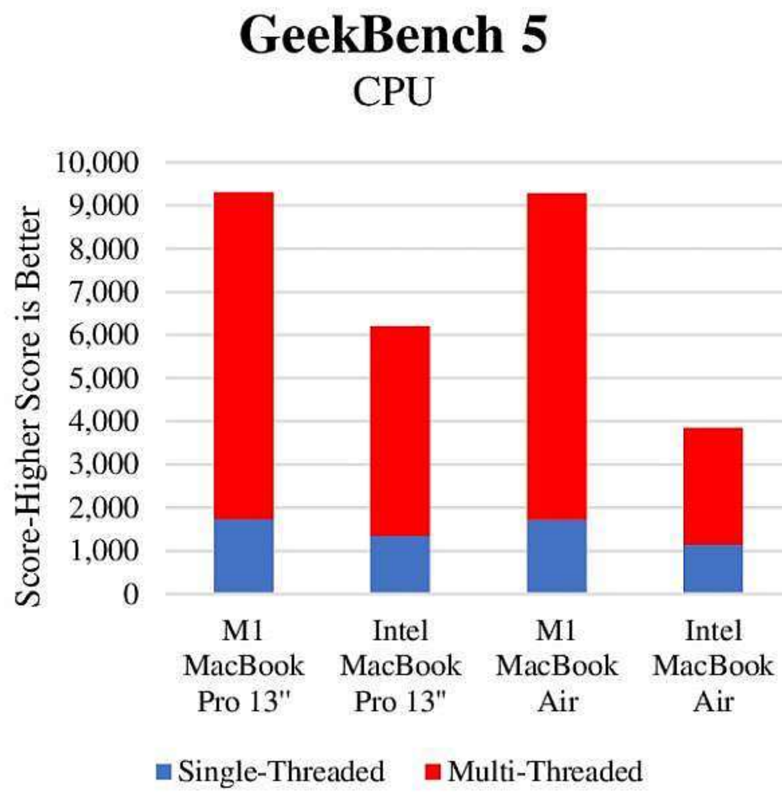
| Feature | Process | Thread |
|---|---|---|
| Resource ownership | Owns its own memory and resources | Shares memory and resources with the parent process |
| Lifetime | Independent lifetime | Dependent on the parent process lifetime |
| Creation | More expensive to create | Less expensive to create |
| Context switching | Expensive | Less expensive |
| Example | Independent projects in different departments | Team members working on a single project |

# Multitasking



# Multitasking



# Java's Multithreading Model

- Java has completely done away with the **event loop/ polling mechanism (Event loop/polling means- Executing one process after another which results in CPU time wastage)**

- In Java, All the libraries and classes are designed with multithreading in mind. This enables the entire system to be asynchronous.

- In Java the **java.lang.Thread** class is used to create thread-based code, imported into all Java applications by default.

# Java's Multithreading Model

- The **Thread** class has two primary thread control methods:
  - **public void start() : The start() method starts a thread execution**
  - **public void run() : The run() method actually performs the work of the thread and is the entry point for the thread**
- The thread *dies* when the **run( )** method terminates
- You never call **run( )** explicitly
- The **start( )** method called on a thread automatically initiates a call to the thread's **run( )** method

# Java's Thread State Diagram



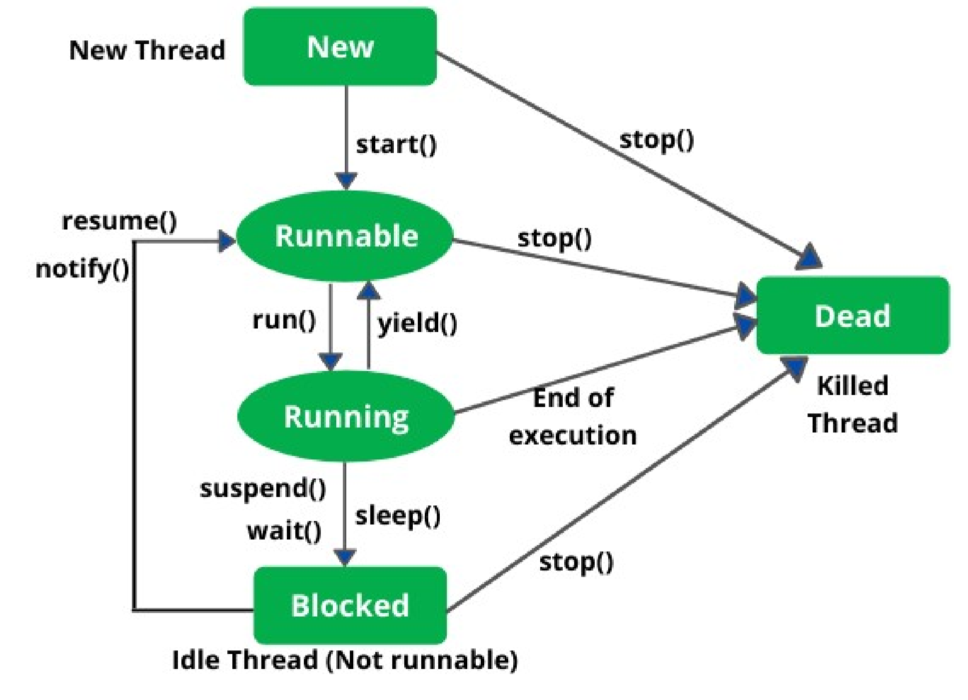



Fig: State Transition Diagram of a Thread

# Creating a Thread

- A thread can be created by instantiating an object of type **Thread**.
- This can be achieved in any of the following two ways :

1. **Implementing the Runnable interface**
2. **Extending the Thread class**

57209254974 | U-8822-2018 | VIDWAN 279509 | drpsaikiran | 9490856188 | psaikiran@pvpsiddhartha.ac.in | Professor, Department of CSE, Prasad V Potluri Siddhartha Institute of Technology | Dr. P Sai Kiran

# Extending the Thread class

- We can also create Threads by extending the Thread class:

1. Instantiate the class that **extends Thread**
2. This class must override **run()** method
3. The **code that should run as a thread will be part of this run() method**
4. We must call the **start()** method on this thread
5. **start( ) in turn calls the thread's run( ) method**

# Extending the Thread class

```
//A very simple demo for creating threads by extending Thread class:-

public class ThreadDemo extends Thread{
public void run(){
   for(int counter=1;counter<=100;counter++){
      System.out.println("thread is running..."+counter);
     }
  }
public static void main(String args[]){
    ThreadDemo threadDemo=new ThreadDemo();
    threadDemo.start();
 }
}
```

SC 57209254974 | P U-8822-2018 | VIDWAN 279509 | in drpsaikiran | 9490856188 | psaikiran@pvpsiddhartha.ac.in | Professor, Department of CSE, Prasad V Potluri Siddhartha Institute of Technology | Dr. P Sai Kiran

# Creating Threads: Implementing Runnable

- Thread can be created by creating a class that implements Runnable interface.
    - `class DemoThread implements Runnable{}`
- After defining the class that implements Runnable, we have to **create an object of type Thread** from within the object of that class.
    - This thread will end when run( ) returns or terminates.
- The **Thread class** defines several **constructors** one of which is:
    - **Thread(Runnable threadOb, String threadName)**

# Creating Threads: Implementing Runnable

Department of
COMPUTER SCIENCE
AND ENGINEERING

```
public class ThreadDemo implements Runnable {

public void run() {
  for(int counter=1;counter<=100;counter++){
      System.out.println("thread is running..."+counter);
     }
  }

public static void main(String args[])
  {
         ThreadDemo threadDemo = new ThreadDemo();
         Thread t1 = new Thread(threadDemo);
         t1.start();
   }
}
```

# The main Thread

- When a Java program starts executing:
  - the main thread begins running
  - the main thread is immediately created when main() commences execution
- Information about the main or any thread can be accessed by obtaining a reference to the thread using a public, static method in the Thread class called **currentThread( )**

# Extending the Thread class

```
public class ThreadInfo {
        public static void main(String args[]) {
                Thread t = Thread.currentThread( );
                System.out.println("Current Thread :" +t);
                t.setName("Demo Thread");
                System.out.println("New name of the thread :" +t);
                try {
                        Thread.sleep(1000);
                }
                catch (InterruptedException e) {
                        System.out.println("Main Thread Interrupted");
                }
        }
}
```

SC 57209254974 | P U-8822-2018 | VIDWAN 279509 | in drpsaikiran | 9490856188 | psaikiran@pvpsiddhartha.ac.in | Professor, Department of CSE, Prasad V Potluri Siddhartha Institute of Technology | Dr. P Sai Kiran

# Control Thread Execution

- Two ways exist by which you can determine whether a thread has finished:

- The **isAlive( )** method will return **true** if the thread upon which it is called is still running; else it will return **false**

- The **join()** method waits until the thread on which it is called terminates.

- **Syntax:**
- final boolean isAlive()
- final void join() throws InterruptedException

57209254974 | U-8822-2018 | VIDWAN 279509 | drpsaikiran | 9490856188 | psaikiran@pvpsiddhartha.ac.in | Professor, Department of CSE, Prasad V Potluri Siddhartha Institute of Technology | Dr. P Sai Kiran

# Thread Priorities

- Every thread has a priority
- When a thread is created it inherits the priority of the thread that created it
- The methods for accessing and setting priority are as follows:
- **public final int getPriority();**
- **public final void setPriority(int level);**

# Synchronization



- It is normal for threads to be sharing objects and data
- Different threads shouldn’t try to access and change the same data at the same time
- Threads must therefore be synchronized
- There is a need for a mechanism to ensure that the shared data will be used by only one thread at a time
- This mechanism is called synchronization.

# Synchronization

```
class Account {
        int balance;
        public Account(){
                balance=5000;
        }
        public void withdraw(int bal){
          balance= balance-bal;
          System.out.println("Balance remaining:::" +
                balance);
        }
}
```

# Synchronization



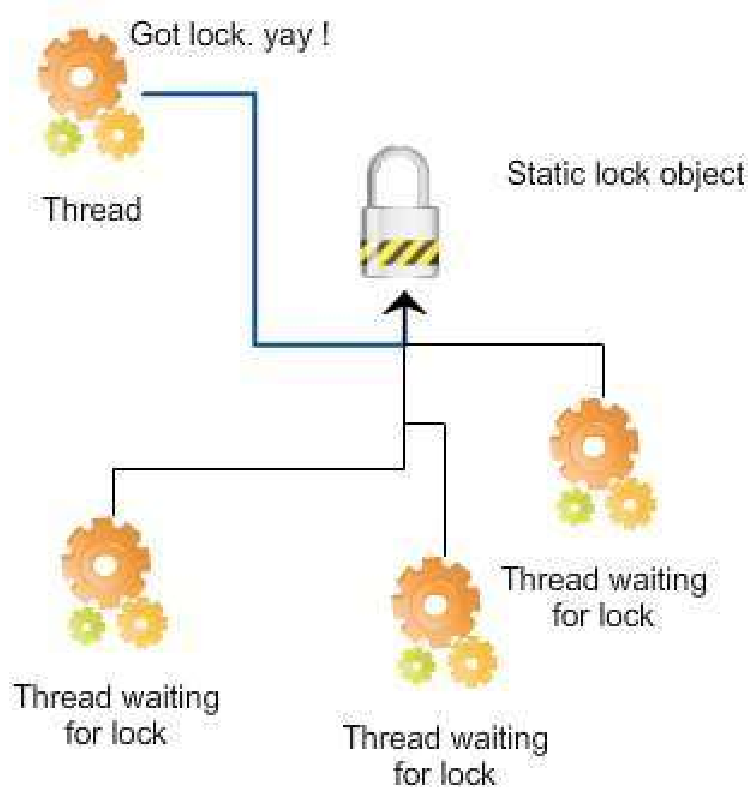
```
class C implements Runnable{
Account obj;
public C(Account a)
  {
     obj=a;
   }
public void run()
  {
     obj.withdraw(500);
  }
}
```

```
class SynchEx{
public static void main (String args[]
)
{
  Account a1=new Account();
  C c1=new C(a1);
  Thread t1=new Thread(c1);
  Thread t2=new Thread(c1);
  t1.start();
  t2.start();
}
}
```

SC 57209254974 | P U-8822-2018 | VIDWAN 279509 | in drpsaikiran | 9490856188 | psaikiran@pvpsiddhartha.ac.in | Professor, Department of CSE, Prasad V Potluri Siddhartha Institute of Technology | Dr. P Sai Kiran

# Synchronization

- When one thread acquires the lock on the shared resource, the other threads need to wait for the lock to be released.



# Synchronization

- Synchronization can be applied to:
- A method
  - **public synchronized withdraw(){…}**
- A block of code
  - **synchronized (objectReference){…}**
- Synchronized methods in subclasses use same locks as their superclasses

# Inter Thread Communication

- Threads are often interdependent - one thread depends on another thread to complete an operation, or to service a request
- The words wait and notify encapsulate the two central concepts to thread communication
- A thread waits for some condition or event to occur. You notify a waiting thread that a condition or event has occurred.

# Inter Thread Communication

- wait( ) directs the calling thread to surrender the monitor, and go to sleep until some other thread enters the monitor of the same object, and calls notify( )

- notify( ) wakes up the other thread which was waiting on the same object(that had called wait() previously on the same object)

# Inter Thread Communication



- Consider the situation, where one thread(Developer) is producing some data, and another thread(Client) is consuming it.

- Suppose that the Developer has to wait until the Client has completed reading before it produces more data. In a polling system, the Client would waste many CPU cycles while it waits for the Developer to produce.

SC 57209254974 | P U-8822-2018 | VIDWAN 279509 | in drpsaikiran | 9490856188 | psaikiran@pvpsiddhartha.ac.in | Professor, Department of CSE, Prasad V Potluri Siddhartha Institute of Technology | Dr. P Sai Kiran

# Inter Thread Communication

```
class QueueClass {
        int number;

        synchronized int get( ) {
                System.out.println( "Got: " + number);
                return number;
        }

        synchronized void put( int number) {
                this.number = number;
                System.out.println( "Put: " + number);
        }
}
```

# Inter Thread Communication

```
class Developer implements Runnable {
        QueueClass queueClass;

        Developer ( QueueClass queueClass) {
                this.queueClass = queueClass;
        }

        public void run( ) {
                int i = 0;
                for(int j=0; j<50; j++) {
                        queueClass.put (i++);
                }
        }
}
```

# Inter Thread Communication

```
class Client implements Runnable {
        QueueClass queueClass;

        Client (QueueClass queueClass) {
                this.queueClass = queueClass;
        }

        public void run( ) {
                for(int j=0; j<50; j++)  {
                        queueClass.get( );
                }
        }

}
```

SC 57209254974 | P U-8822-2018 | VIDWAN 279509 | in drpsaikiran | 9490856188 | psaikiran@pvpsiddhartha.ac.in | Professor, Department of CSE, Prasad V Potluri Siddhartha Institute of Technology | Dr. P Sai Kiran

# Inter Thread Communication

```
public class Caller {

        public static void main(String args[]) {
                QueueClass queueClass = new QueueClass( );
                Developer Dev = new Developer(queueClass);
                Client cl = new Client (queueClass);
                Thread t1=new Thread(Dev);
                Thread t2=new Thread(cl);
                t1.start();
                t2.start();
                System.out.println("Press Control-C to stop");
        }
}
```

# wait() and notify()

```
class QueueClass {
        int number;
        boolean valueset = false;
```

```
    synchronized void put( int number) {
            if (valueset==true)
                    try {
                            wait( );
                    }
                    catch (InterruptedException e) {
                            System.out.println("InterruptedException caught");
                    }
            this.number = number;
            valueset = true;
            System.out.println( "Put: " + number);
            notify( );
    }
}
```

# wait() and notify()

```
synchronized int get( ) {
        if (valueset==false)
          try {
                wait( );
              }
           catch (InterruptedException e) {
                System.out.println("InterruptedException caught");
              }
        System.out.println( "Got: " + number);
        valueset = false;
        notify( );
        return number;
}
```

# Java Collections

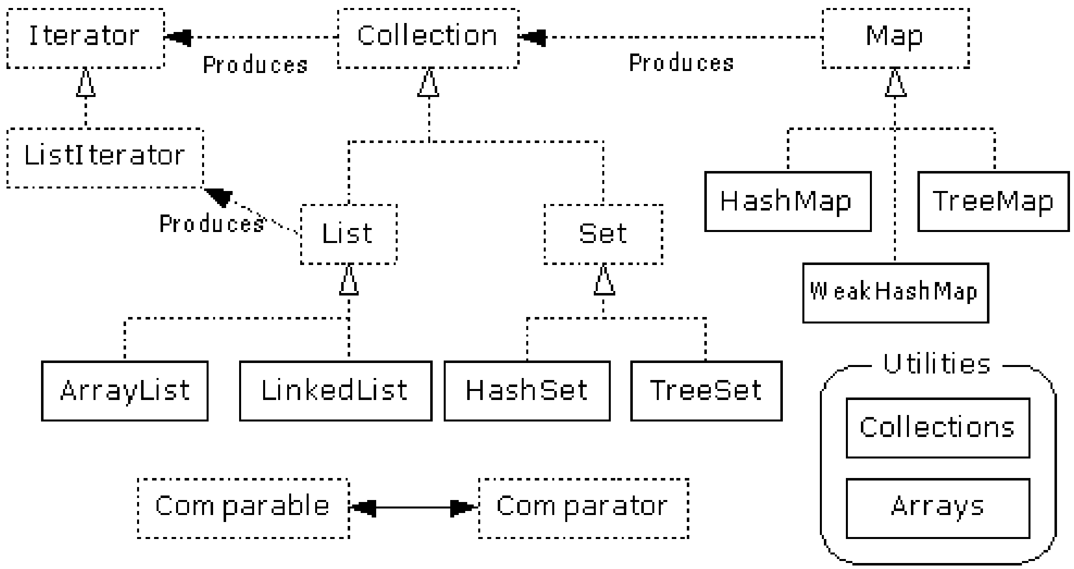
# Java 2 Collections

- A collection is an object that groups multiple elements into a single unit
- Very useful
  - store, retrieve and manipulate data
  - transmit data from one method to another
  - data structures and methods written by hotshots in the field
    - Joshua Bloch, who also wrote the Collections tutorial

# Collections Framework

- Unified architecture for representing and manipulating collections.
- A collections framework contains three things
  - Interfaces
  - Implementations
  - Algorithms

# Collections Framework Diagram




- Interfaces, Implementations, and Algorithms
- From Thinking in Java, page 462

# Collection Interface

- Defines fundamental methods
  - `int size();`
  - `boolean isEmpty();`
  - `boolean contains(Object element);`
  - `boolean add(Object element);    // Optional`
  - `boolean remove(Object element); // Optional`
  - `Iterator iterator();`
- These methods are enough to define the basic behavior of a collection
- Provides an Iterator to step through the elements in the Collection

# Iterator Interface

- Defines three fundamental methods
  - `Object next()`
  - `boolean hasNext()`
  - `void remove()`
- These three methods provide access to the contents of the collection
- An Iterator knows position within collection
- Each call to next() "reads" an element from the collection
  - Then you can use it or remove it

# Iterator Position

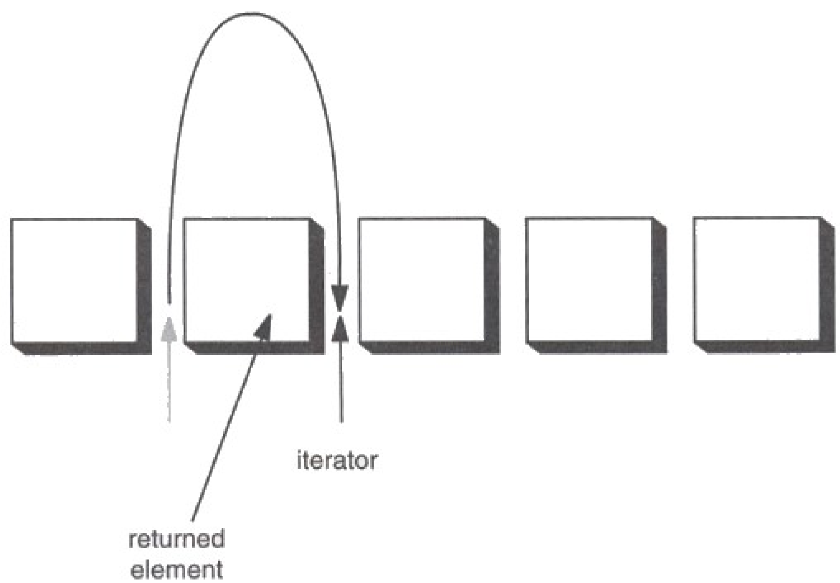
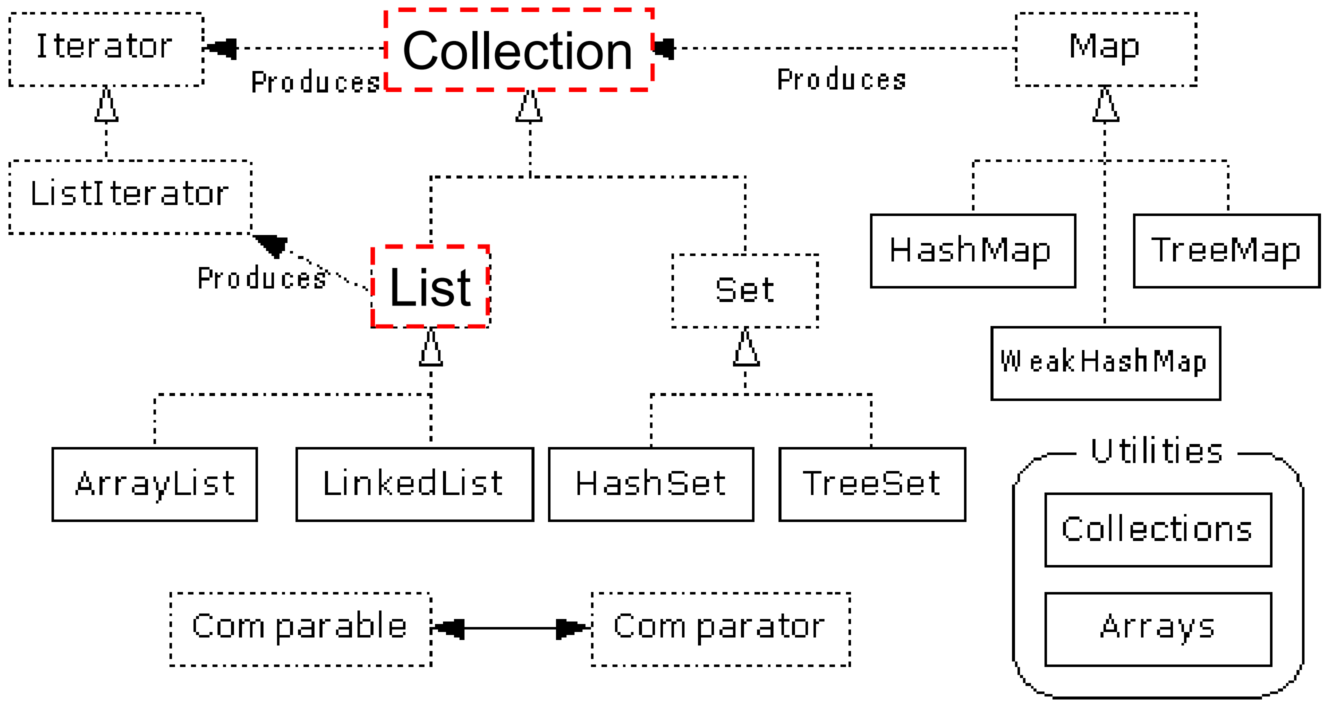



**Figure 2–3: Advancing an iterator**

# Example - SimpleCollection

```
public class SimpleCollection  {
  public static void main(String[] args) {
    Collection c;
    c = new ArrayList();
    System.out.println(c.getClass().getName());
    for (int i=1; i <= 10; i++) {
           c.add(i + " * " + i + " = "+i*i);
    }
    Iterator iter = c.iterator();
    while (iter.hasNext())
           System.out.println(iter.next());
  }
}
```

# List  Interface Context



# List Interface

- The List interface adds the notion of *order* to a collection
- The user of a list has control over where an element is added in the collection
- Lists typically allow *duplicate* elements
- Provides a ListIterator to step through the elements in the list.

# ListIterator Interface

- Extends the Iterator interface
- Defines three fundamental methods
  - **`void add(Object o)`** - before current position
  - **`boolean hasPrevious()`**
  - **`Object previous()`**
- The addition of these three methods defines the basic behavior of an ordered list
- A ListIterator knows position within list

# Iterator Position - `next()`, `previous()`

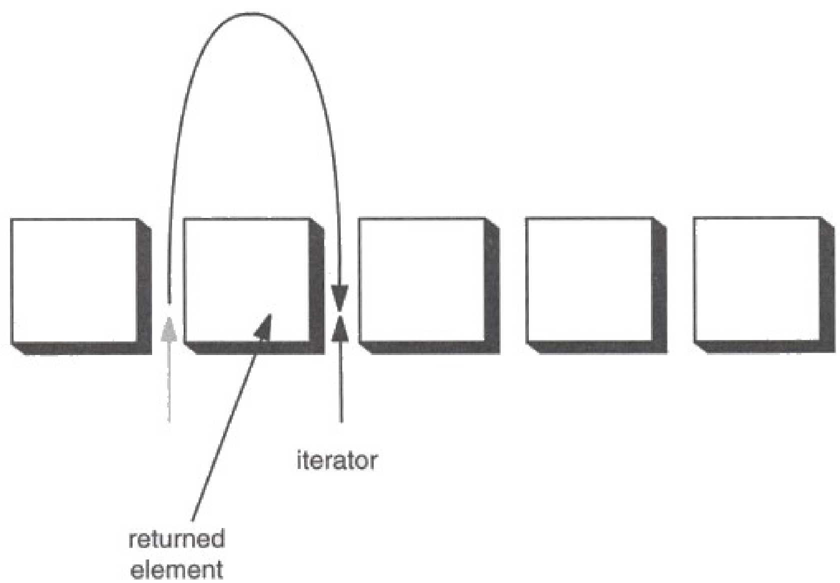
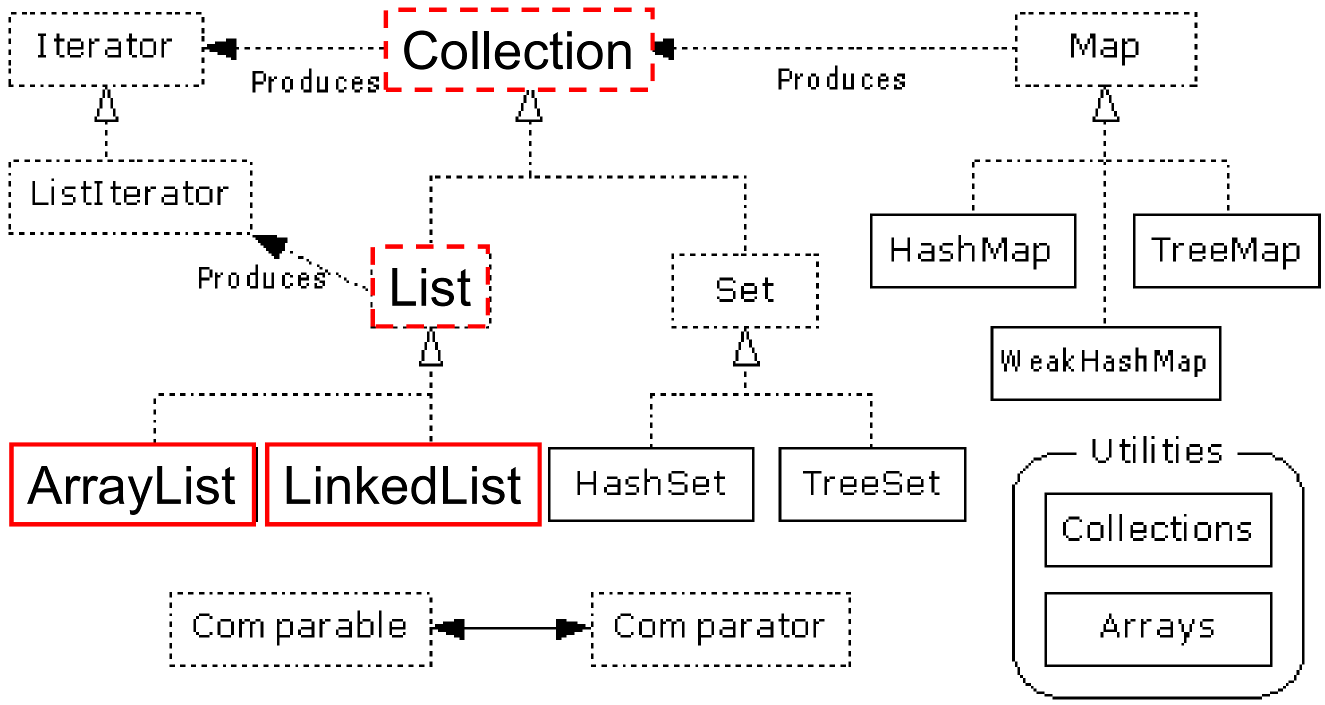



**Figure 2–3: Advancing an iterator**

# ArrayList and LinkedList Context



# List Implementations

- ArrayList
  - low cost random access
  - high cost insert and delete
  - array that resizes if need be
- LinkedList
  - sequential access
  - low cost insert and delete
  - high cost random access

# ArrayList overview

- Constant time positional access (it's an array)
- One tuning parameter, the initial capacity

```
public ArrayList(int initialCapacity) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException(
                "Illegal Capacity: "+initialCapacity);
    this.elementData = new Object[initialCapacity];
}
```

# ArrayList methods

- The indexed get and set methods of the List interface are appropriate to use since ArrayLists are backed by an array
  - **`Object get(int index)`**
  - **`Object set(int index, Object element)`**
- Indexed add and remove are provided, but can be costly if used frequently
  - **`void add(int index, Object element)`**
  - **`Object remove(int index)`**
- May want to resize in one shot if adding many elements
  - **`void ensureCapacity(int minCapacity)`**

# LinkedList overview

- Stores each element in a node
- Each node stores a link to the next and previous nodes
- Insertion and removal are inexpensive
  - just update the links in the surrounding nodes
- Linear traversal is inexpensive
- Random access is expensive
  - Start from beginning or end and traverse each node while counting

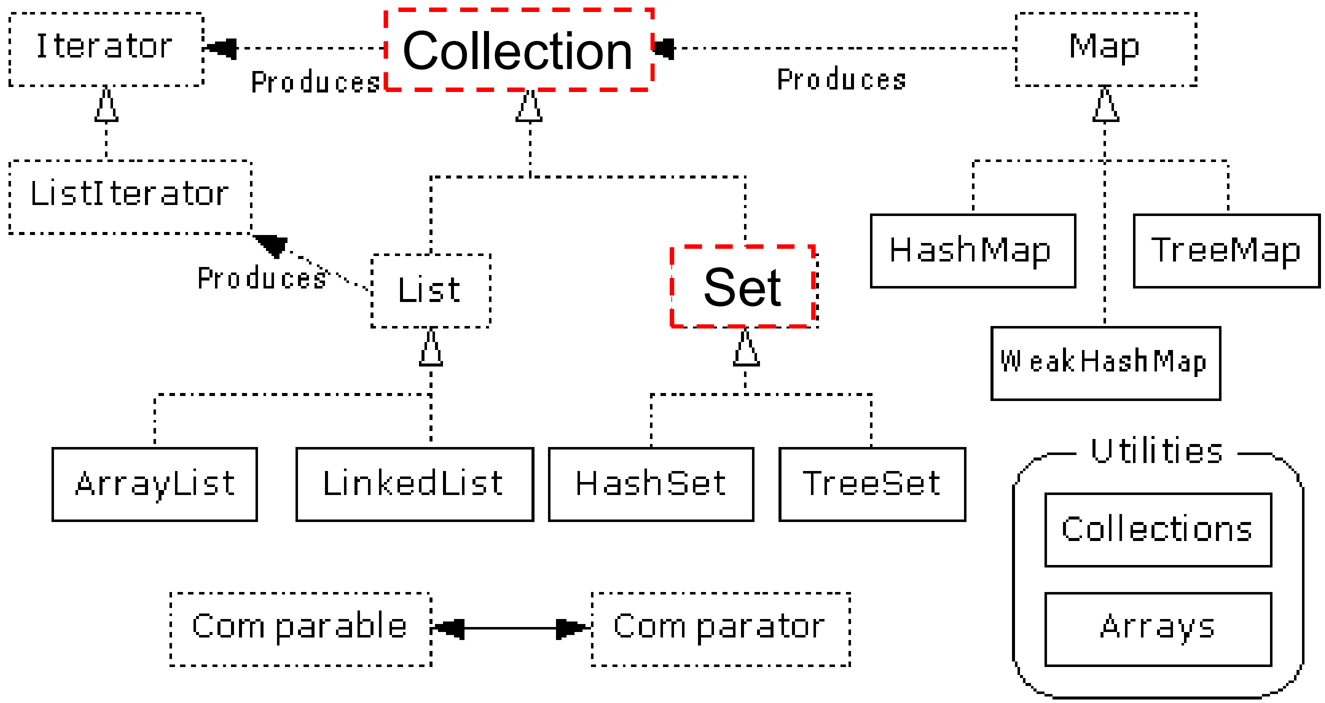
# LinkedList entries

```
private static class Entry {
  Object element;
  Entry next;
  Entry previous;

  Entry(Object element, Entry next, Entry previous) {
      this.element = element;
      this.next = next;
      this.previous = previous;
  }
}

private Entry header = new Entry(null, null, null);

public LinkedList() {
  header.next = header.previous = header;
}
```

# LinkedList methods

- The list is sequential, so access it that way
  - **`ListIterator listIterator()`**
- ListIterator knows about position
  - use **`add()`** from ListIterator to add at a position
  - use **`remove()`** from ListIterator to remove at a position
- LinkedList knows a few things too
  - **`void addFirst(Object o), void addLast(Object o)`**
  - **`Object getFirst(), Object getLast()`**
  - **`Object removeFirst(), Object removeLast()`**

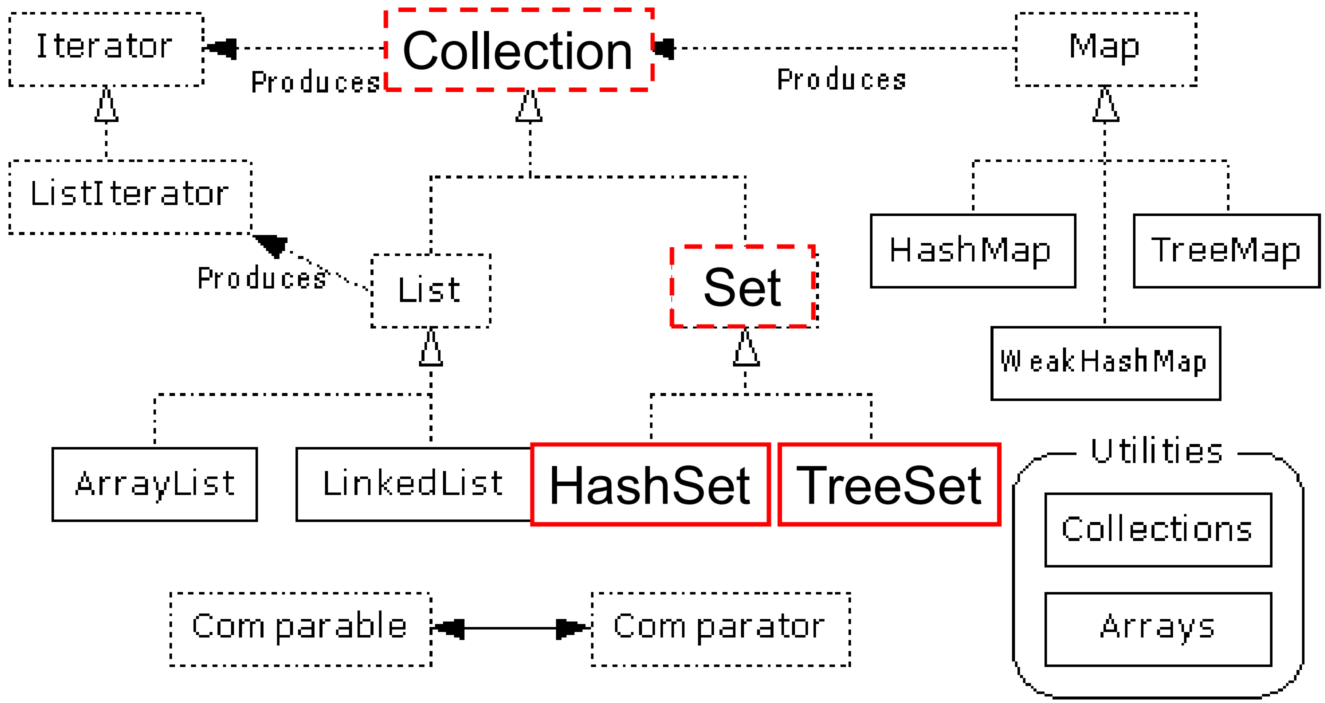
# Set  Interface Context



# Set Interface

- Same methods as Collection
  - different contract - no duplicate entries
- Defines two fundamental methods
  - **`boolean add(Object o)`** - reject duplicates
  - **`Iterator iterator()`**
- Provides an Iterator to step through the elements in the Set
  - No guaranteed order in the basic Set interface
  - There is a SortedSet interface that extends Set

# HashSet and TreeSet Context



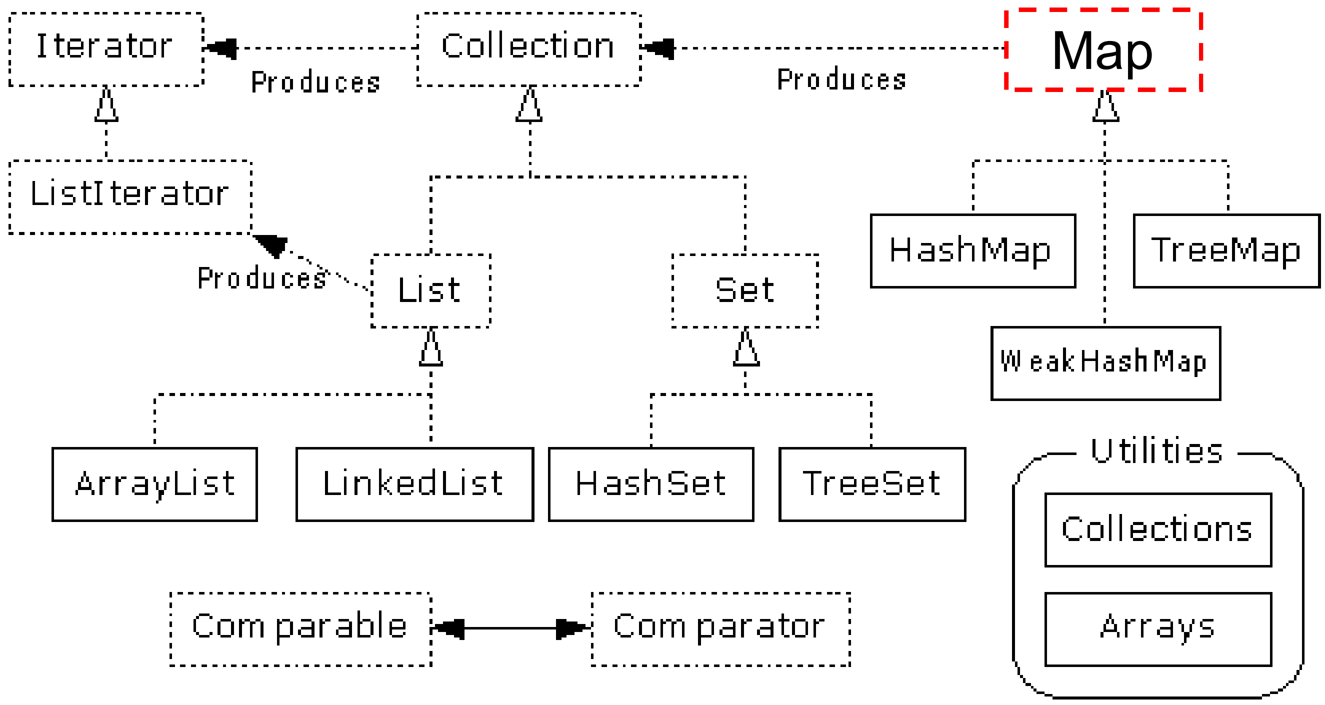
# HashSet

- Find and add elements very quickly
  - uses hashing implementation in HashMap
- Hashing uses an array of linked lists
  - The **`hashCode()`** is used to index into the array
  - Then **`equals()`** is used to determine if element is in the (short) list of elements at that index
- No order imposed on elements
- The **`hashCode()`** method and the **`equals()`** method must be compatible
  - if two objects are equal, they must have the same **`hashCode()`** value

# TreeSet

- Elements can be inserted in any order
- The TreeSet stores them in order
  - Red-Black Trees out of Cormen-Leiserson-Rivest
- An iterator always presents them in order
- Default order is defined by natural order
  - objects implement the Comparable interface
  - TreeSet uses **`compareTo(Object o)`** to sort
- Can use a different Comparator
  - provide Comparator to the TreeSet constructor

# Map  Interface Context



# Map Interface

- Stores key/value pairs
- Maps from the key to the value
- Keys are unique
  - a single key only appears once in the Map
  - a key can map to only one value
- Values do not have to be unique

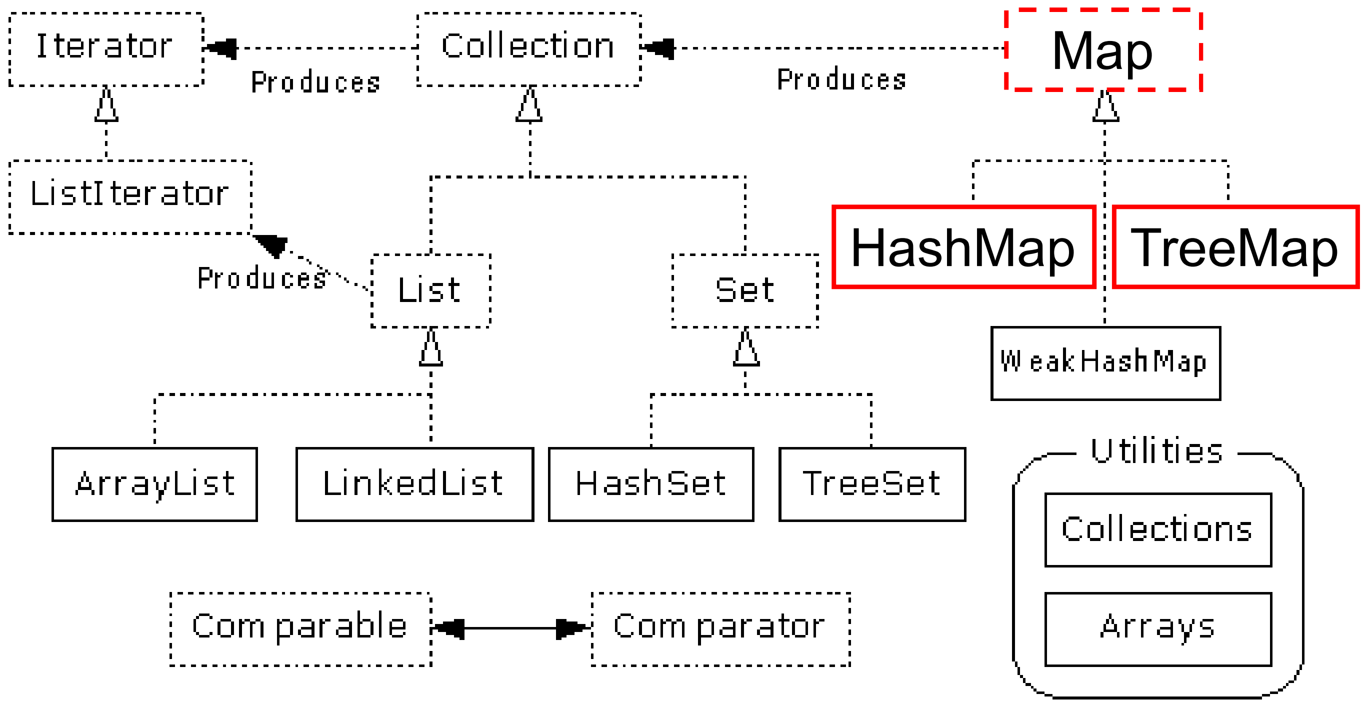
# Map methods

```
Object put(Object key, Object value)
Object get(Object key)
Object remove(Object key)
boolean containsKey(Object key)
boolean containsValue(Object value)
int size()
boolean isEmpty()
```

# Map views

- A means of iterating over the keys and values in a Map
- **`Set keySet()`**
  - returns the Set of keys contained in the Map
- **`Collection values()`**
  - returns the Collection of values contained in the Map. This Collection is not a Set, as multiple keys can map to the same value.
- **`Set entrySet()`**
  - returns the Set of key-value pairs contained in the Map. The Map interface provides a small nested interface called Map.Entry that is the type of the elements in this Set.

# HashMap and TreeMap Context



# HashMap and TreeMap

- HashMap
  - The keys are a set - unique, unordered
  - Fast

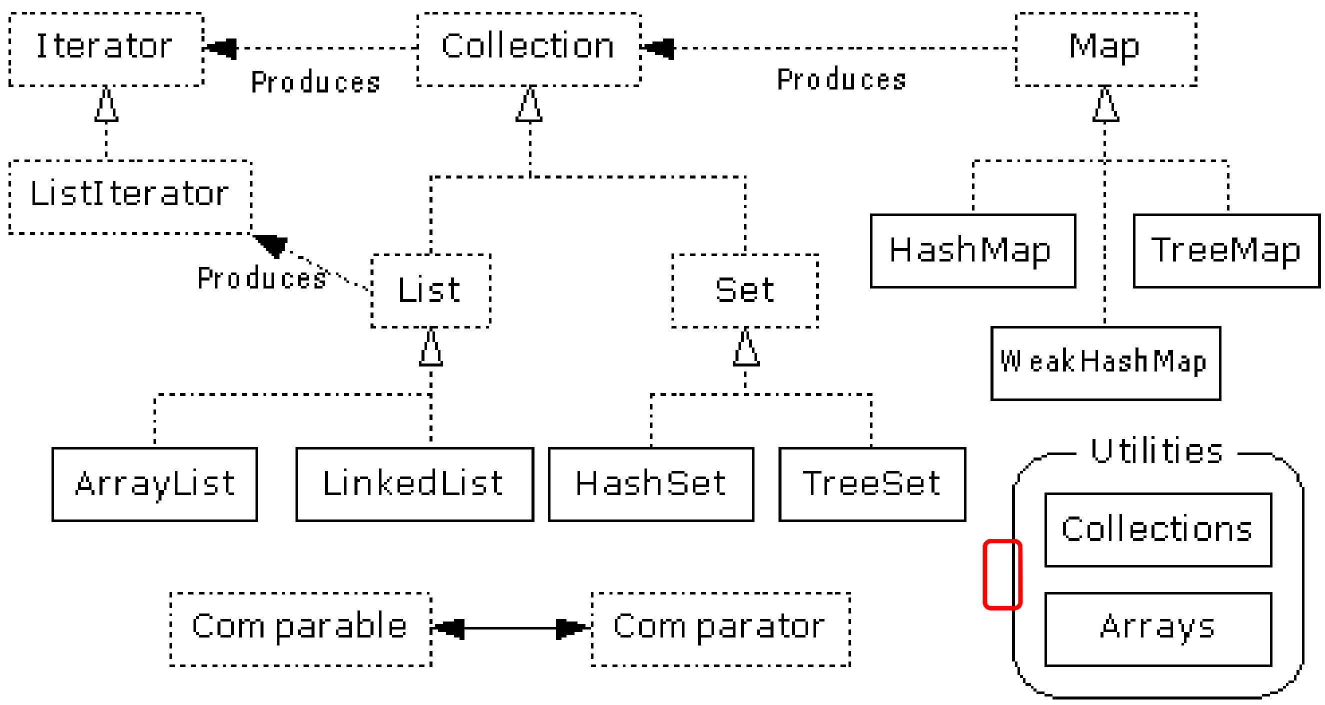
- TreeMap
  - The keys are a set - unique, ordered
  - Same options for ordering as a TreeSet
    - *Natural order (Comparable, compareTo(Object))*
    - *Special order (Comparator, compare(Object, Object))*

# Bulk Operations

- In addition to the basic operations, a Collection may provide “bulk” operations

```
boolean containsAll(Collection c);
boolean addAll(Collection c);    // Optional
boolean removeAll(Collection c); // Optional
boolean retainAll(Collection c); // Optional
void clear();                    // Optional
Object[] toArray();
Object[] toArray(Object a[]);
```

# Utilities Context



# Utilities

- The Collections class provides a number of static methods for fundamental algorithms
- Most operate on Lists, some on all Collections
  - Sort, Search, Shuffle
  - Reverse, fill, copy
  - Min, max
- Wrappers
  - synchronized Collections, Lists, Sets, etc
  - unmodifiable Collections, Lists, Sets, etc

# Legacy classes

- Still available
- Don’t use for new development
  - unless you have to, eg, J2ME, J2EE in some cases
- Retrofitted into Collections framework
- Hashtable
  - use HashMap
- Enumeration
  - use Collections and Iterators
  - if needed, can get an Enumeration with Collections.enumeration(Collection c)

# More Legacy classes

- Vector
  - use ArrayList
- Stack
  - use LinkedList
- BitSet
  - use ArrayList of boolean, unless you can't stand the thought of the wasted space
- Properties
  - legacies are sometimes hard to walk away from …
  - see next few pages

# Properties class

- Located in java.util package
- Special case of Hashtable
  - Keys and values are Strings
  - Tables can be saved to/loaded from file

# System properties

- Java VM maintains set of properties that define system environment
  - Set when VM is initialized
  - Includes information about current user, VM version, Java environment, and OS configuration

```
Properties prop = System.getProperties();
Enumeration e = prop.propertyNames();
while (e.hasMoreElements()) {
   String key = (String) e.nextElement();
   System.out.println(key + " value is " +
        prop.getProperty(key));
}
```