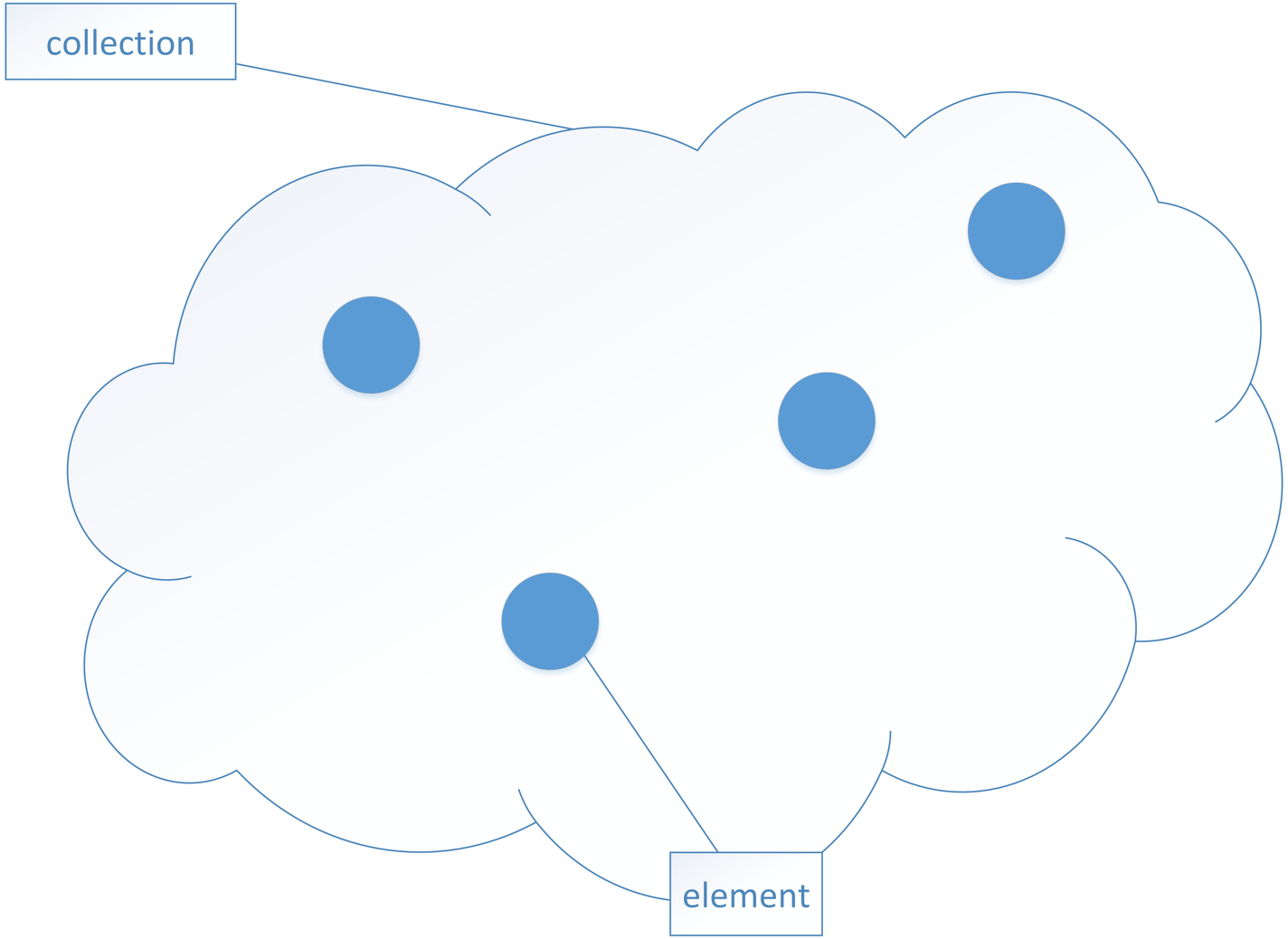
Programming Using Java

Session 9: Collections and Concurrency



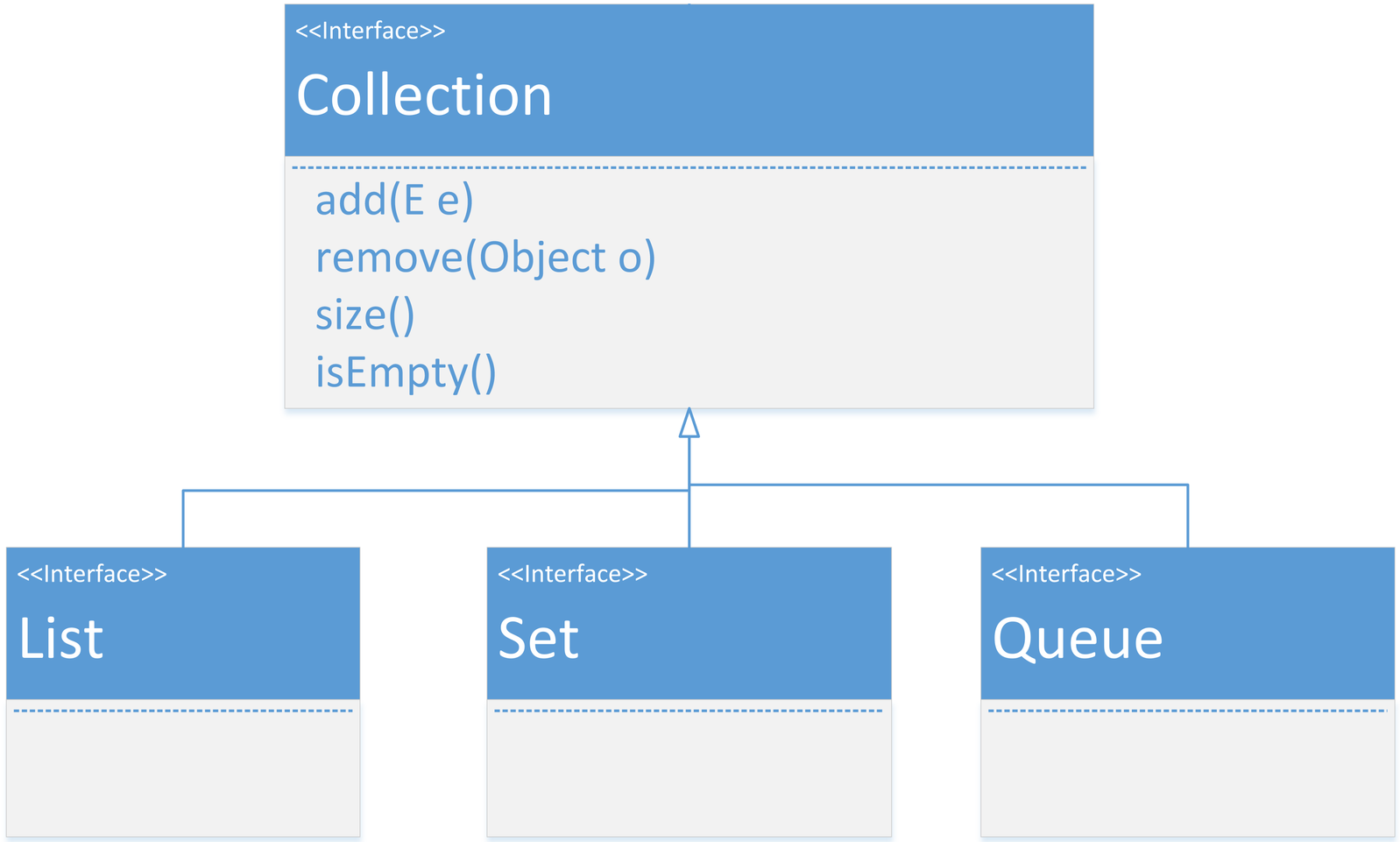
Collections

"…data structure that holds object references…"

Interface	Description
Collection	root interface
Set	contains no duplicates
List	ordered collection, can contain duplicates
Map	has keys associated with values, duplicate keys are disallowed
Queue	first-in, first-out, models waiting line

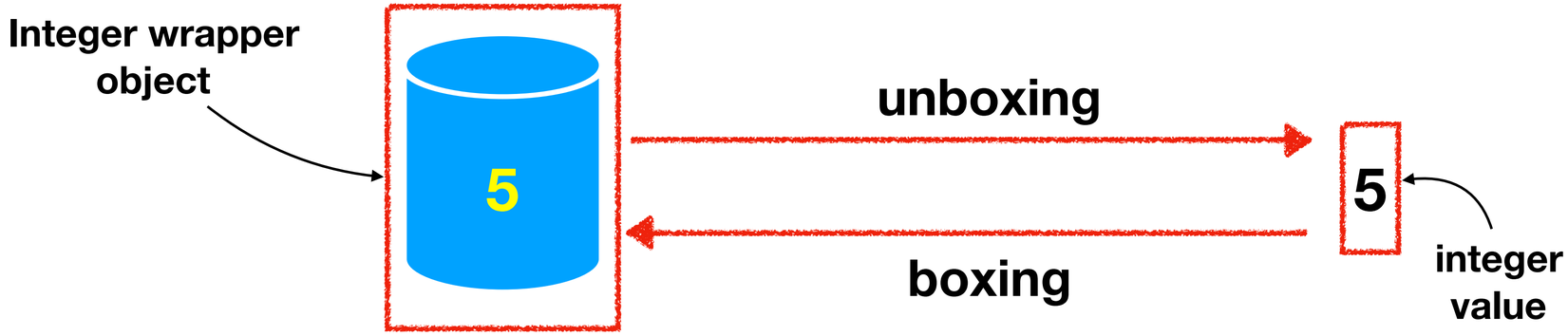
Collection Interfaces

"…defines operations which can be performed…"



Collection Hierarchy

Autoboxing



- primitive types have a related wrapper class
- conversion between primitive value and wrapper object is automatic
- wrapper classes contain utility methods

```
class MyClass {
 public static void main() {

  Integer i[] = new Integer[5];
  i[1] = 5;
  int intValue = i[1];

  int convertedInt = Integer.parseInt("5");
 }
}
```

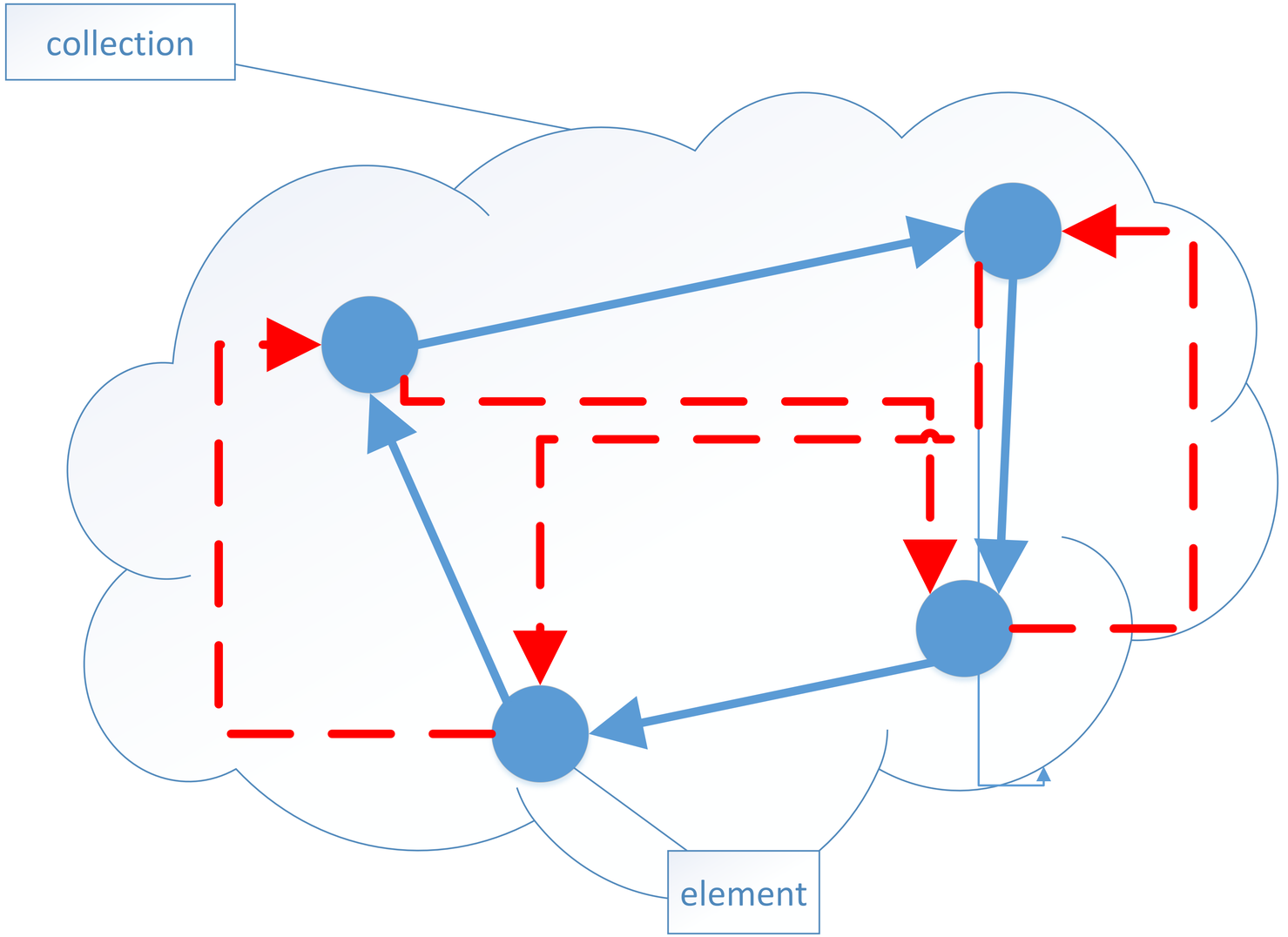
autoboxing

unboxing

Collection Interface

- cannot store primitive types, only object references
- has operations performed on whole collection
- provides access to Iterator object
- Collections class has a set of static methods

```
public class CollectionSample {

  public static void main(String[] args) {

    Collection<Integer> c = new ArrayList<>();
    System.out.println("empty?" + c.isEmpty());
    c.add(1); c.add(2); c.add(3);

    System.out.println("5 exists?" + c.contains(5));
    System.out.println("empty?" + c.isEmpty());
    System.out.println("size:" + c.size());


    Iterator i = c.iterator();
    while(i.hasNext())
      System.out.println("element:" + i.next());

    System.out.println("max:" + Collections.max(c));
   }
}
```



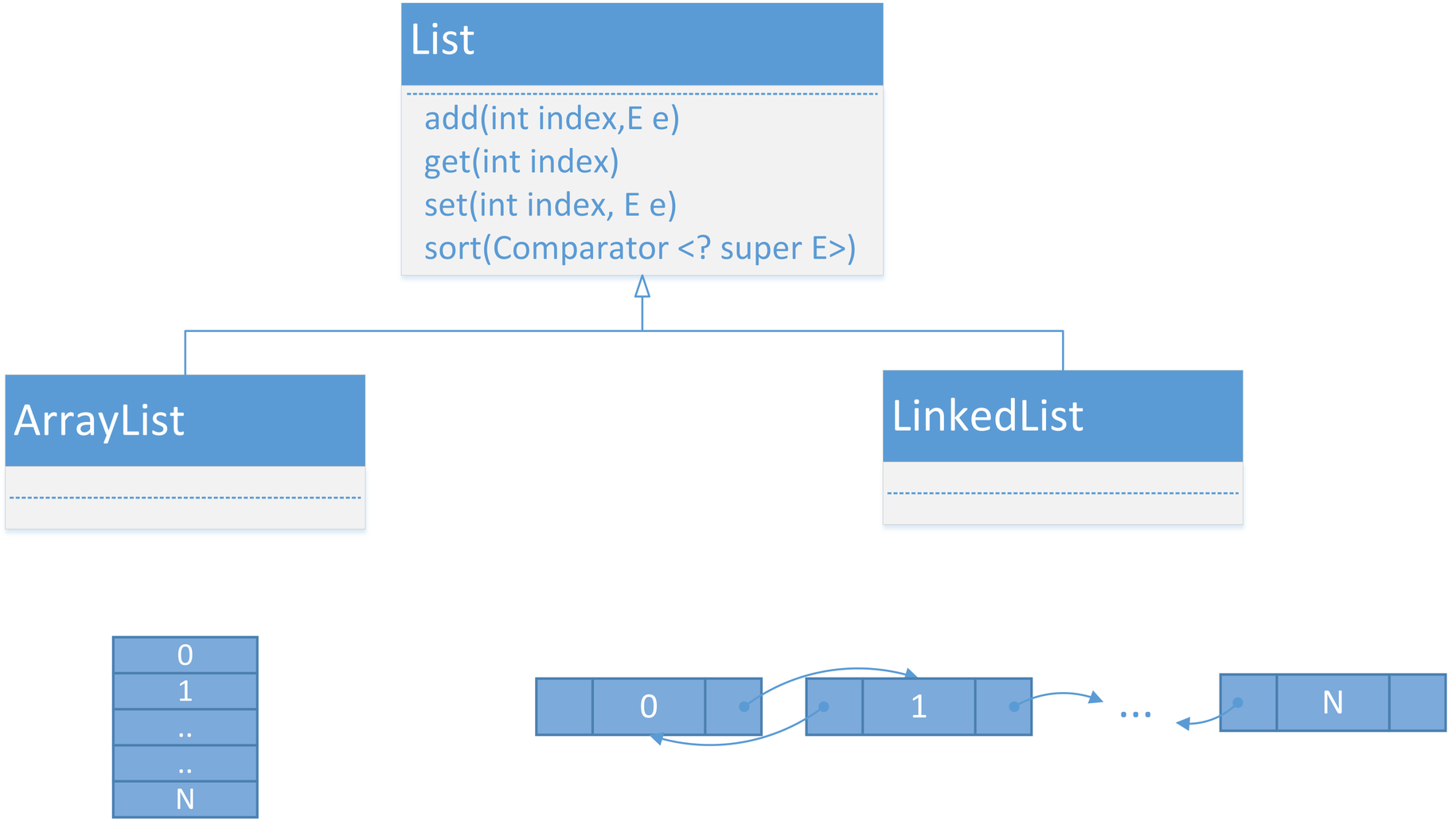
Iterators

"…object which allows traversal of elements…"

List Interface

- sequential collection
- elements accessed by index
- similar to array functionality except resizable

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

    List<Integer> intList = new ArrayList<>();
    intList.add(1); intList.add(0,2);intList.add(3);
    List<String> subList =
          intList.subList(1, intList.size());

    // print all elements
    Iterator<String> it = intList.iterator();
    while(it.hasNext())System.out.println(it.next());

    // alternate implementation
    intList.forEach(System.out::println);
} }
```



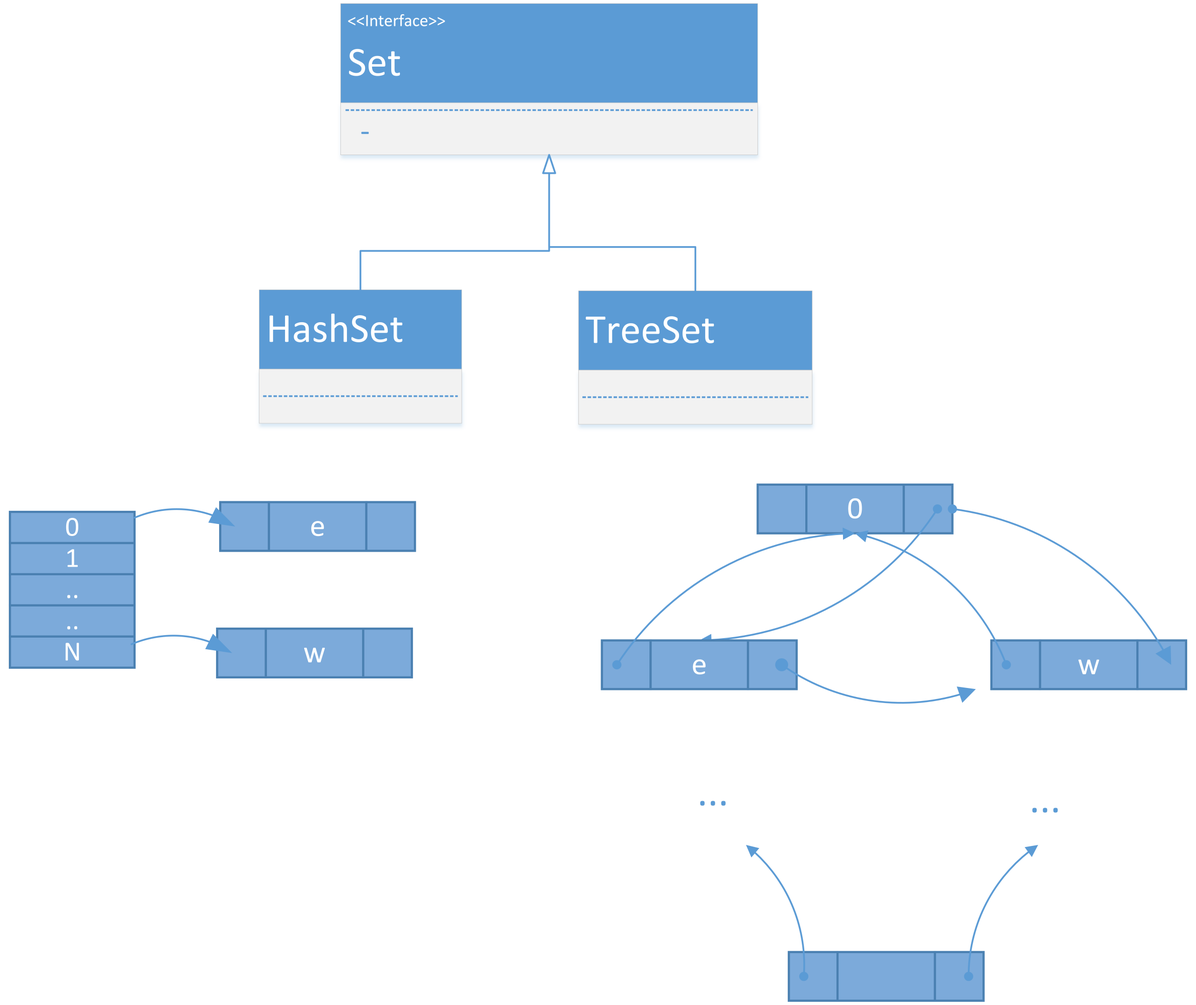
List Implementations

"…optimized per use case…"

Set Interface

- unordered structure with no duplicates...
- main operation is membership test
- useful for other set-algebric operations (union, intersection, ...)

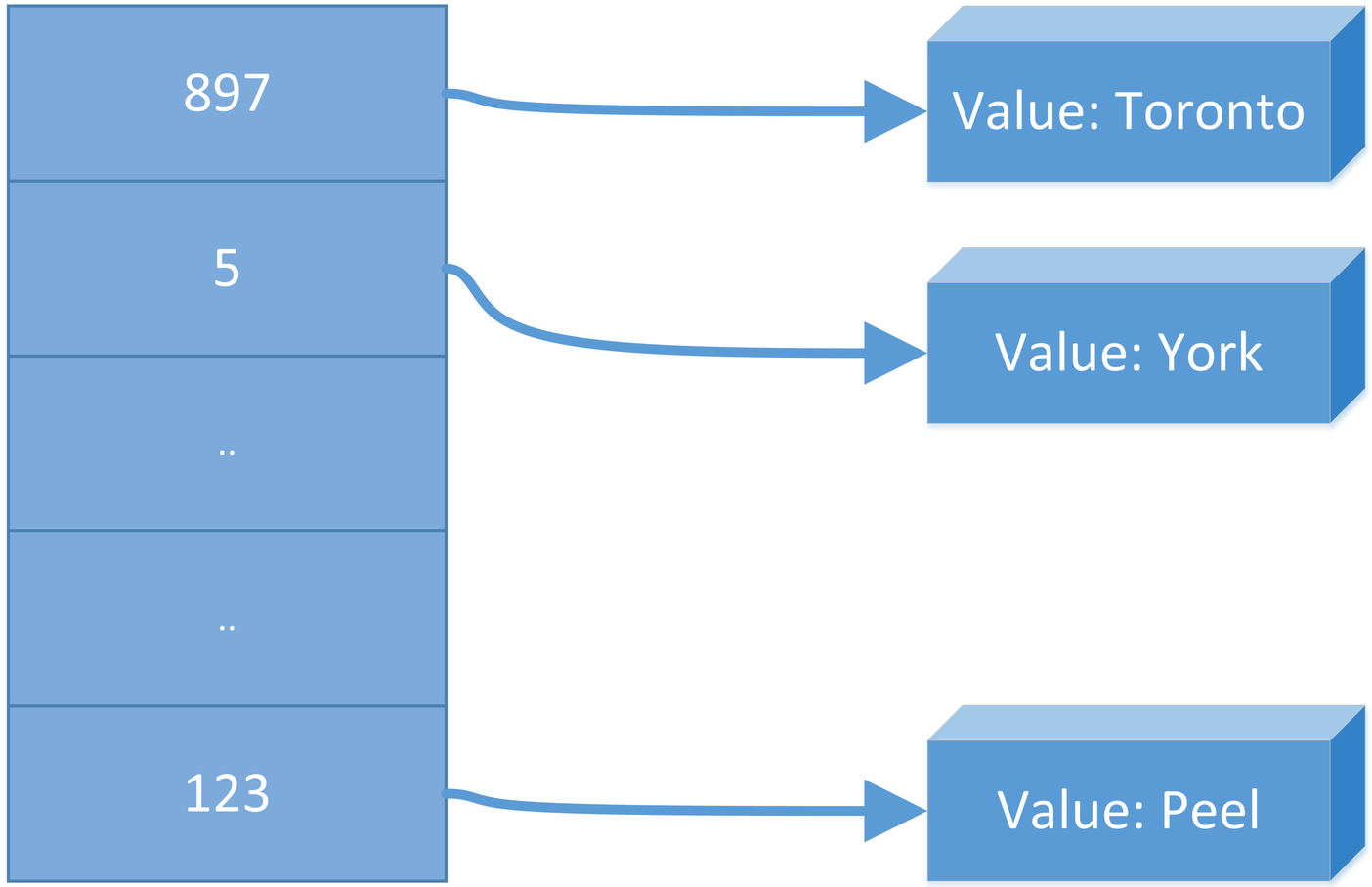
```
public class SetExample {
  public static void main(String[] args) {

    Set<String> set = new HashSet<>();
    if (!set.add("testElement"))
      throw new MyDuplicateException();

  }
}
```



Set Implementations

Map Interface

- maps keys to values (like a function)
- keys are unique
- each key is associated to one value



Map Interface

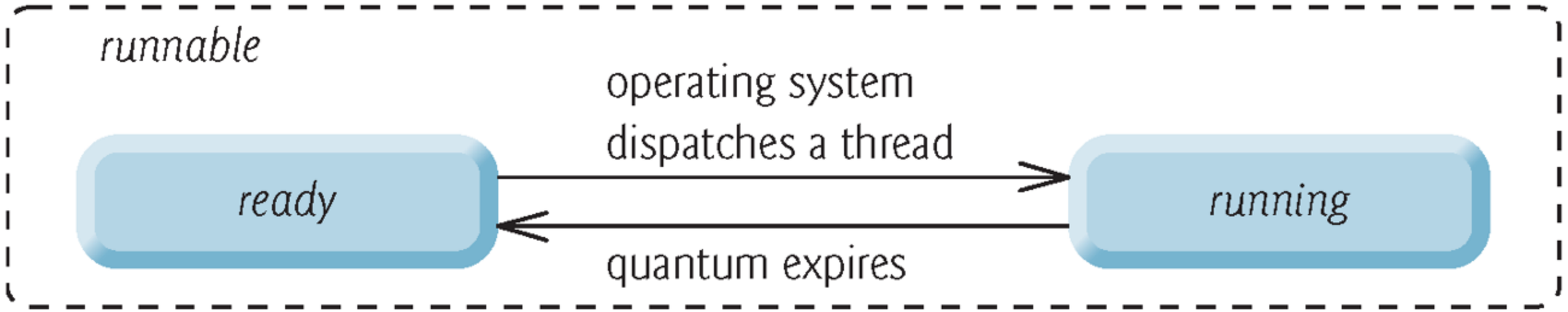
- maps keys to values
- keys are unique
- each key is associated to a value

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

   Map<String, Integer> c = new HashMap<>();
   c.put("Alice",1); c.put("Alice",2);
   int count = c.get("Alice");
   count = c.getOrDefault(“Barney”, 0);

   for (Map.Entry<String,Integer> entry : c.entrySet()) {
     String key = entry.getKey();
     Integer value = entry.getValue();
     process(key, value);
  }

  public static void process(String k, Integer v) {..}
}
```

Concurrency

Concurrency

- when multiple tasks progress over time
- parallelism, when multiple tasks execute simultaneously
- thread, unit of execution with its own stack and program counter
- multithreading, multiple threads can execute while sharing system resources

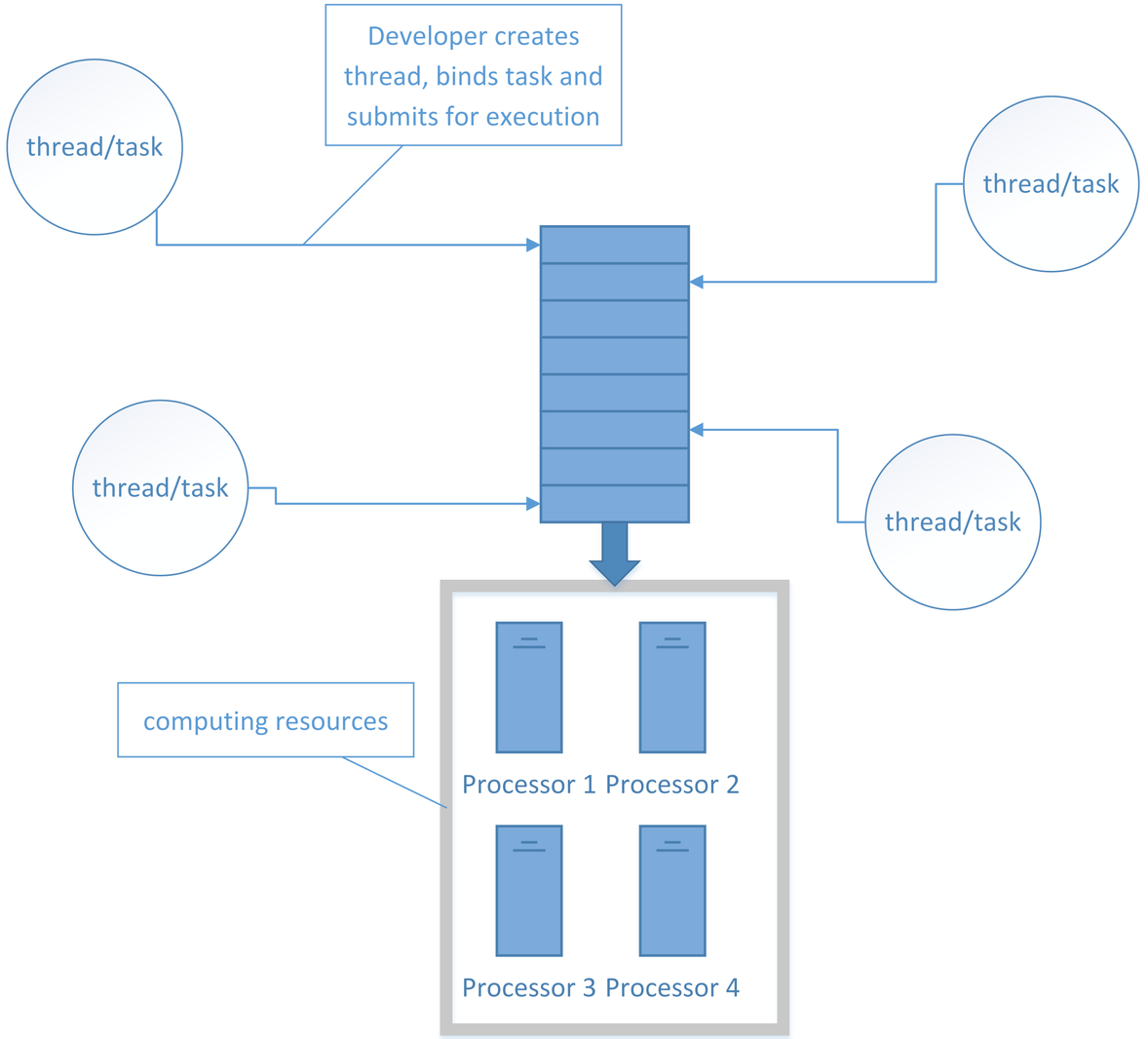


Operating system's internal view of Java's *runnable* state.

(How to Program Java, Deital 2018)

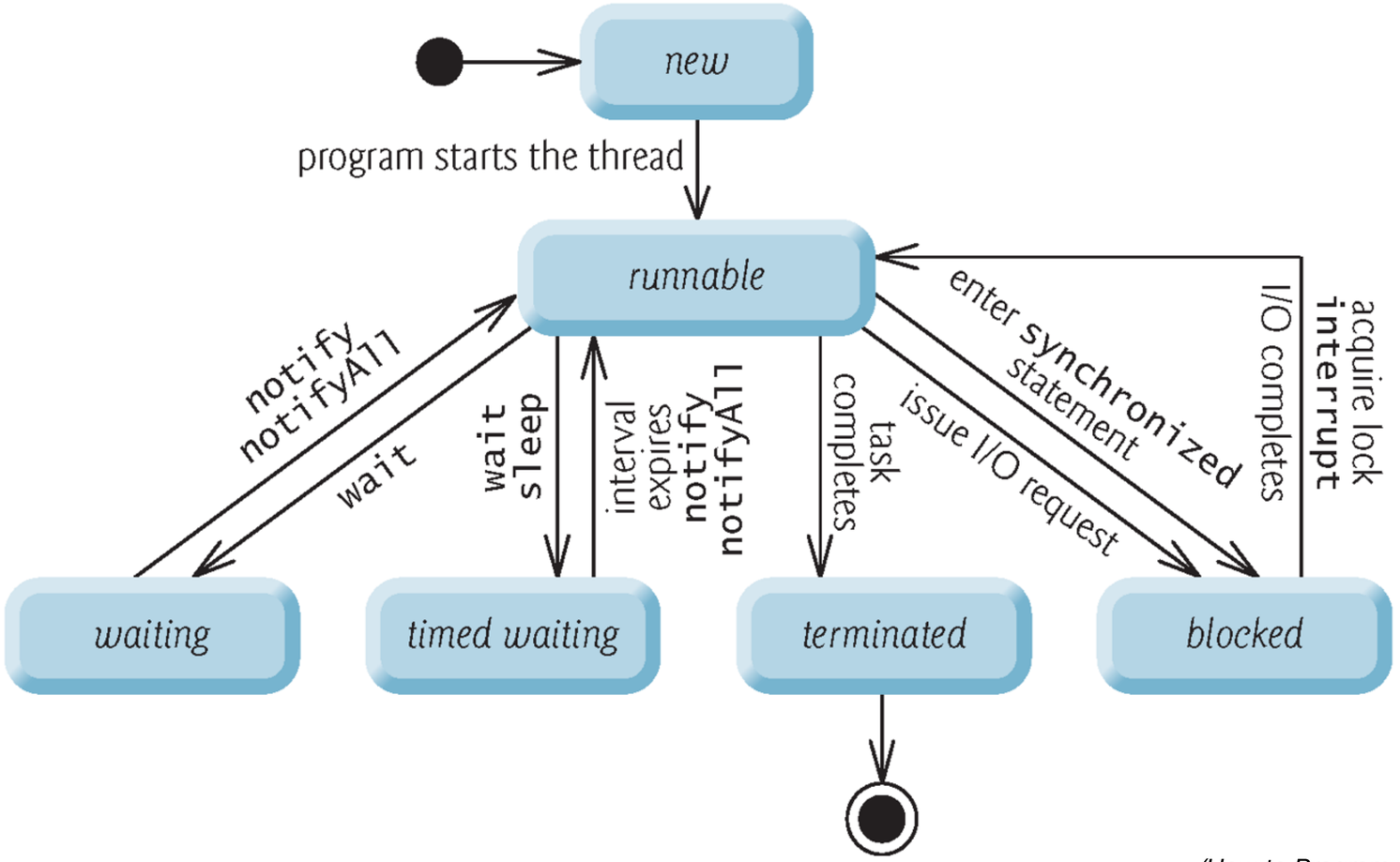
Operating System View

"…programmer has no control over thread execution …"



Thread Object

"…create a thread to be selected for execution…"



(How to Program Java, Deital 2018)

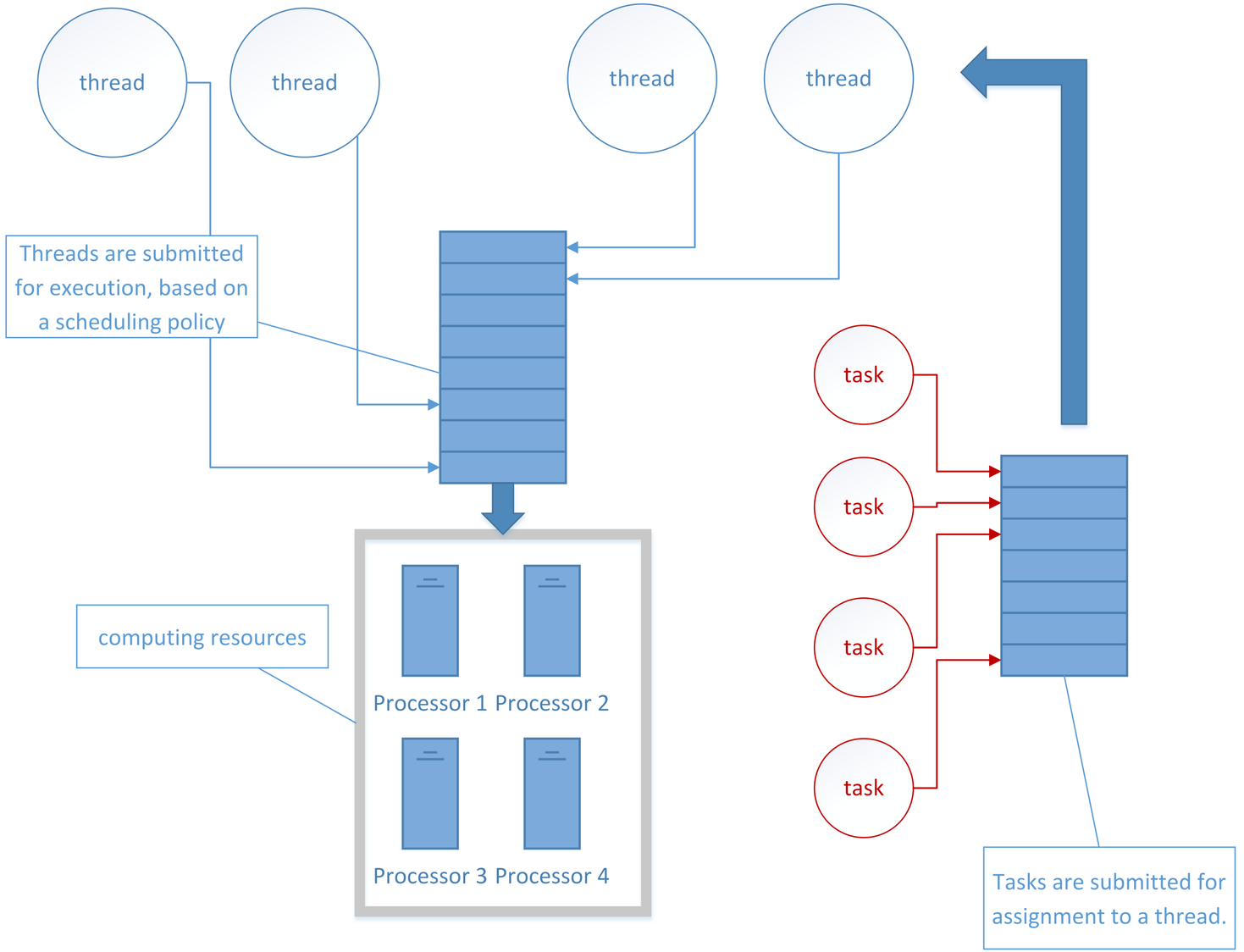
Fig. 23.1 | Thread life-cycle UML state diagram.

Thread Lifecycle

"…possible states during program execution…"

Creating Concurrent Applications

- create tasks
- execute tasks with an Executor

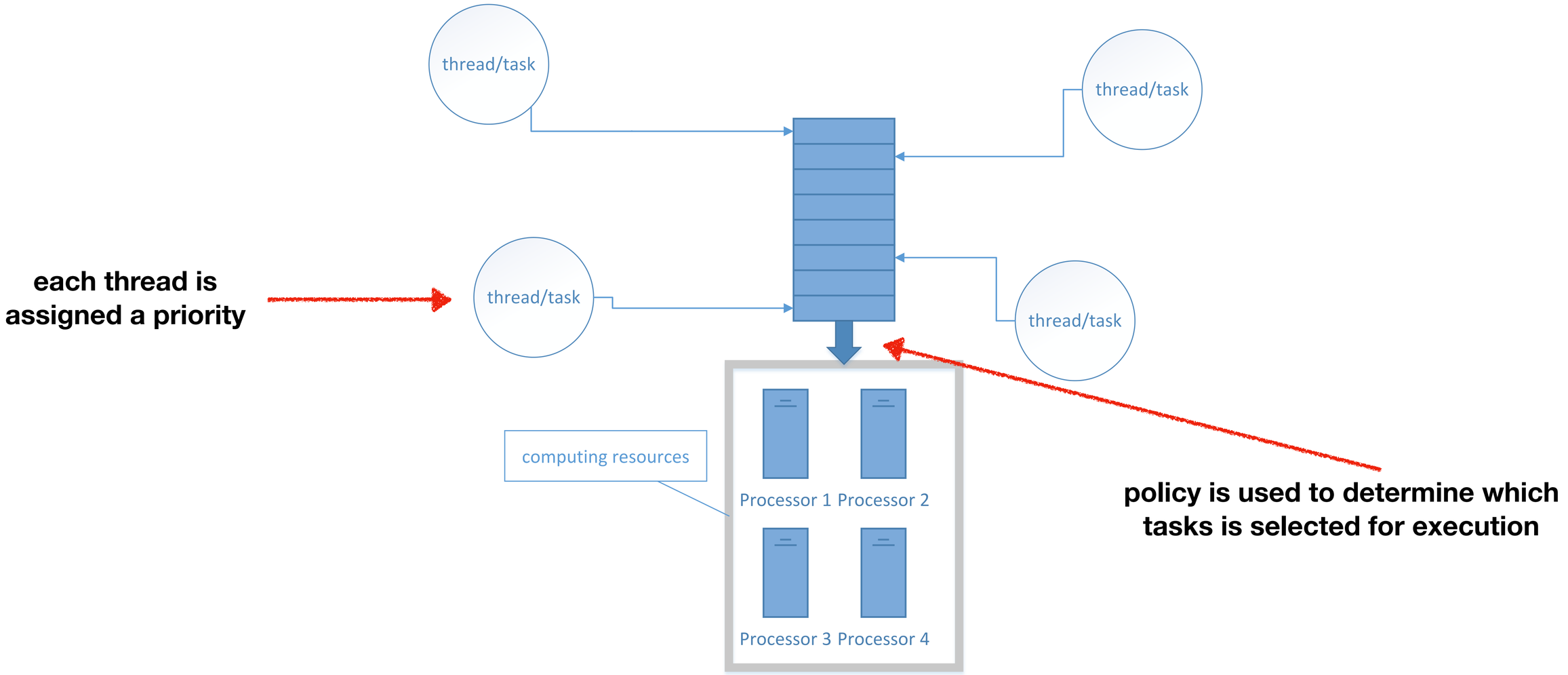


Creating Concurrent Applications

```
class PrintTask implements Runnable {
    @Override
    public void run() { // task work }
}

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

    PrintTask t1 = new PrintTask(“task1”); PrintTask t2 = new PrintTask("task2");
    PrintTask t3 = new PrintTask("task3");

    ExecutorService e = Executors.newCachedThreadPool();

    e.execute(t1); e.execute(t2); e.execute(t3);
    e.shutdown();

    }
}
```

task definition

create group of threads



Starvation

"… when a task fails to progress over time…"

Concurrency Challenges

task1 commands	order	task2 commands
put x into register (tmp1=0)	1	
add 1 (tmp1 = 1)	2	
store result in x (x = tmp1)	3	
	4	put x into register (tmp2=1)
	5	add 1 (tmp2 = 2)
	6	store result in x (x = tmp2)

Single Thread Environment

"…steps always occur in a particular order…"

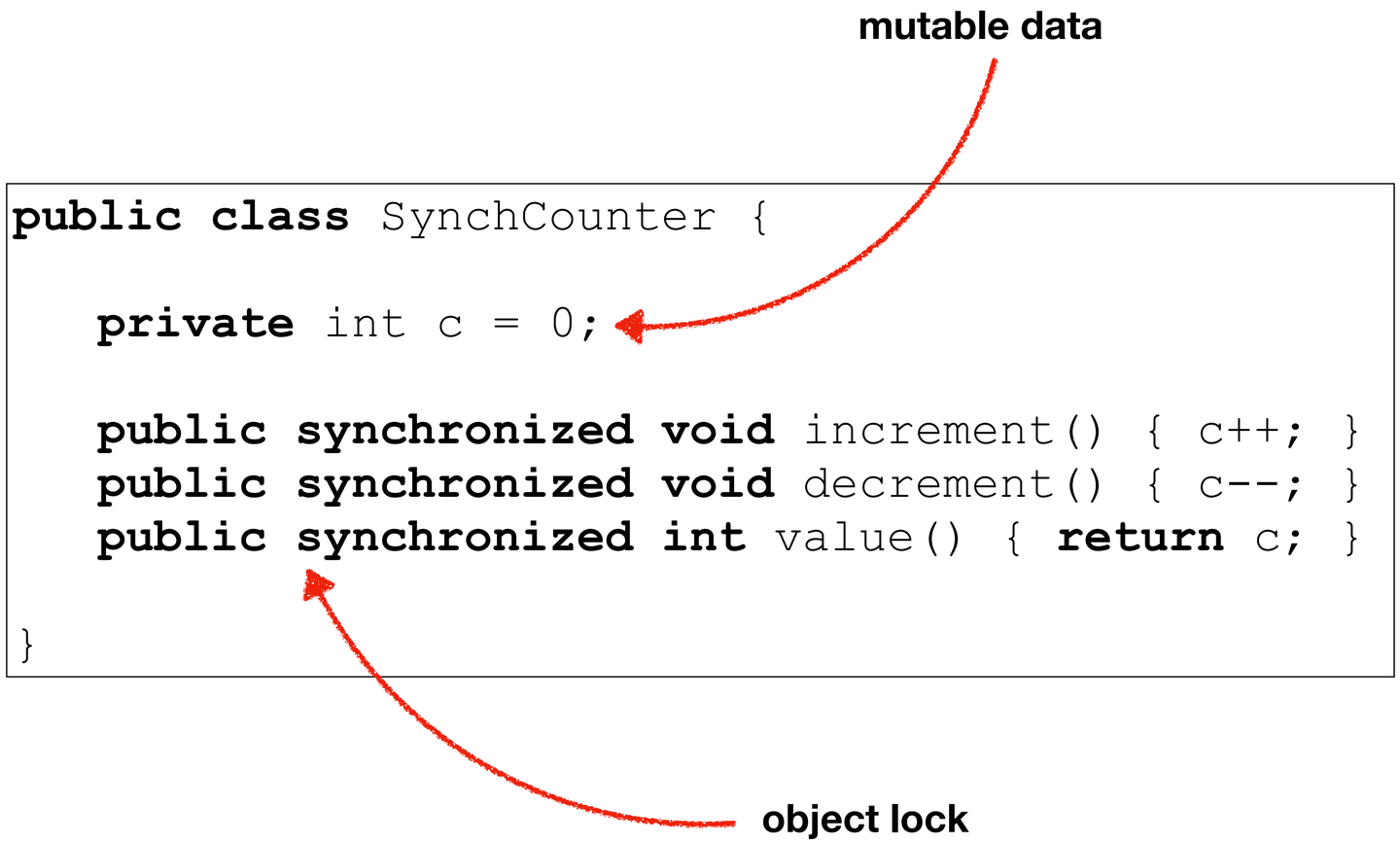
task1 commands	order	task2 commands
put x into register (tmp1=0)	1	
	2	**put x into register (tmp2=0)**
add 1 (tmp1 = 1)	3	
	4	add 1 (tmp2 = 1)
store result in x (x = tmp1)	5	
	6	store result in x (x = tmp2)

Multithreaded Environment

"…steps are interleaved occur in an unpredictable order…"

Race Condition

- when multiple threads access a shared resource
- results may differ each time
- access to resources must be synchronized to prevent corruption



Deadlock

when a group of threads are waiting for each other to release a resource