03-Java : Collections and Exception Handling

Join Google+ community

http://goo.gl/U7qVS

You can ask all your doubts, questions and queries by posting on this G+ community during/after webinar

http://openandroidlearning.org

Exceptions in Java

- An *exception* is an error condition that changes the normal flow of control in a program
- Exceptions in Java separates error handling from main business logic
- Based on ideas developed in Ada, Eiffel and C++
- Java has a uniform approach for handling all synchronous errors
  - From very unusual (e.g. out of memory)
  - To more common ones your program should check itself (e.g. index out of bounds)
  - From Java run-time system errors (e.g., divide by zero)
  - To errors that programmers detect and raise deliberately

Throwing and catching

- An error can *throw* an *exception*

```
throw <exception object>;
```

- By default, exceptions result in the thread terminating after printing an error message
- However, exception handlers can *catch* specified exceptions and recover from error

```
catch (<exception type> e) {
 //statements that handle the exception
}
```

Exceptional flow of control

- Exceptions break the normal flow of control.
- When an exception occurs, the statement that would normally execute next is not executed.
- What happens instead depends on:
  - whether the exception is caught,
  - where it is caught,
  - what statements are executed in the ‘catch block’,
  - and whether you have a ‘finally block’.

Approaches to handling an exception

1. Prevent the exception from happening
2. Catch it in the method in which it occurs, and either
   a. Fix up the problem and resume normal execution
   b. Rethrow it
   c. Throw a different exception
3. Declare that the method throws the exception
4. With 1. and 2.a. the caller never knows there was an error.
5. With 2.b., 2.c., and 3., if the caller does not handle the exception, the program will terminate and display a stack trace

Exception hierarchy

- Java organizes exceptions in inheritance tree:
  - Throwable
  - Error
  - Exception
    - RuntimeException
    - TooManyListenersException
    - IOException
    - AWTException

Java is strict

- Unlike C++, is quite strict about catching exceptions
- If it is a checked exception
  - (all except Error, RuntimeException and their subclasses),
  - Java compiler forces the caller must either catch it
  - or explicitly re-throw it with an exception specification.
- Why is this a good idea?
- By enforcing exception specifications from top to bottom, Java guarantees exception correctness at compile time.
- Here's a method that ducks out of catching an exception by explicitly re-throwing it:

```
void f() throws tooBig, tooSmall, divZero {
```

  - The caller of this method now must either catch these exceptions or rethrow them in its specification.

Error and RuntimeException

- Error
  - “unchecked”, thus need not be in ‘throws’ clause
  - Serious system problems (e.g. ThreadDeath, OutOfMemoryError)
  - It’s very unlikely that the program will be able to recover, so generally you should NOT catch these.
- RuntimeException
  - “unchecked”, thus need not be in ‘throws’ clause
  - Also can occur almost anywhere, e.g. ArithmeticException, NullPointerException, IndexOutOfBoundsException
  - Try to prevent them from happening in the first place!
- System will print stop program and print a trace

Catching an exception

```
try {  // statement that could throw an exception
    }
catch (<exception type> e) {
  // statements that handle the exception
}
catch (<exception type> e) { //e higher in hierarchy
  // statements that handle the exception
}
finally {
  // release resources
}
//other statements
```

- At most one catch block executes
- `finally` block always executes once, whether there's an error or not

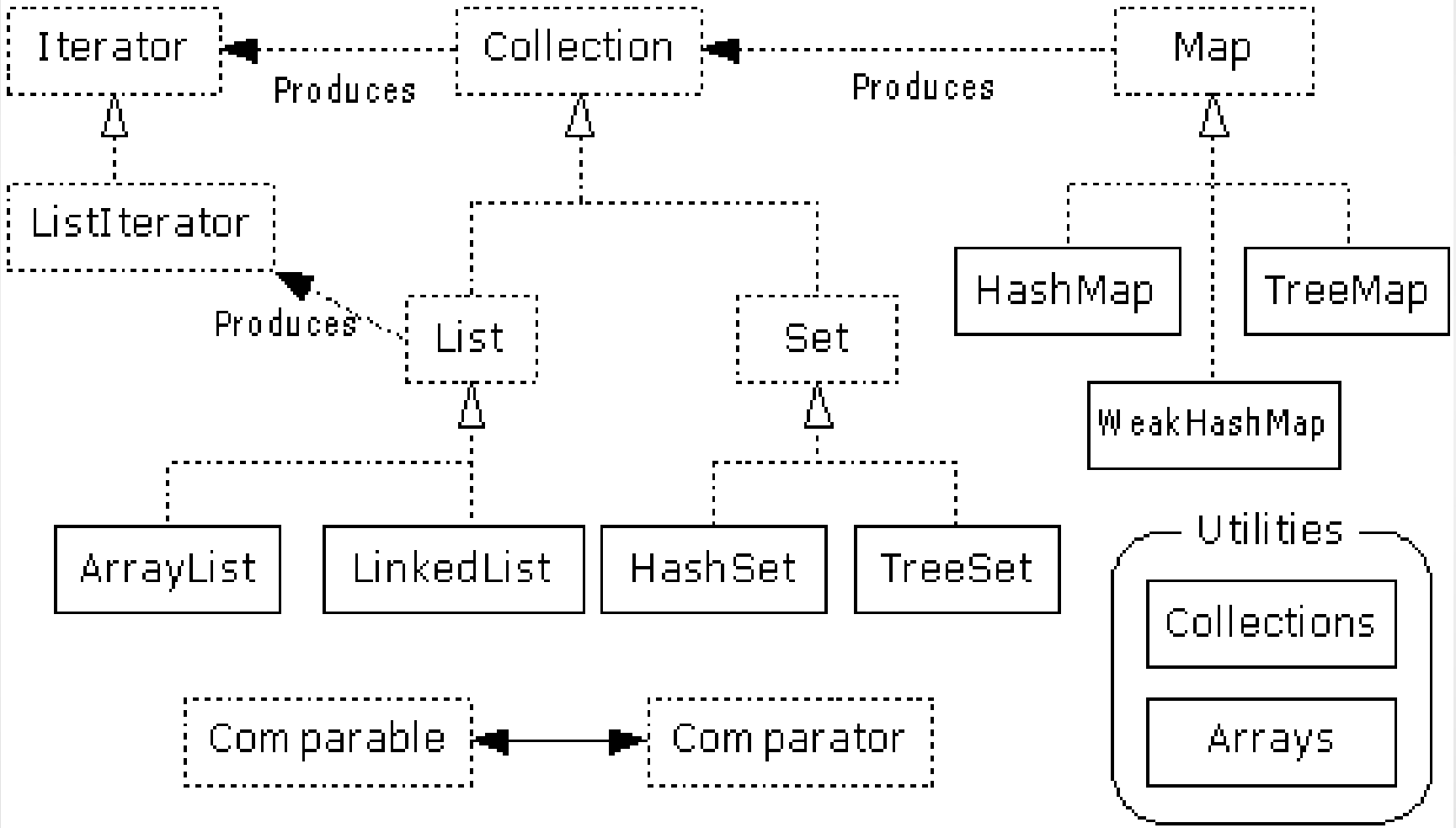
Execution of try catch blocks

- For normal execution:
  - try block executes, then finally block executes, then other statements execute
- When an error is caught and the catch block throws an exception or returns:
  - try block is interrupted
  - catch block executes (until throw or return statement)
  - finally block executes
- When error is caught and catch block doesn't throw an exception or return:
  - try block is interrupted
  - catch block executes
  - finally block executes
  - other statements execute
- When an error occurs that is not caught:
  - try block is interrupted
  - finally block executes

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
- A collections framework contains three things
  - Interfaces
  - Implementations
  - Algorithms

Collections



http://openandroidlearning.org

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

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

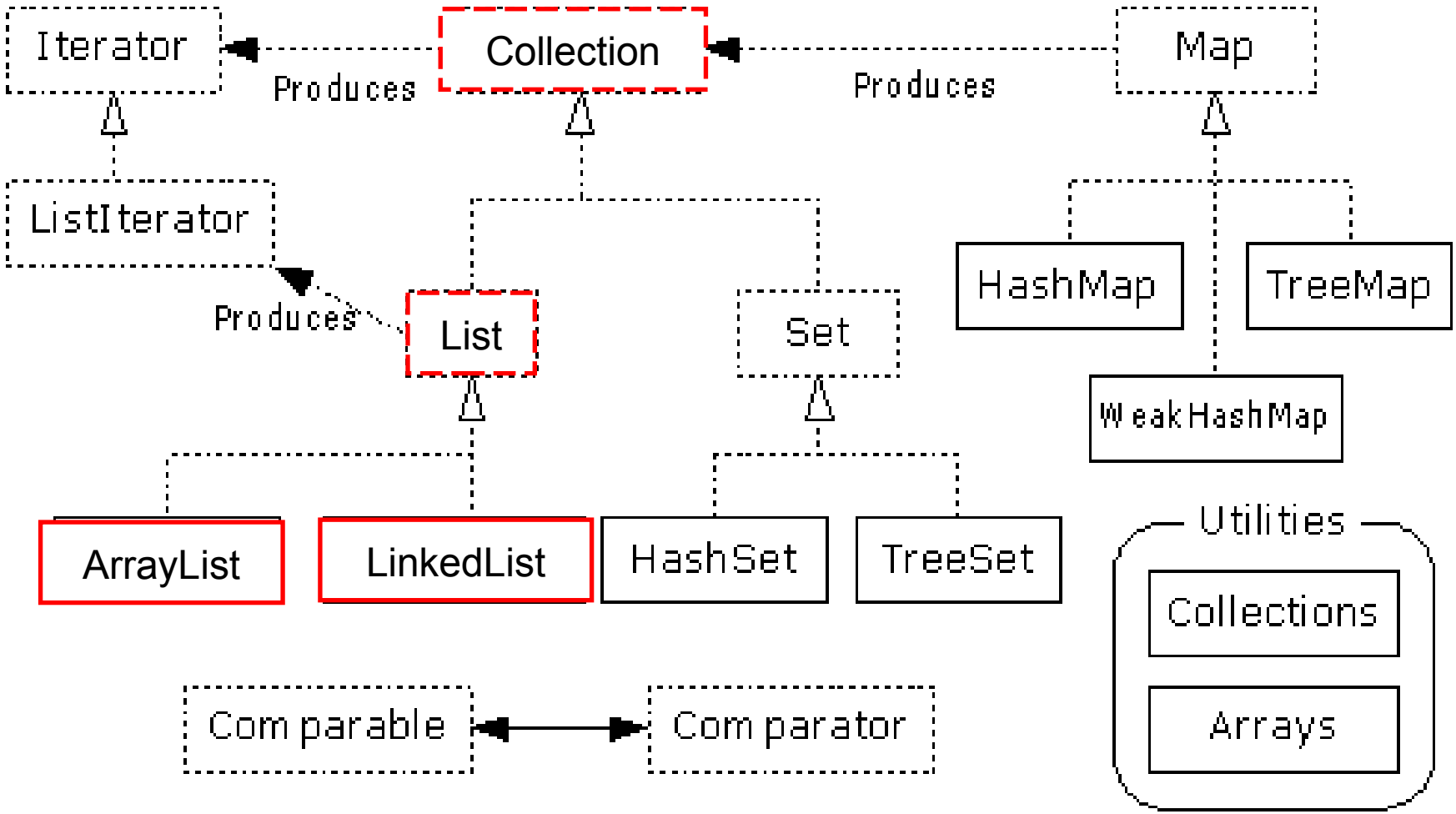
Iterator Interface

- 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

Simple Collection Example

```
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

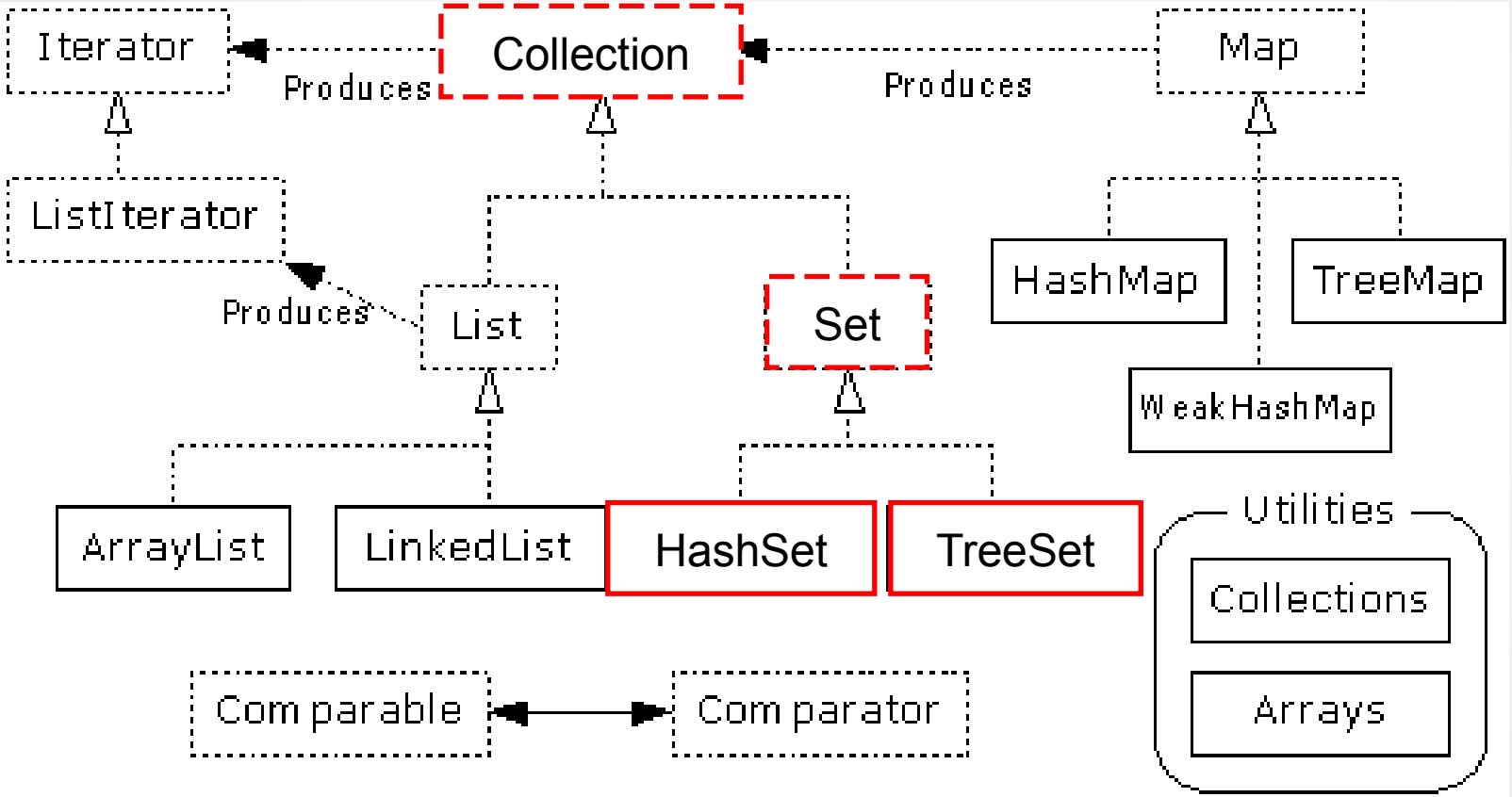


http://openandroidlearning.org

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

Set Interface



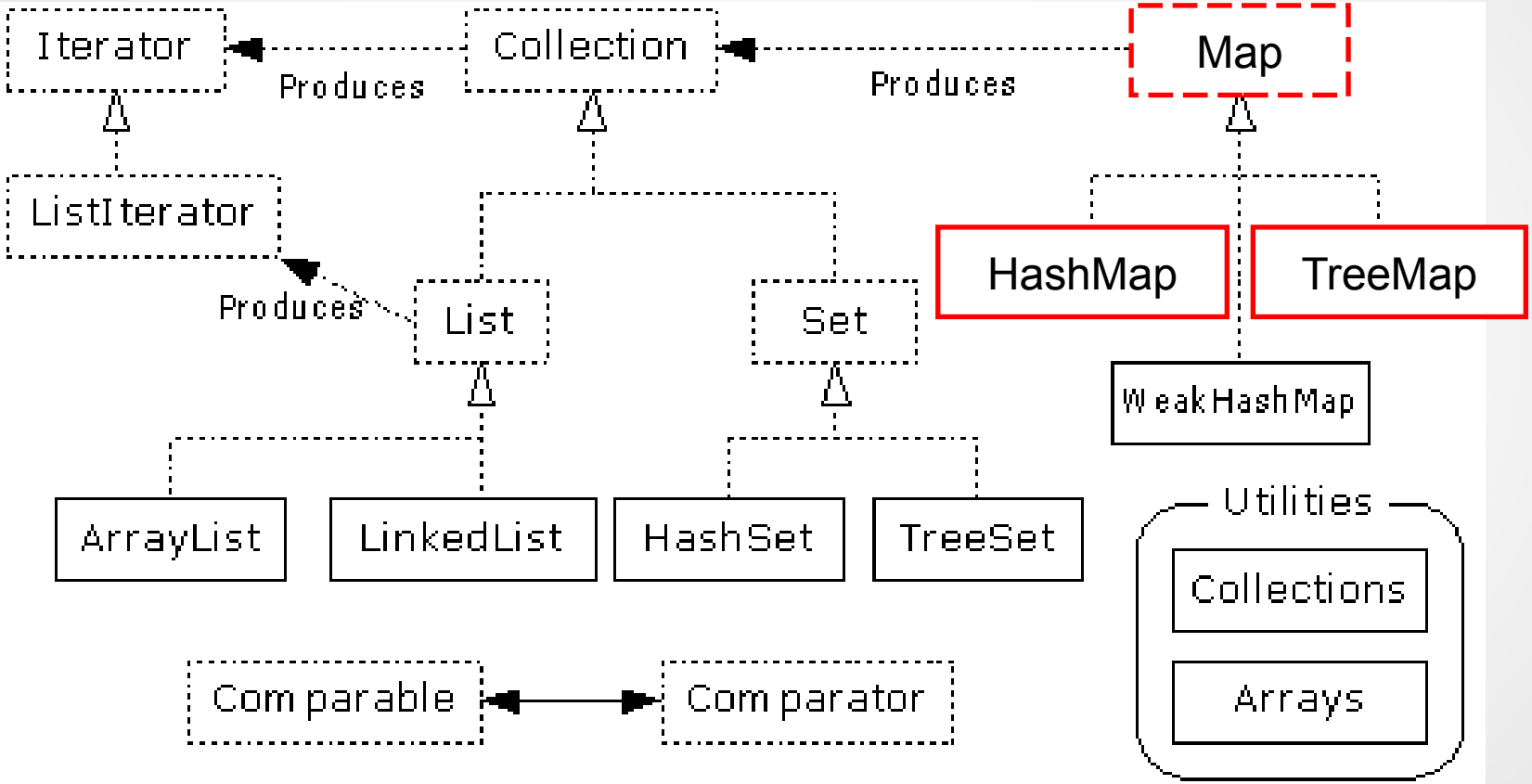
HashSet

- 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

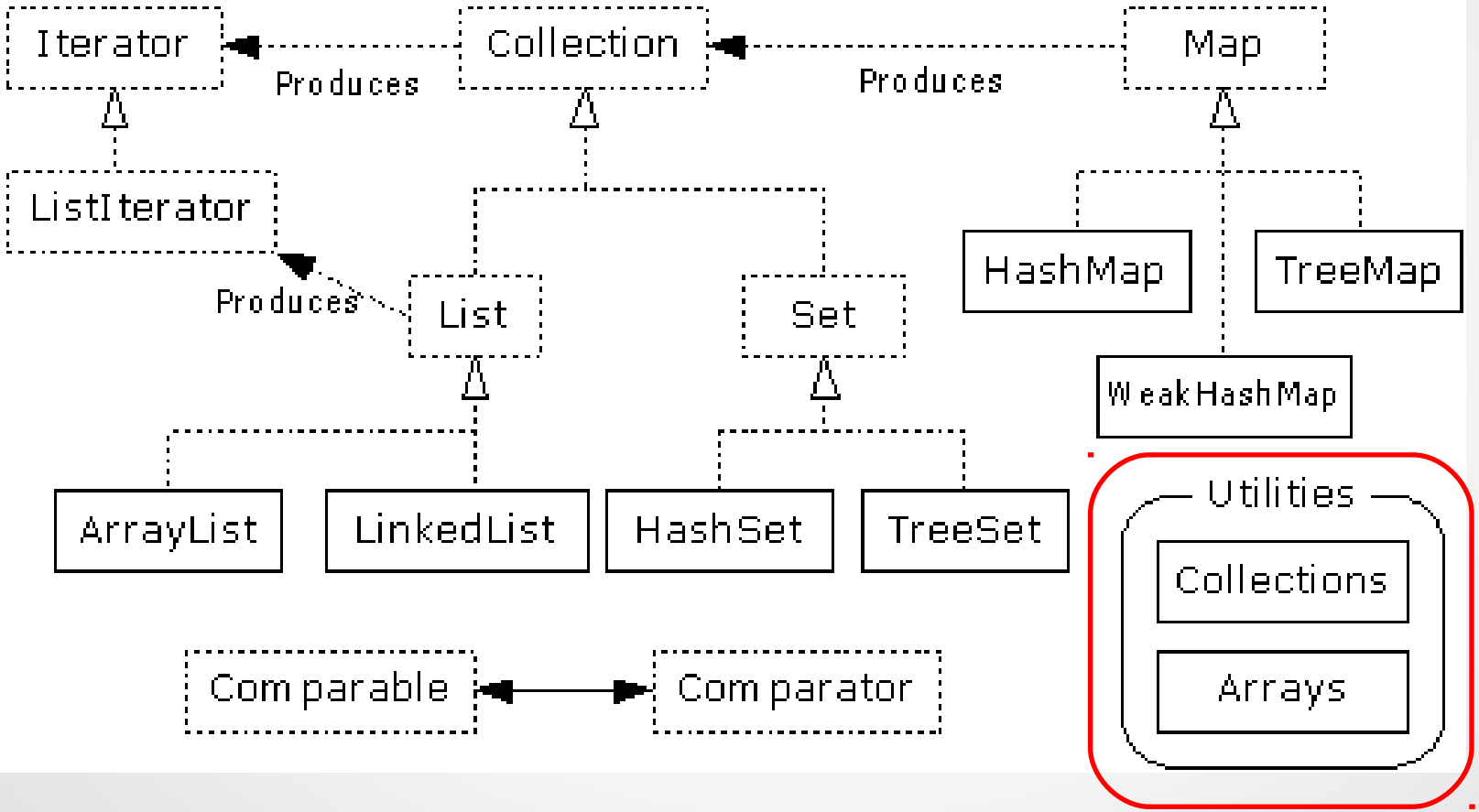


HashMap and TreeMap

- 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))*

Utilities


Iterator
Produces
Collection
Produces
Map
ListIterator
Produces
List
Set
HashMap
TreeMap
WeakHashMap
ArrayList
LinkedList
HashSet
TreeSet
Utilities
Collections
Arrays
Comparable
Comparator

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