# Java Collections

Engi- 5895

Hafez Seliem

# Wrapper classes

- Provide a mechanism to “wrap” primitive values in an object so that the primitives can be included in activities reserved for objects, like as being added to Collections, or returned from a method with an object return value.
- To provide an assortment of utility functions for primitives.

# Wrapper classes

| Primitive | Wrapper Class | Constructor Arguments |
|---|---|---|
| boolean | Boolean | boolean or String |
| byte | Byte | byte or String |
| char | Character | char |
| double | Double | double or String |
| float | Float | float or String |
| int | Integer | int or String |
| long | Long | long or String |
| short | Short | short or String |

# The Wrapper Constructors

All of the wrapper classes except Character provide two constructors: one that takes a primitive of the type being constructed, and one that takes a String representation of the type being constructed—for example,

```
Integer i1 = new Integer(42);
Integer i2 = new Integer("42");
```

or

```
Float f1 = new Float(3.14f);
Float f2 = new Float("3.14f");
```

# The Wrapper Constructors

- The Character class provides only one constructor, which takes a *char* as an argument—for example,

Character c1 = new Character('c');

- ***The constructors for the Boolean wrapper take either a*** *boolean* ***value*** true ***or*** false***, or a case-****insensitive* ***String with the value*** *“true”* ***or*** *“false”****.***
- Boolean b = new Boolean("false");

# The valueOf() Methods

Take a String representation of the appropriate type of primitive as their first argument.

The second method (when provided) takes an additional argument, int radix, which indicates in what base (for example binary, octal, or hexadecimal) the first argument is represented—for example,

```
Integer i2 = Integer.valueOf("101011", 2);
// converts 101011 to 43 and assigns the value 43 to the Integer object i2

Float f2 = Float.valueOf("3.14f");
// assigns 3.14 to the Float object f2
```

# xxxValue()

```
Integer i2 = new Integer(42); // make a new wrapper object
byte b = i2.byteValue(); // convert i2's value to a byte primitive
short s = i2.shortValue(); // another Integer's xxxValue method
double d = i2.doubleValue(); // another Integer's xxxValue method

Float f2 = new Float(3.14f); // make a new wrapper object
short s = f2.shortValue(); // convert f2's value to a short primitive
System.out.println(s); // result is 3 (truncated, not rounded)
```

# xxxValue()

- When you need to convert the value of a wrapped numeric to a primitive, use one of the many xxxValue() methods.

| Method<br>s = static<br>n = NFE exception | Boolean | Byte | Character | Double | Float | Integer | Long | Short |
|---|---|---|---|---|---|---|---|---|
| byteValue | | x | | x | x | x | x | x |
| doubleValue | | x | | x | x | x | x | x |
| floatValue | | x | | x | x | x | x | x |
| intValue | | x | | x | x | x | x | x |
| longValue | | x | | x | x | x | x | x |
| shortValue | | x | | x | x | x | x | x |

# parseXxx() and valueOf()

```
double d4 = Double.parseDouble("3.14"); // convert a String to a primitive
System.out.println("d4 = " + d4); // result is "d4 = 3.14"
Double d5 = Double.valueOf("3.14"); // create a Double object
System.out.println(d5 instanceof Double ); // result is "true"

The next examples involve using the radix argument, (in this case binary):
long L2 = Long.parseLong("101010", 2); // binary String to a primitive
System.out.println("L2 = " + L2); // result is "L2 = 42"
Long L3 = Long.valueOf("101010", 2); // binary String to Long object
System.out.println("L3 value = " + L3); // result is "L2 value = 42"
```

# parseXxx() and valueOf()

- parseXxx() returns the named primitive
- valueOf() returns a newly created wrapped object of the type that invoked the method

But both

- Take a String as an argument
- Throw a NumberFormatException if the String argument is not properly formed
- Can convert String objects from different bases (radix), when the underlying primitive type is any of the four integer types

# toString()

```
Double d = new Double("3.14");
System.out.println("d = " + d.toString() ); // result is "d = 3.14“
```

All of the numeric wrapper classes provide an overloaded, static toString() method that takes a primitive numeric of the appropriate type:

```
System.out.println("d = " + Double.toString(3.14)); // result is "d = 3.14“
```

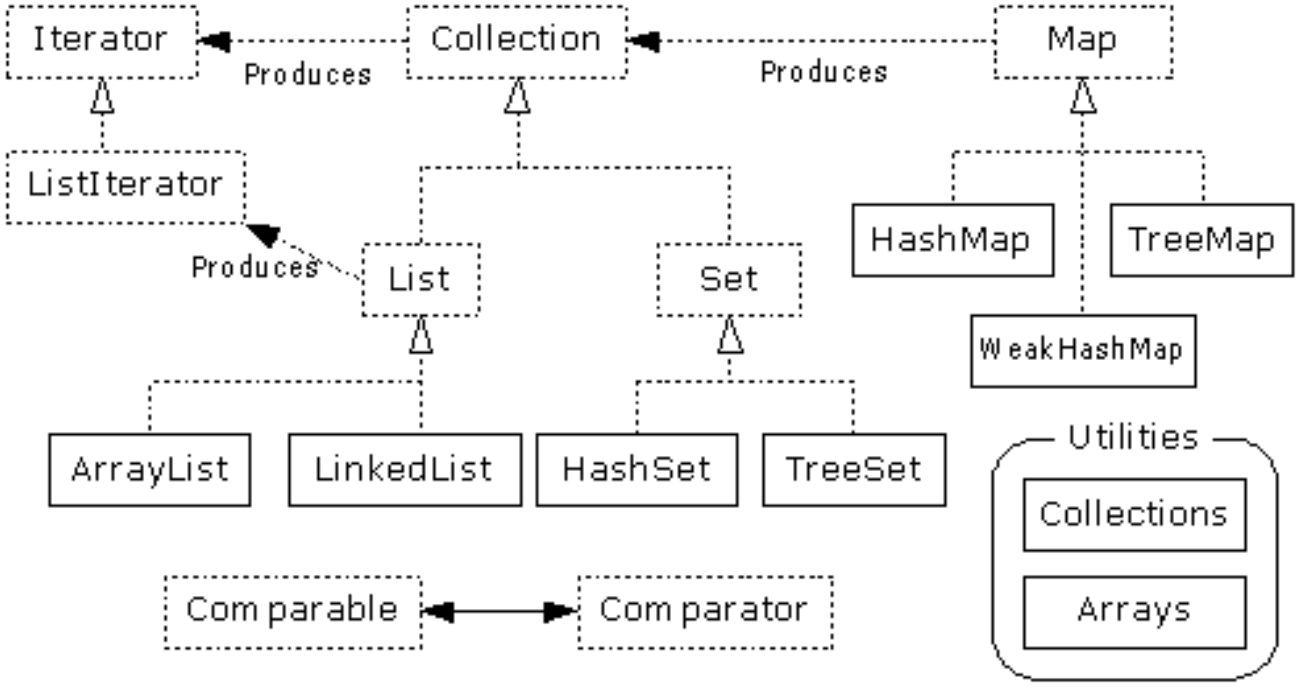
Finally, Integer and Long provide a third toString() method. It is static, its first argument is the appropriate primitive, and its second argument is a *radix*. The radix argument tells the method to take the first argument (which is radix 10 or base 10 by default), and convert it to the radix provided, then return the result as a String—for instance,

```
System.out.println("hex = " + Long.toString(254,16); // result is "hex = fe"
```

# Java Collections

- A collection is an object that groups multiple elements into a single unit
- A collections framework contains three things
  - Interfaces
  - Implementations
  - Algorithms
- Very useful
  - store, retrieve and manipulate data
  - transmit data from one method to another
  - data

# Collections Framework Diagram



# SimpleCollection

```
public class SimpleCollection  {
  public static void main(String[] args) {
    Collection c;
    c = new ArrayList();
    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());
  }
}
```

# Generics

**Generics are a way to define which types are allowed in your class or function**

```
// old way
List myIntList1 = new LinkedList();
myIntList1.add(new Integer(0));
Integer x1 = (Integer) myIntList1.iterator().next();

// with generics
List<Integer> myIntList2 = new LinkedList<Integer>();
myIntList2.add(new Integer(0));'
Integer x2 = myIntList2.iterator().next();
```

Can put here just 0, using autoboxing

# 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

# 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

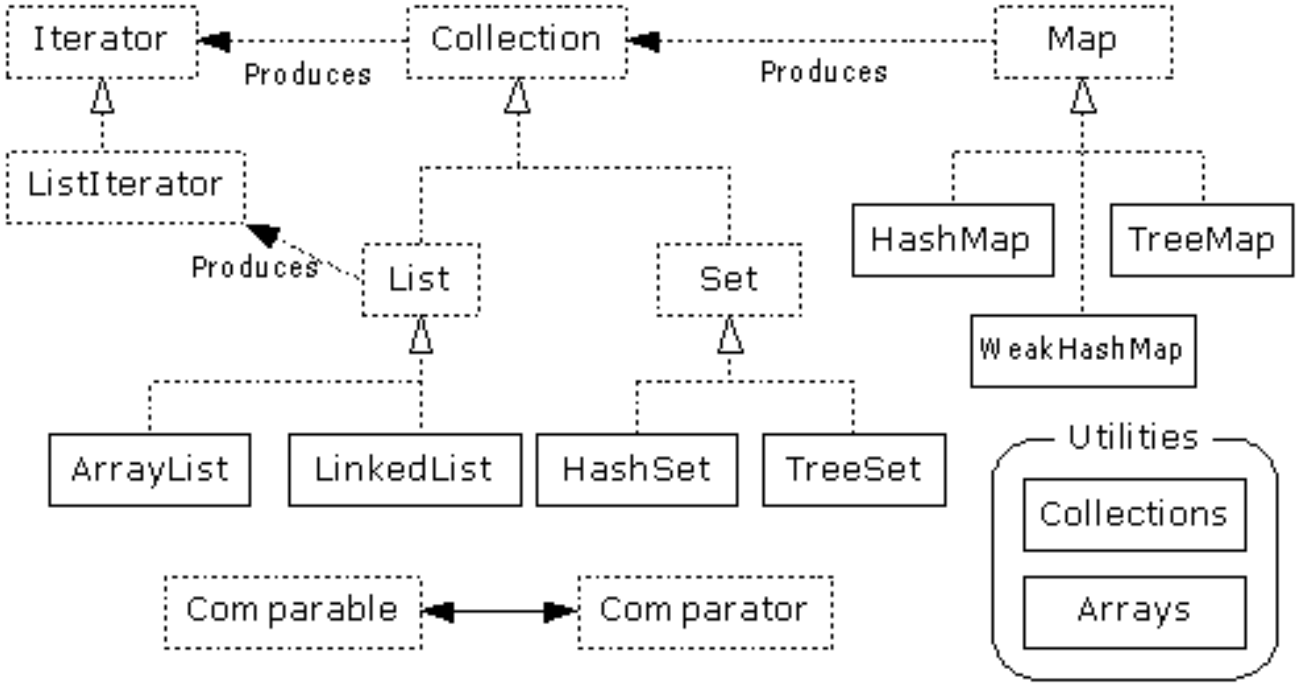
# ArrayList methods

- 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 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()`**

# Collections Framework Diagram



# 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

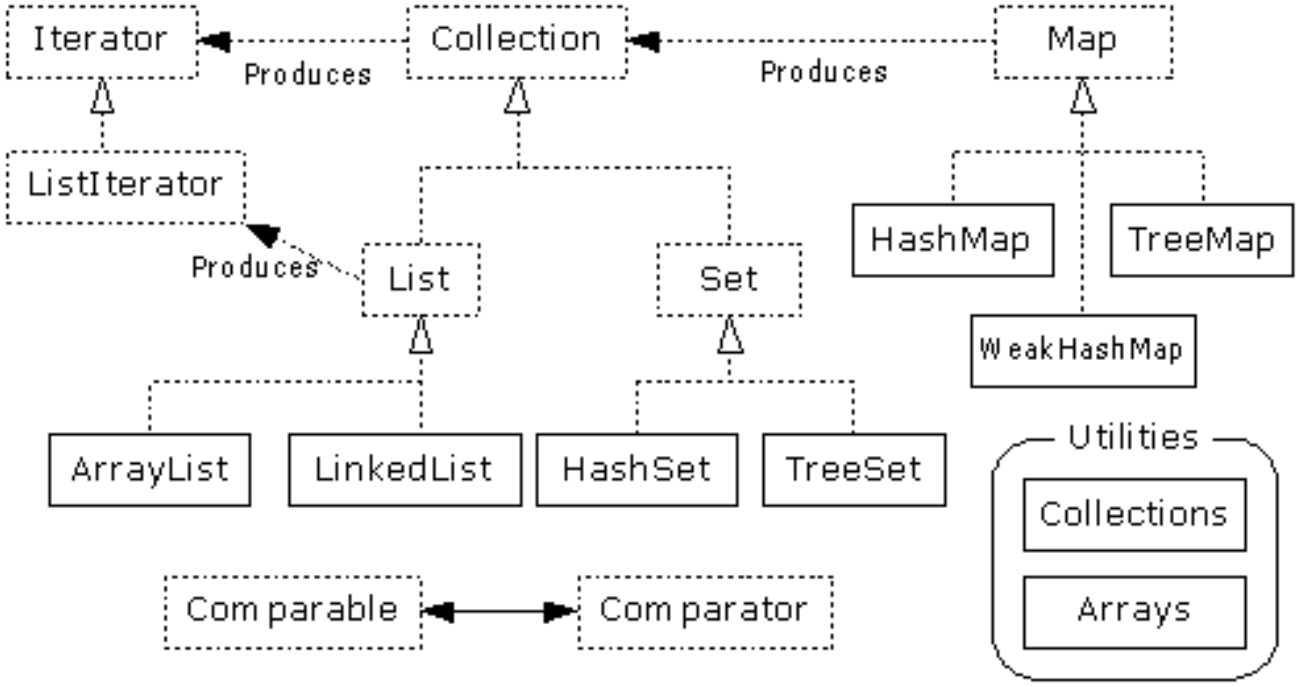
# TreeSet

- 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

# HashSet

```
Set set = new HashSet();
set.add(obj);
int n = set.size();
if (set.contains(obj)) {...}
// iterate through the set
Iterator iter = set.iterator();
while (iter.hasNext()) {
  Object e = iter.next();
}
```

# Collections Framework Diagram



# 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))*

# HashMap

```
HashMap<String, Integer> frequency(String[] names) {
   HashMap<String, Integer> frequency =
         new HashMap<String, Integer>();
   for(String name : names) {
      Integer currentCount = frequency.get(name);
      if(currentCount == null) {
         currentCount = 0; // auto-boxing
      }
      frequency.put(name, ++currentCount);
   }
   return frequency;
}
```

# Throwing Exceptions

- You can throw exceptions from your own methods
- To throw an exception, create an instance of the exception class and "throw" it.
- If you throw checked exceptions, you must indicate which exceptions your method throws by using the

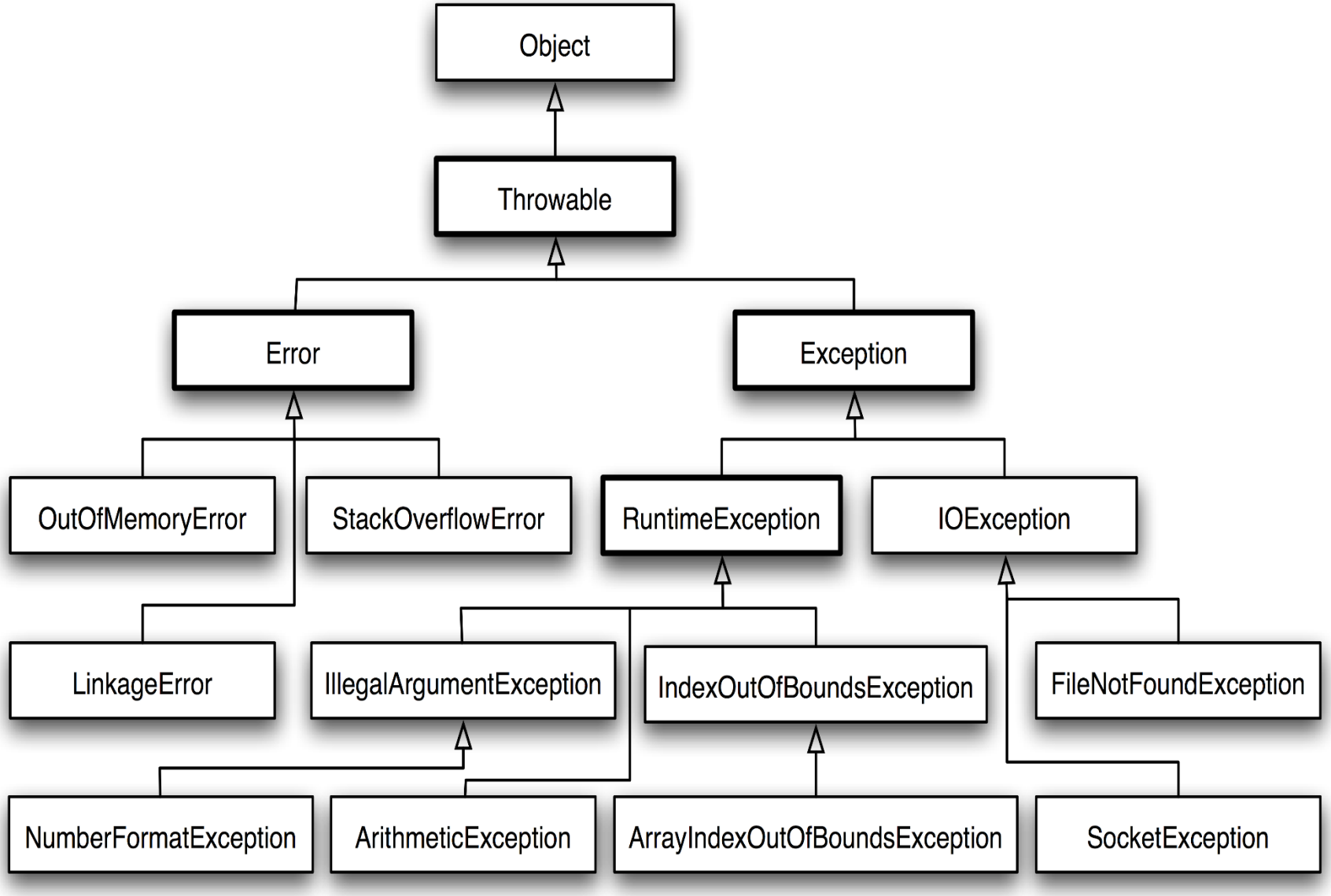
```
public void withdraw(float anAmount) throws Exception
{
    if (anAmount<0.0)
            throw new Exception("Cannot withdraw negative amt");

    if (anAmount>balance)
            throw new Exception("Not enough cash");

    balance = balance - anAmount;
}
```

# Handling Exceptions

```
try
{
	// Code which might throw an exception
	// ...
}
catch(FileNotFoundException x)
{
	// code to handle a FileNotFound exception
}
catch(IOException x)
{
	// code to handle any other I/O exceptions
}
catch(Exception x)
{
	// Code to catch any other type of exception
}
finally
{
	// This code is ALWAYS executed whether an exception was thrown
	// or not.  A good place to put clean-up code. ie. close
	// any open files, etc...
}
```


Object
Throwable
Error
Exception
OutOfMemoryError
StackOverflowError
RuntimeException
IOException
LinkageError
IllegalArgumentException
IndexOutOfBoundsException
FileNotFoundException
NumberFormatException
ArithmeticException
ArrayIndexOutOfBoundsException
SocketException