CS3101-3 Programming Language – Java

Fall 2004

Sept. 29

Road Map today

- Java review
- Homework review
- Exception revisited
- Containers
- I/O

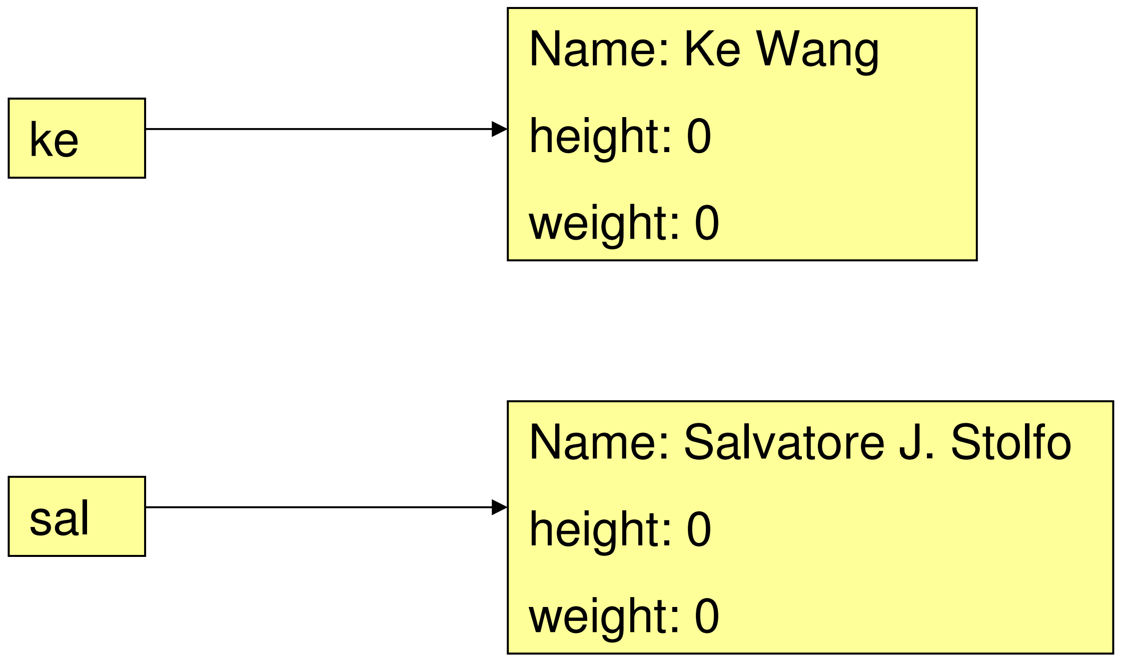
What is Java

- A programming language
- A virtual machine – JVM
- A runtime environment – JRE
  - Predefined libraries
- Portable, but slow
  - Interpreter
  - JIT helps

Object and class

- A class is a blueprint
- An object is an instance created from that blueprint
- All objects of the same class have the same set of attributes
  - Every Person object have name, weight, height
- But different value for those attributes
  - ke.name=Ke Wang, sal.name=Sal Stolfo

Class Person: illustration



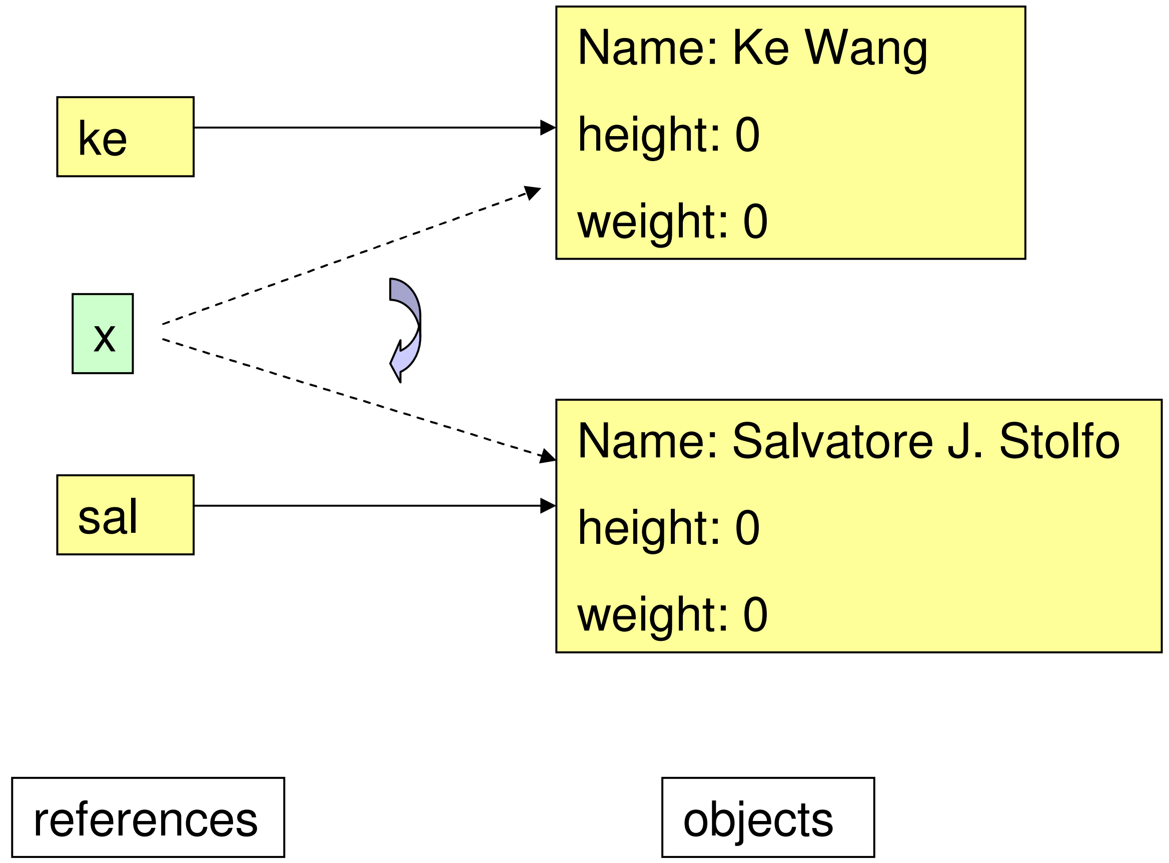
Reference

```
Person ke;          //only created the reference, not an object.
                    It points to nothing now (null).

ke = new Person();          //create the object (allocate storage
                            in memory), and ke is initialized.

ke.name="Ke Wang";          //access the object through
                            the reference
```

Can have multiple reference to one object

No reference means the object is inaccessible forever – goes to garbage collector

Class Person: variables



Arrays in Java: declaration

- Declaration
  - int[] arr;
  - Person[] persons;
  - Also support: int arr[]; Person persons[]; (confusing, should be avoided)
- Creation
  - int[] arr = new int[1024];
  - int [][] arr = { {1,2,3}, {4,5,6} };
  - Person[] persons = new Person[50];

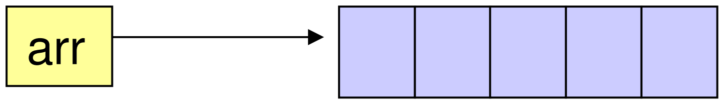
Arrays in Java: safety

- Cannot be accessed outside of its range
  - ArrayIndexOutOfBoundsException
- Guaranteed to be initialized
  - Array of primitive type will be initialized to their default value
    - Zeroes the memory for the array
  - Array of objects: actually it's creating an array of references, and each of them is initialized to ***null.***

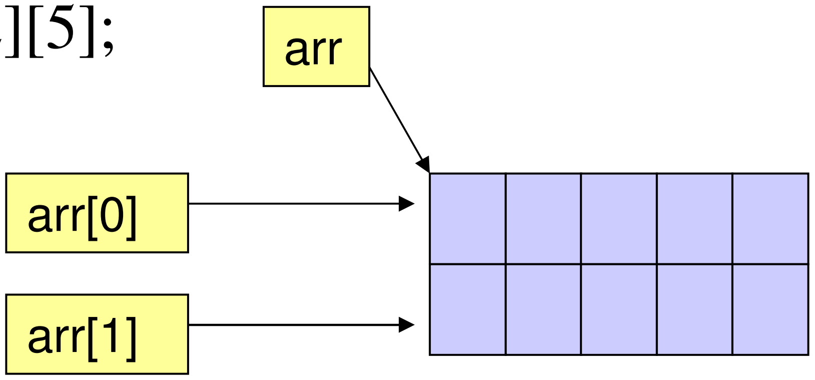
Arrays in Java:

- second kind of reference types in Java

int[] arr = new int [5];



int[][] arr = new int [2][5];



Reference vs. primitive

- Java handle objects and arrays always by reference.
- Java always handle values of the primitive types directly
- differ in two areas:
  - copy value
  - compare for equality

Visibility of fields and methods

- Generally make fields **private** and provide **public** getField() and setField() accessor functions
- O-O term: encapsulation
- Private fields and methods cannot be accessed from outside of the class.

Static vs. non-static

- Static: class variable/method
- Non-static: instance variable/method
- Static ones are associated with class, not object. Can be called using class name directly
- main() is static
    - Even though it’s in a class definition, no instance of the class exist when main starts executing

Static vs. non-static (cont.)

- Instance fields define an object; the values of those fields make one object distinct from another
- Instance method operates on an instance of a class (object) instead of operating on the class itself.
- Class methods can only use class fields; while instance methods can use both instance fields and class fields

Keyword **this**

- Invisible additional parameter to all instance methods
  - Value is the instance through which it was called
    - tc.instanceMethod(); -> this=tc
- Three common usage
  - Same name variable
  - Passing the object myself
  - Call another constructor

Constructor

- Method with same name as class
- No return type
- Called automatically by new()
- Java provides a default one
  - No parameter, default initialization (0/null)
- User can define their own
  - The default one is gone

Method overloading

- Same method name and return type, different parameter list
  - Different type, order, number…
- Return type is NOT enough

Inheritance

- Child class can extend parent class
- Gets all the parent’s fields and methods
  - Private vs. protected
- Can use child for anything that’s expecting parent (upcasting)
  - But not vice-versa
- Can only extend one class
  - No multiple inheritance

Inheritance

- Class Foo extends Bar {
- Can override parent’s implementation
- Other classes that only know Bar can use Foo as well, but not any extra methods Foo added

polymorphism

- We have an array of Shapes, and we ask each Shape to draw() itself
- The correct method will be called
  - The Circle’s draw() method if the Shape object is actually Circle, Square’s draw() if it’s actually a Square
- O-O term: polymorphism

The Master Class

- All Classes extend Object class
- Thus Object references are “universal” references
  - Like void *
- toString()

Abstract classes and interfaces

- Don’t provide implementation for some/all methods
- Can not be instantiated
- Subclasses that wish to be instantiable must implement all abstract/interface methods
- Allows us to provide a “contract” without a default implementation
- Can have references to abstract classes and interfaces
- Can implement as many interface as we want

Encapsulation keywords

- Public: everyone
- Private: same class only
- Protected: self and subclasses

- Control visibility

Keyword final

- Like C++ const
- Field: value cannot be changed once set
  - Does not be to initialized – “blank final”
  - Convention: make name all CAPS, e.g. Math.PI
- Method: cannot be overridden
- Class: cannot be extended

Command line arguments

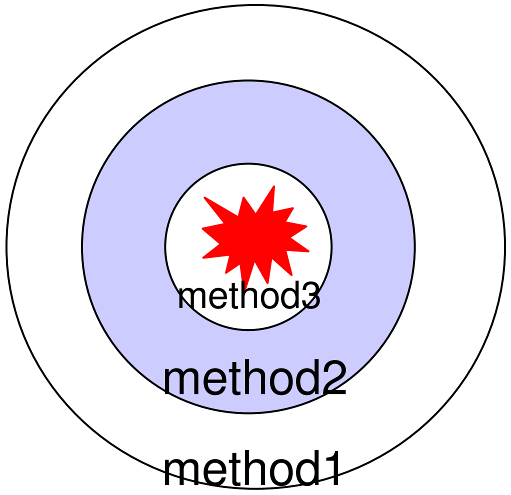
- Show up in that String[] args array passed to main()
- Note the first parameter is args[0]
  - Not the name of the class/program
  - Java Foo param1 param2
  - All in String object, parse if necessary
- Can check args.length to see the number of parameters

Homework review

- HW1: MyDate.java
  - Validity check
  - Leap year
- HW2: AccountTest.java

Explosions

- void method1() {…method2()}
- void method2() {…method3()}
- void method3() {…x=5/0} //BOOM!!



Error handling

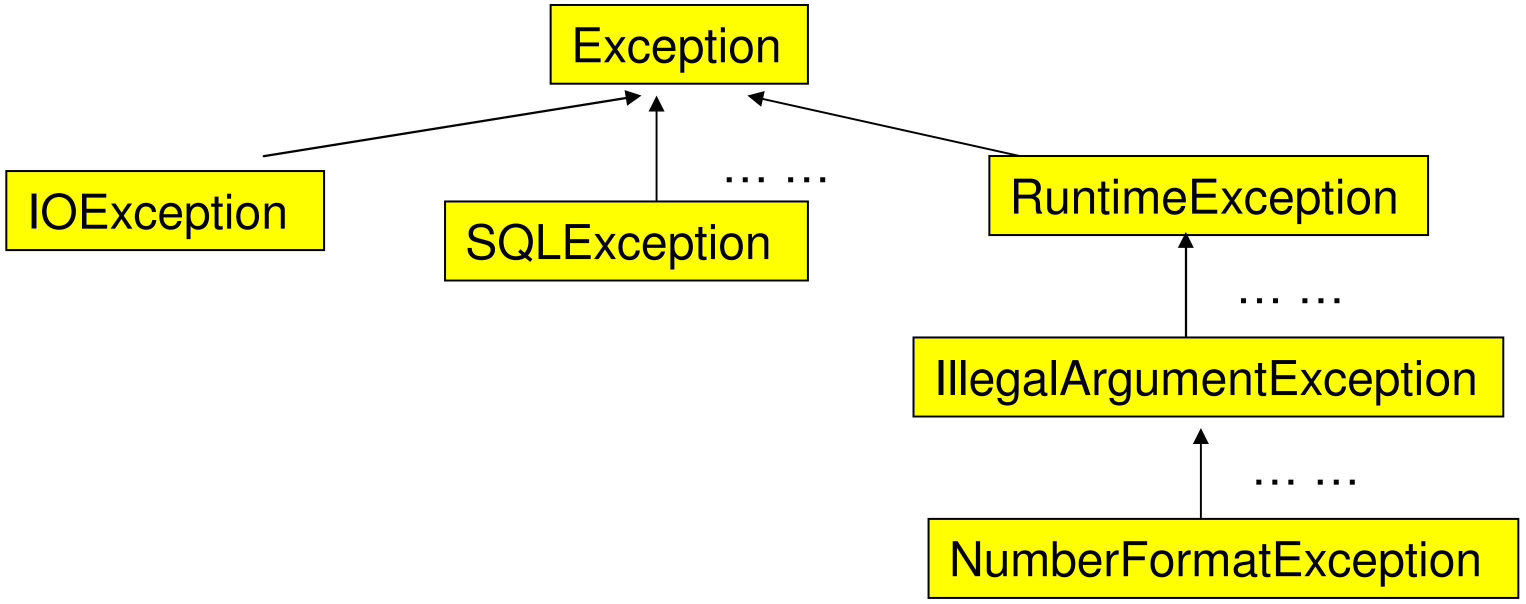
- Java philosophy: “badly formed code will not be run”
- Ideal time to catch error: compile
- Not all errors can be detected at compile time; the rest must be handled at run time
- Java: exception handling
  - The only official way that Java reports error
  - Enforced by compiler

Unexpected situation

- User input errors
- Device errors
- Physics limits
- Programmer errors

Exceptions are objects

- throw new IOException();
- throw new IOException(“file not open”);



Catching an exception

- Guarded region
  - Try block
  - Exception handler

```
try{
     //code that might generate exceptions
} catch (Type1 id1){
    // handle exception for Type1
} catch (Type2 id2){
    // handle exception for Type2
}
```

Only the first catch block with matching exception type will be execute

Create your own exception

- Create your own to denote a special problem
- Example: ExceptionTest.java

RuntimeException

- Always thrown automatically by Java
- You can **only** ignore RuntimeException in coding, all other handling is carefully enforced by compiler
  - RuntimeException represents programming error
    - NullPointerException
    - ArrayIndexOutOfBoundsException
    - NumberFormatException

Finally clause – clean up

- Always execute, regardless of whether the body terminates normally or via exception
- Provides a good place for required cleanup
  - Generally involves releasing resources, for example, close files or connections

```
try{
     //code that might throw A or B exception
} catch (A a){
    // handler for A
} catch (B b){
    //handler for B
} finally {
    //activities that happen every time, do cleanup
}
```

When to use Exception

- 90% of time: because the Java libraries force you to
- Other 10% of the time: your judgement
- Software engineering rule of thumb
  - Your method has *preconditions* and *postcondition*
  - If preconditions are met, but you can’t fulfill your postcondition, throw an exception

Containers

- Hold a group of objects
- Significantly increase your programming power
- All perform bound checking
- array: efficient, can hold primitives
- Collection: a group of individual elements
    - List, Set
- Map: a group of key-value object pairs
    - HashMap
- Misleading: sometimes the whole container libraries are also called collection classes

array

- Most efficient way to do random access
- Size is fixed and cannot be changed for the lifetime
- If run out of space, have to create a new one and copy everything
- Advantage: can hold primitives

Other containers

- Can only take object
- Have to “wrap” primitives
  - int -> Integer, double-> Double
- Have to cast or unwrap on retrieval
- Slow, error prone, tedious….
- Fixed by JDK1.5, hopefully
- Advantage: automatic expanding

Arrays class

- In java.util, a “wrapper” class for array
- A set of static utility methods
  - fill(): fill an array with a value
  - equals(): compare two arrays for equality
  - sort(): sort an array
  - binarySearch(): find one element in a sorted array
- All these methods overload for all primitive types and Object

Arrays.fill()

- Arrays.fill(arrayname, value)
  - Assigns the specified value to each element of the specified array
- Arrays.fill(arrayname, start, end, value)
  - Assigns the specified byte value to each element of the specified range of the specified array
- Value’s type must be the same as, or compatible with the array type

Arrays.fill() Example

```
import java.util.*;

int[] a1=new int[5];
Arrays.fill(a1, 0, 2, 2); // [2, 2, 0, 0, 0]
Arrays.fill(a1, 4);  // [4, 4, 4, 4, 4]
Arrays.fill(a1, 2, 4, 5); //[4, 4, 5, 5, 4]

String[] a2 = new String[5];
Arrays.fill(a2, 1, 5, “hi”); // [null hi hi hi hi]
Arrays.fill(a2, 0, 6, “columbia”); //error! IndexOutOfBound
```

System.arraycopy()

- Overloaded for all types
- *Shallow copy* – only copy reference for objects, copy value for primitives
- (src_array, src_offset, dst_array, dst_offset, num_of_elements)

```
int[] a1=new int[5];
Arrays.fill(a1, 2, 4, 4);  // [0, 0, 4, 4, 0]
int[] a2 = new int[7];
Arrays.fill(a2, 6); // [6, 6, 6, 6, 6, 6, 6]

System.arraycopy(a1, 0, a2, 2, 5);  //a2= [6, 6, 0, 0, 4, 4, 0]
```

Arrays.sort()

- Sorts the objects into ascending order, according to their *natural ordering*
- This sort is guaranteed to be *stable*: equal elements will not be reordered as a result of the sort
- You can specify a range. The range to be sorted extends from index fromIndex, inclusive, to index toIndex, exclusive.
- The objects need to comparable or there is a special comparator

Arrays.sort() cont.

- sort(array), sort(array, fromIndex, toIndex)
- All elements in the array must implement the *Comparable* interface

- sort(array, comparator)
- sort(array, fromIndex, toIndex, comparator)
- All elements in the array must be *mutually comparable* by the specified comparator

Comparable interface

- With a single method compareTo()
- Takes another Object as argument
- And returns:
  - Negative value if *this* is less than argument
  - Zero value if *this* is equal to argument
  - positive value if *this* is greater than argument

Comparator interface

- Two methods: compare(), equals()
- Only need to implement compare()
- Takes two Object as argument: compare(Object o1, Object o2)
- And returns
  - Negative value if o1 is less than o2
  - Zero value if o1 is equal to o2
  - positive value if o1 is greater than o2

Sort example: compareExp.java

Array.binarySearch()

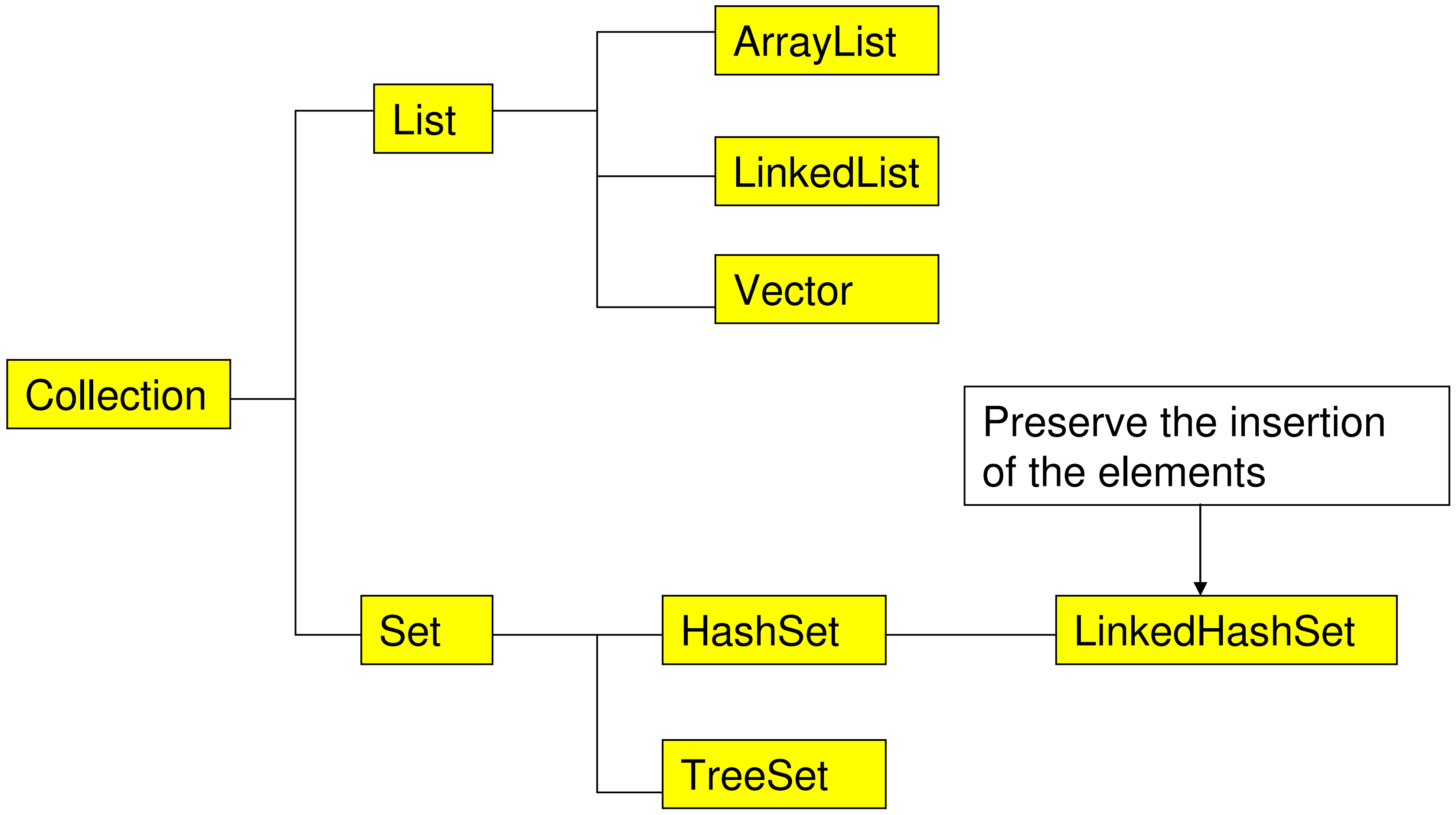
- Only usable on sorted array!
  - Otherwise, result unpredictable
- If there are multiple elements equal to the specified object, there is no guarantee which one will be found.
- Return:
  - Location if find the key (positive number)
  - (-(*insertion point*) - 1) if not find key (negative)

search example: compareExp.java

Collection: hold one item at each location

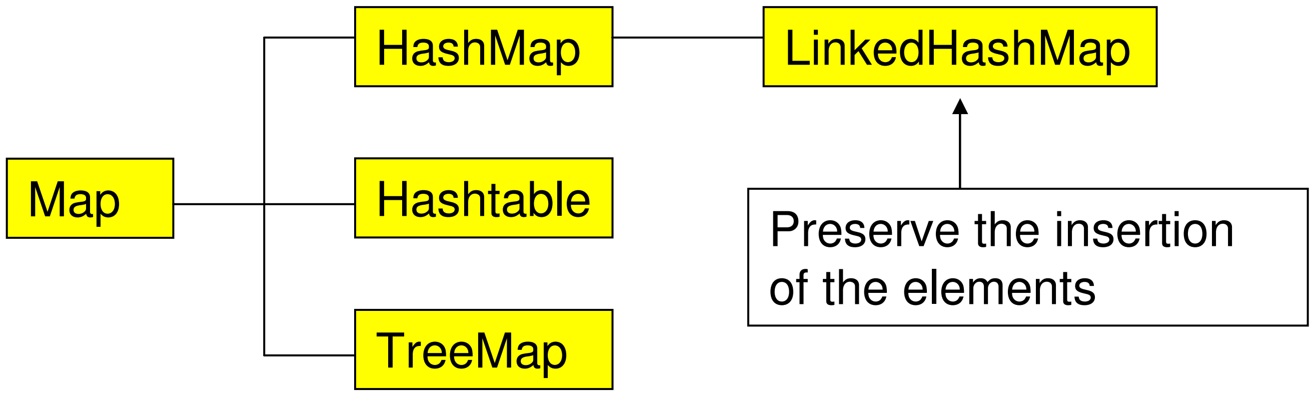
List: items in order

Set: no duplicates, no ordering



Map: key-value pairs, fast retrieval

no duplicate keys, no ordering



Disadvantages of container

- Cannot hold primitives
  - Have to wrap it
- Lose type information when put object into container
  - Everything is just Object type once in container
- Have to do cast when get it out
  - You need to remember what’s inside
- Java do run time type check
  - ClassCastException

ArrayList

- An array that automatically expand itself
- Put objects using add()
- Get out using get(int index)
  - Need to cast type
- Method size() to get the number of objects
  - Similar to .length attribute of array
- Example: CatsAndDogs.java

Iterator object

- Access method regardless of the underlying structure
- Generic programming
  - Can change underlying structure easily
- “light-weight” object
  - Cheap to create
- Can move in only one direction

Iterator constraints

- Container.**iterator**() returns you an Iterator, which is ready to return the first element in the sequence on your first call to **next**()
- Get the next object in the sequence with **next**()
- Set there are more objects in the sequence with **hasNext**()
- Remove the last element returned by the iterator with **remove**()
- Example: revisit CatsAndDogs.java

ArrayList vs. LinkedList

- ArrayList
  - Rapid random access
  - Slow when inserting or removing in the middle
- LinkedList
  - Optimal sequential access
  - Fast insertion and deletion from the middle
  - addFirst(), addLast(), getFirst(), removeFirst()
  - Easy to be used as queue, stack

Set interface

- Each element add to the Set must be unique, otherwise won’t add.
- Objects added to Set must define equals() to establish object uniqueness
- Not maintain order

Set

- HashSet
  - Fast lookup time by hashing function
- TreeSet
  - Ordered Set backed by a tree (red-black tree)
  - Can extract ordered sequence
- LinkedHashSet
  - Has the lookup speed of a HashSet
  - Maintain the insertion order by linked list

Set example

- revisit CatsAndDogs.java

Map interface

- Key-value associative array
- Look up object using another object
  - Array use index
- put(Object key, Object value)
- Object get(Object key)
- containsKey(), containsValue()

Map

- HashMap
  - Based on a hash table
  - Constant time for insertion and locating pairs
  - Most commonly used
- LinkedHashMap
  - Like HashMap
  - When iterate through, get pairs in insertion order
- TreeMap
  - Based on red-black tree, can viewed in order
  - Need to implement Comparable or Comparator

Map example

- MapExample.java