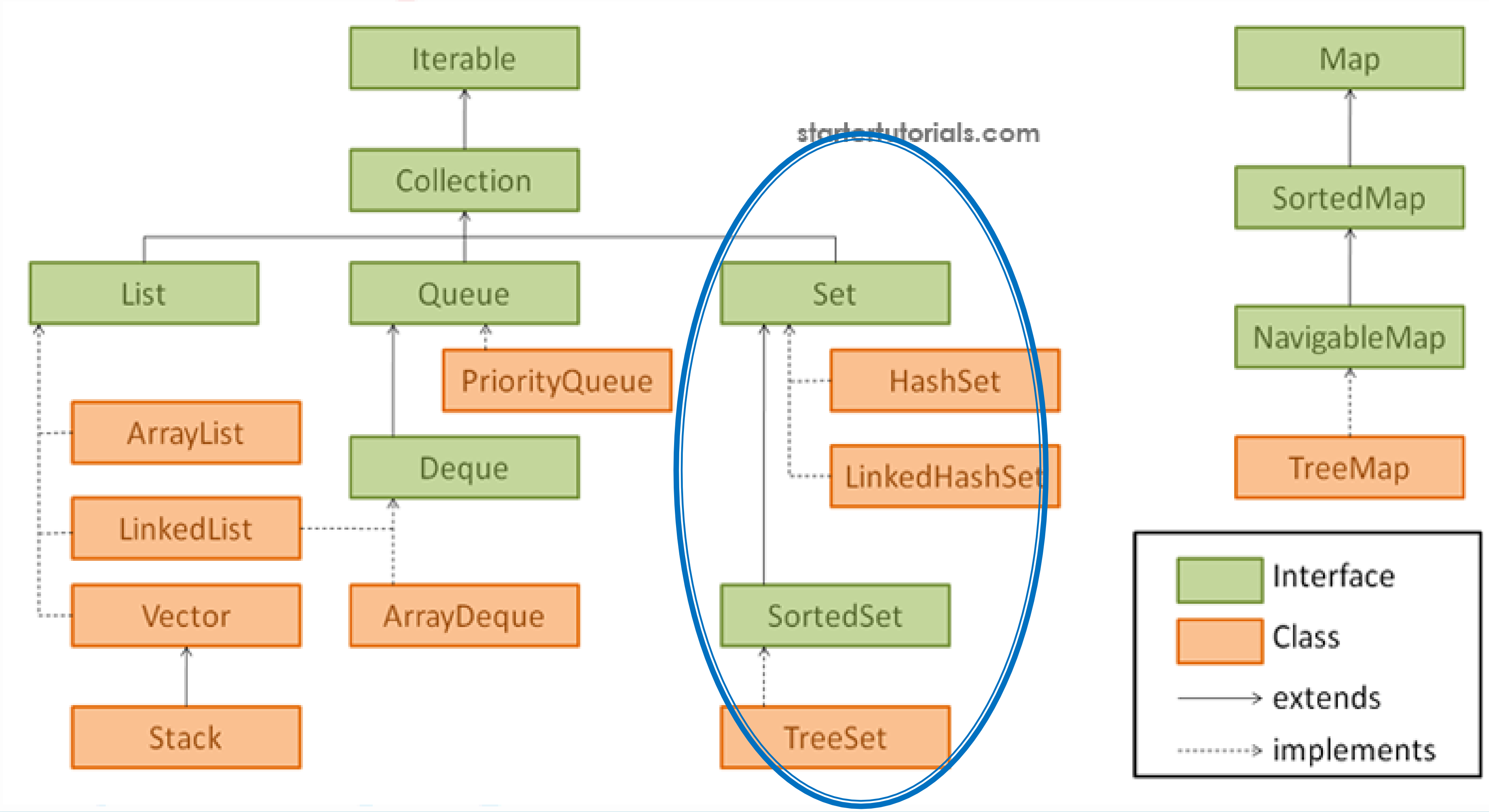
FACULTY OF INFORMATION TECHNOLOGY

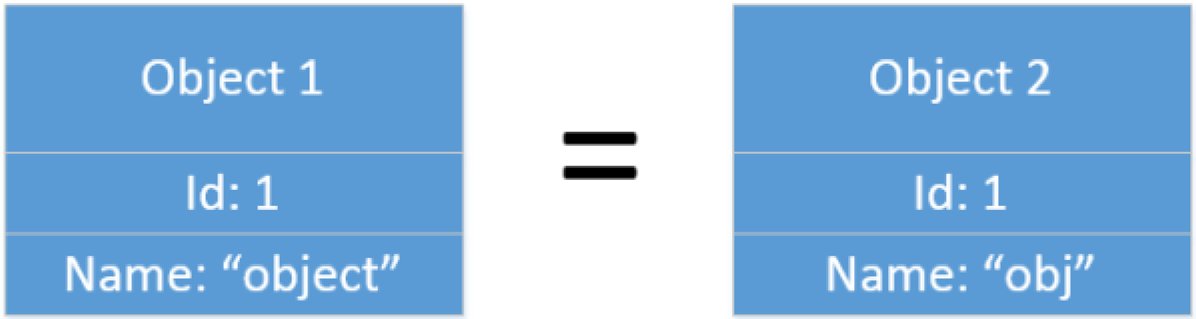
DATA STRUCTURES (CTDL)

Semester 1, 2021/2022

Java Collection framework



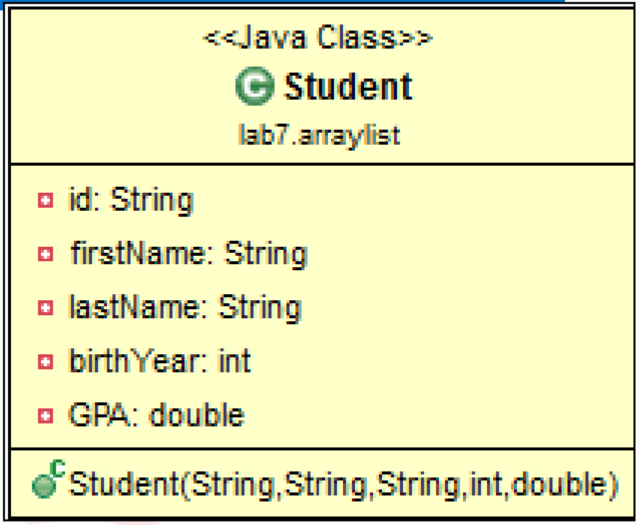
Objects Equality



and Algorithms
http://www.javaguides.net

Example

- For a given Student class as follows:

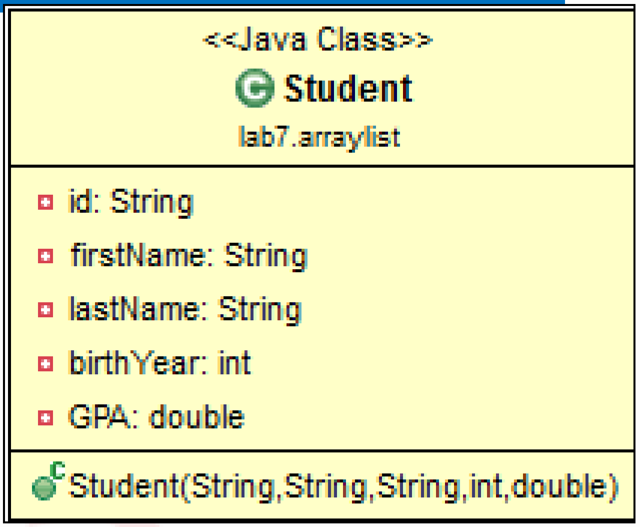


- What is the result of the following code fragment.

```
public static void main(String[] args) {
    Student st1 = new Student("18130006", "Binh", "Nguyen", 1996, 8.5);
    Student st2 = st1;
    Student st3 = new Student("18130006", "Binh", "Nguyen", 1996, 8.5);

    System.out.println(st1 == st2);// ???
    System.out.println(st1 == st3);// ???
    System.out.println(st1.equals(st2));// ???
    System.out.println(st1.equals(st3));// ???
}
```

Example

- For a given Student class as follows:



- What is the result of the following code fragment.

```
public static void main(String[] args) {
    Student st1 = new Student("18130006", "Binh", "Nguyen", 1996, 8.5);
    Student st2 = st1;
    Student st3 = new Student("18130006", "Binh", "Nguyen", 1996, 8.5);

    System.out.println(st1 == st2);// ???
    System.out.println(st1 == st3);// ???
    System.out.println(st1.equals(st2));// ???
    System.out.println(st1.equals(st3));// ???
}
```

true
false
true
false

Approaches to comparing 2 Objects

- Two ways to compare equality of two Objects:
  - **Shallow comparison**:
    - The default implementation of equals method in Java.lang.Object class
    - Simply, it checks if x == y, meaning two Object references (say x and y) refer to the same Object
  - **Deep Comparison**:
    - A class provides its own implementation of equals() method in order to compare the Objects of that class w.r.t state of the Objects.
    - Data members (i.e. fields) of Objects are to be compared with one another.

Java Data Structures and Algorithms http://www.javaguides.net

Equality comparison for primitives

- Comparison a == b, a != b:
  - Primitives:
    - Equal values
  - Objects: Compares references, not values. The use of == with object references is generally limited to the following:
    - Comparing to see if a reference is null.
    - Comparing two enum values. This works because there is only one object for each enum constant.
    - Comparing to see if two references are to the *same object*

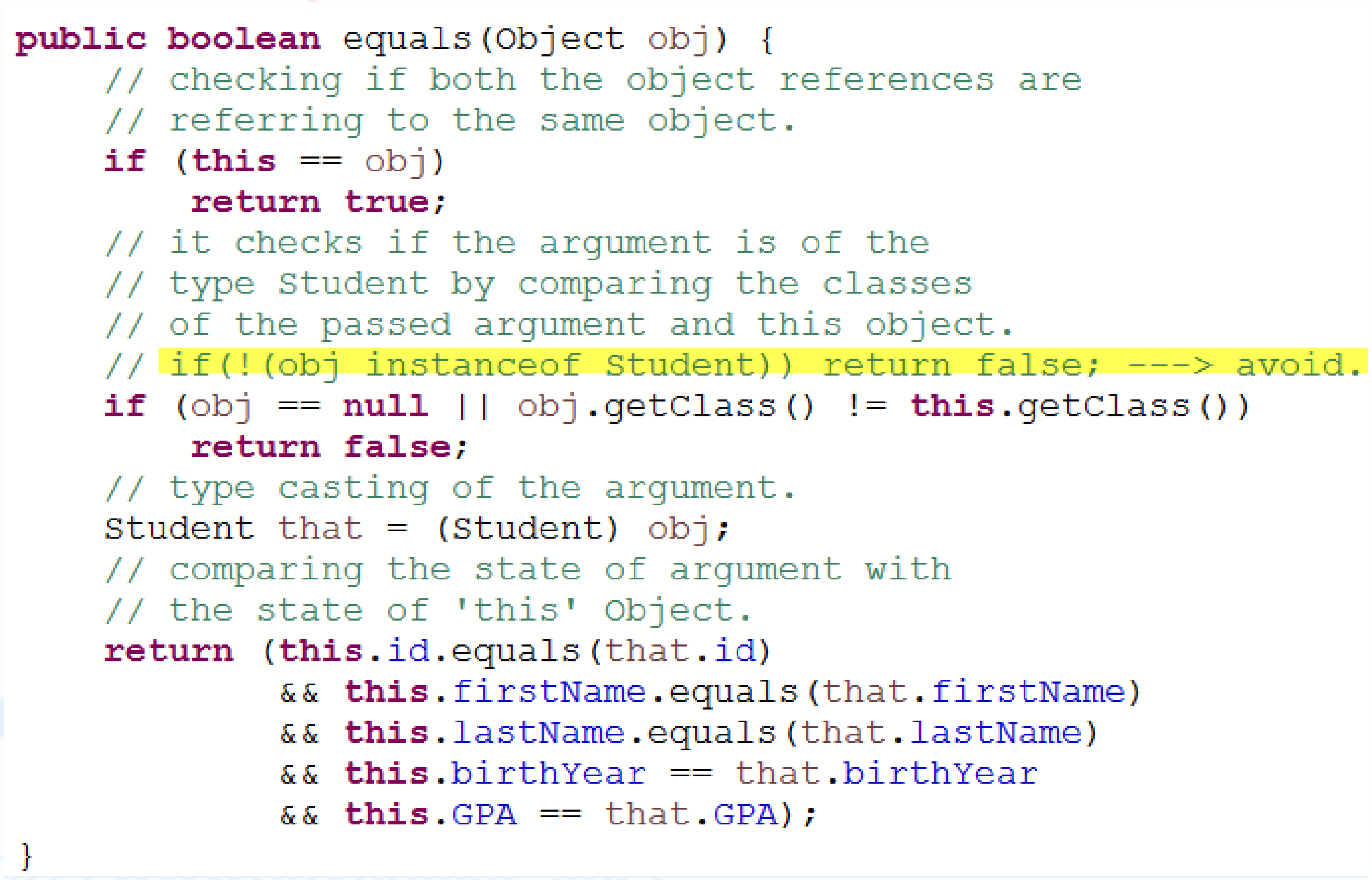
Java Data Structures and Algorithms http://www.javaguides.net

Principles of equals

- If some other object is equal to a given object, then it follows these rules:
  - Reflexive: for any reference value a, a.equals(a) should return true.
  - Symmetric: for any reference values a and b, if a.equals(b) should return true then b.equals(a) must return true.
  - Transitive: for any reference values a, b, and c, if a.equals(b) returns true and b.equals(c) returns true, then a.equals(c) should return true.
  - Consistent: for any reference values a and b, multiple invocations of a.equals(b) consistently return true or consistently return false, provided no information used in equals comparisons on the object is modified.

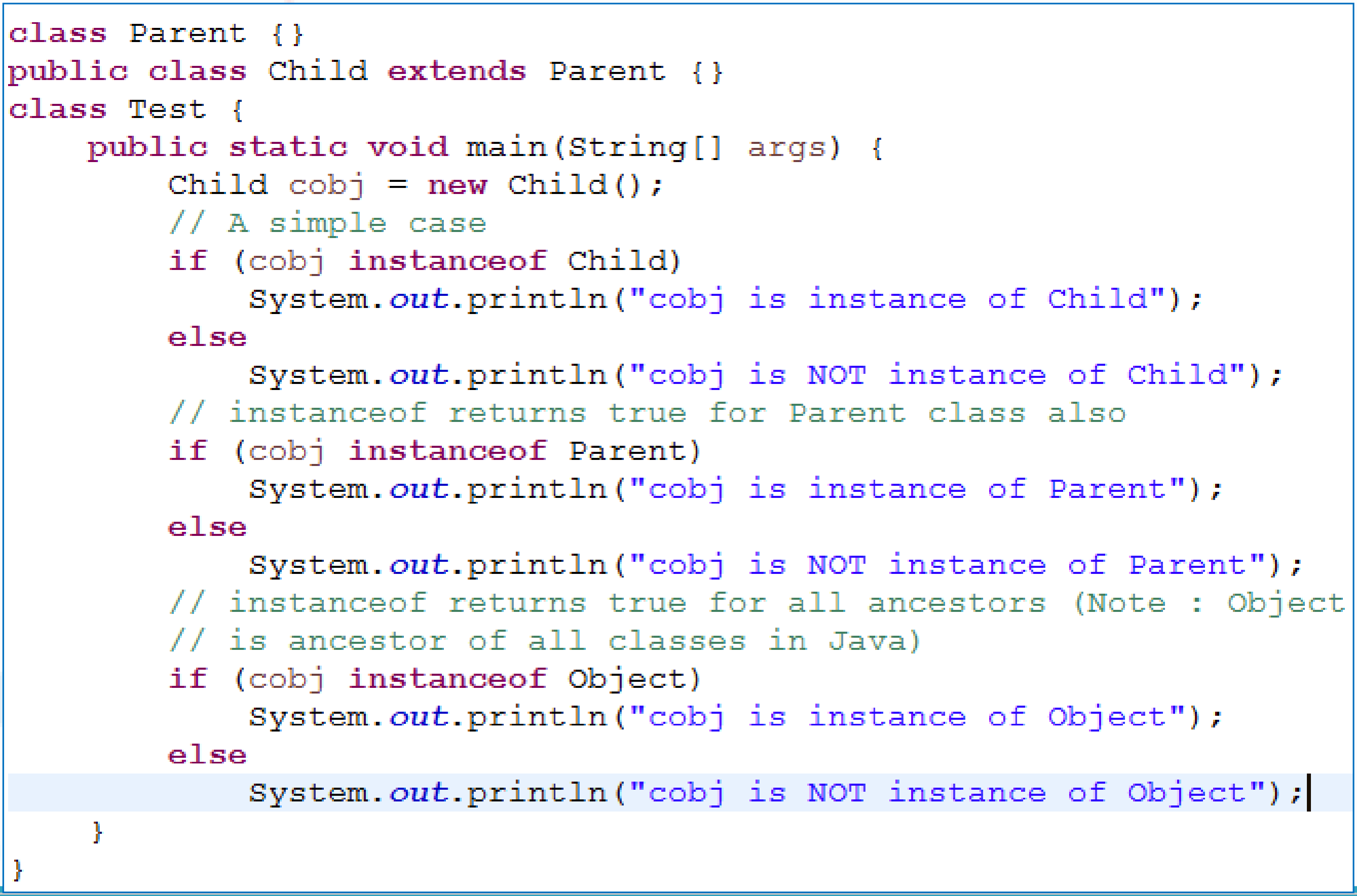
Java Data Structures and Algorithms http://www.javaguides.net

Equals method implementation



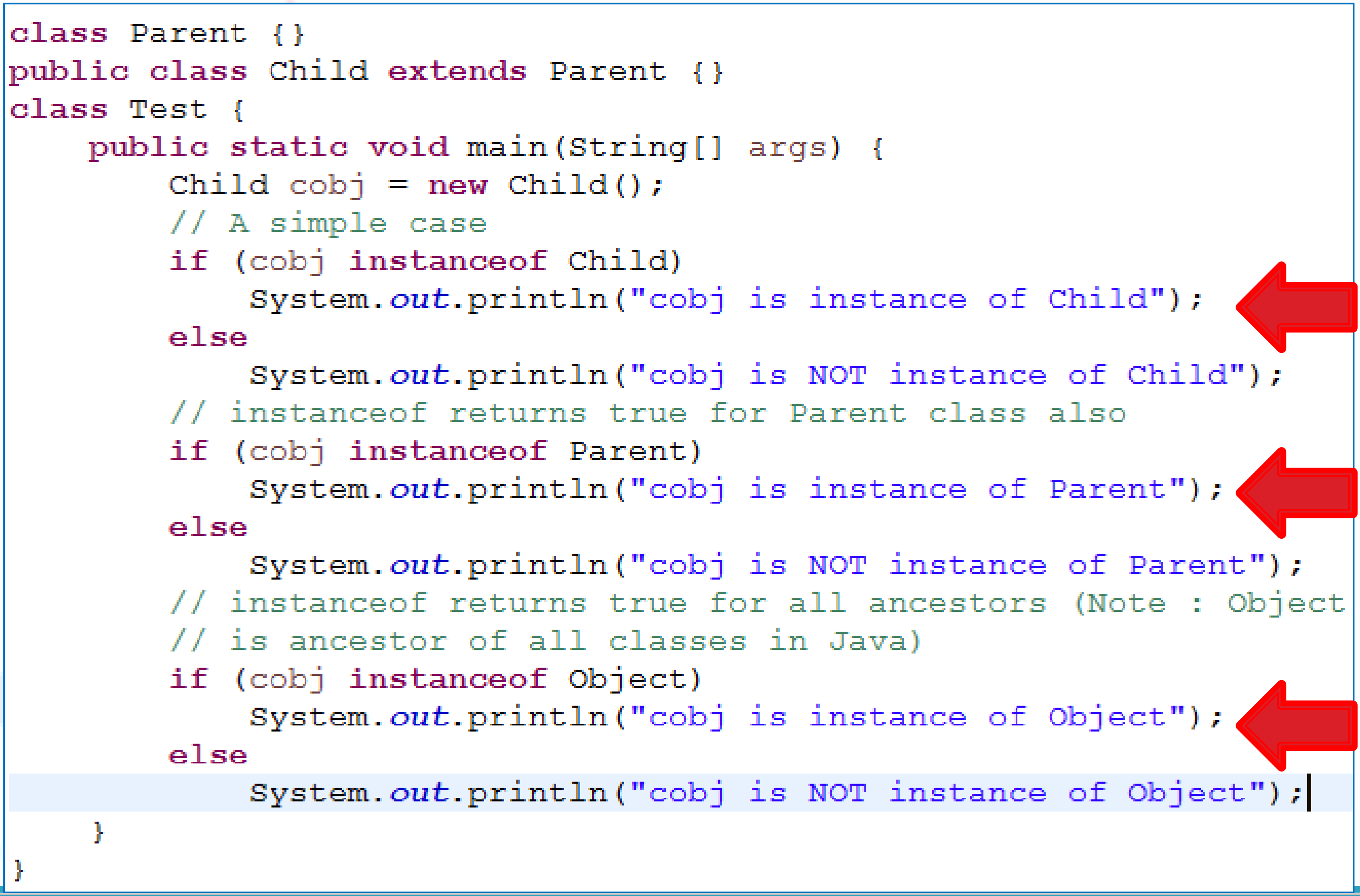
```
public boolean equals(Object obj) {
    // checking if both the object references are
    // referring to the same object.
    if (this == obj)
        return true;
    // it checks if the argument is of the
    // type Student by comparing the classes
    // of the passed argument and this object.
    // if(!(obj instanceof Student)) return false; ---> avoid.
    if (obj == null || obj.getClass() != this.getClass())
        return false;
    // type casting of the argument.
    Student that = (Student) obj;
    // comparing the state of argument with
    // the state of 'this' Object.
    return (this.id.equals(that.id)
            && this.firstName.equals(that.firstName)
            && this.lastName.equals(that.lastName)
            && this.birthYear == that.birthYear
            && this.GPA == that.GPA);
}
```

Why avoiding instanceof?



```
class Parent {}
public class Child extends Parent {}
class Test {
    public static void main(String[] args) {
        Child cobj = new Child();
        // A simple case
        if (cobj instanceof Child)
            System.out.println("cobj is instance of Child");
        else
            System.out.println("cobj is NOT instance of Child");
        // instanceof returns true for Parent class also
        if (cobj instanceof Parent)
            System.out.println("cobj is instance of Parent");
        else
            System.out.println("cobj is NOT instance of Parent");
        // instanceof returns true for all ancestors (Note : Object
        // is ancestor of all classes in Java)
        if (cobj instanceof Object)
            System.out.println("cobj is instance of Object");
        else
            System.out.println("cobj is NOT instance of Object");
    }
}
```

Why avoiding instanceof?



```java
class Parent {}
public class Child extends Parent {}
class Test {
    public static void main(String[] args) {
        Child cobj = new Child();
        // A simple case
        if (cobj instanceof Child)
            System.out.println("cobj is instance of Child");
        else
            System.out.println("cobj is NOT instance of Child");
        // instanceof returns true for Parent class also
        if (cobj instanceof Parent)
            System.out.println("cobj is instance of Parent");
        else
            System.out.println("cobj is NOT instance of Parent");
        // instanceof returns true for all ancestors (Note : Object
        // is ancestor of all classes in Java)
        if (cobj instanceof Object)
            System.out.println("cobj is instance of Object");
        else
            System.out.println("cobj is NOT instance of Object");
    }
}
```

Notes on instanceof

- A parent object is not an instance of Child

```
Parent pobj = new Parent();
if (pobj instanceof Child)
   System.out.println("pobj is instance of Child");
else
   System.out.println("pobj is NOT instance of Child");
```

- A parent reference referring to a Child is an instance of Child

```
// Reference is Parent type but object is
// of child type.
Parent cobj = new Child();
if (cobj instanceof Child)
   System.out.println("cobj is instance of Child");
else
   System.out.println("cobj is NOT instance of Child");
```

http://www.javaguides.net

Notes on instanceof

- A parent object is not an instance of Child

```
Parent pobj = new Parent();
if (pobj instanceof Child)
    System.out.println("pobj is instance of Child");
else
    System.out.println("pobj is NOT instance of Child");
```

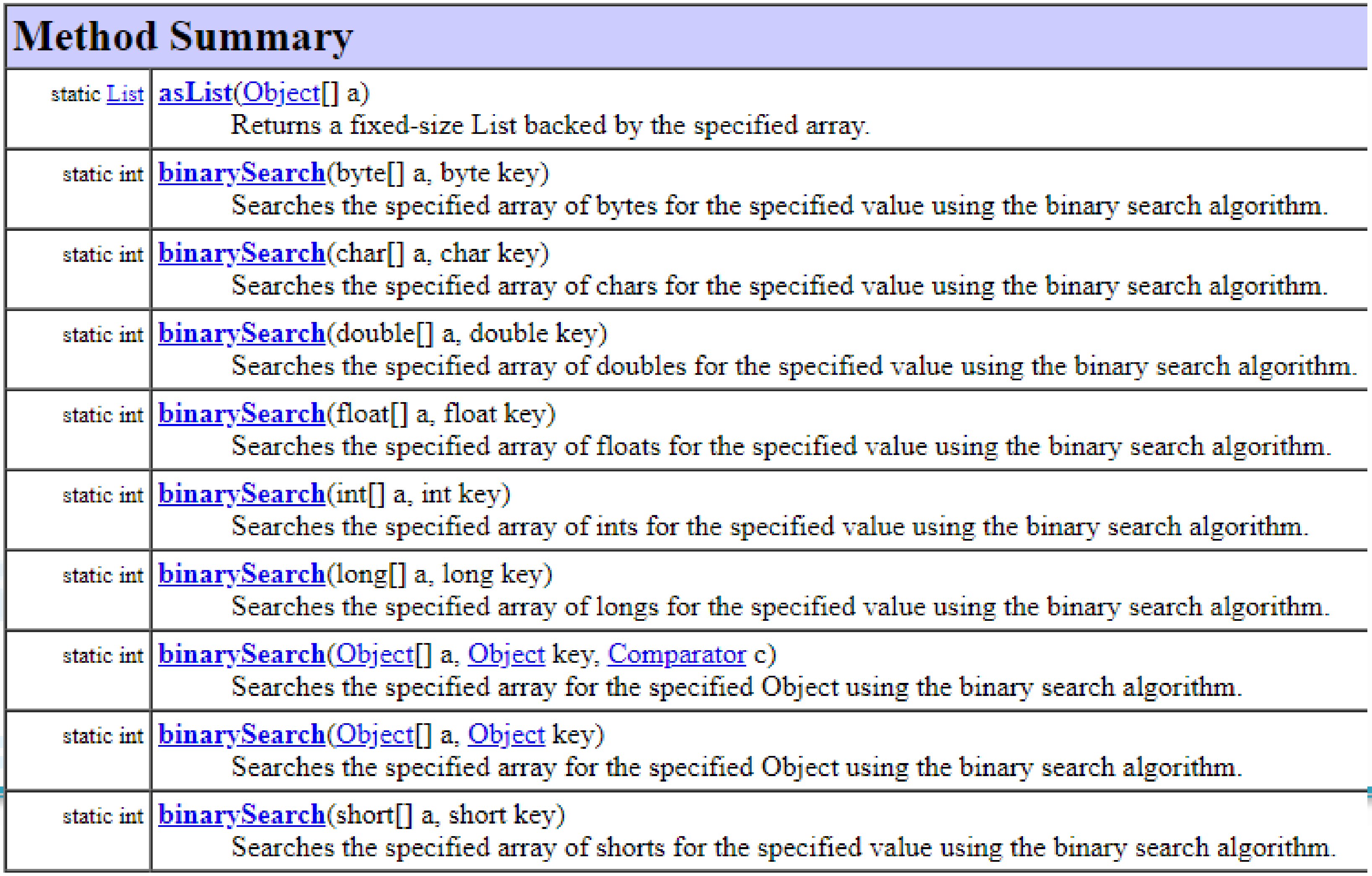
- A parent reference referring to a Child is an instance of Child

```
// Reference is Parent type but object is
// of child type.
Parent cobj = new Child();
if (cobj instanceof Child)
    System.out.println("cobj is instance of Child");
else
    System.out.println("cobj is NOT instance of Child");
```

http://www.javaguides.net

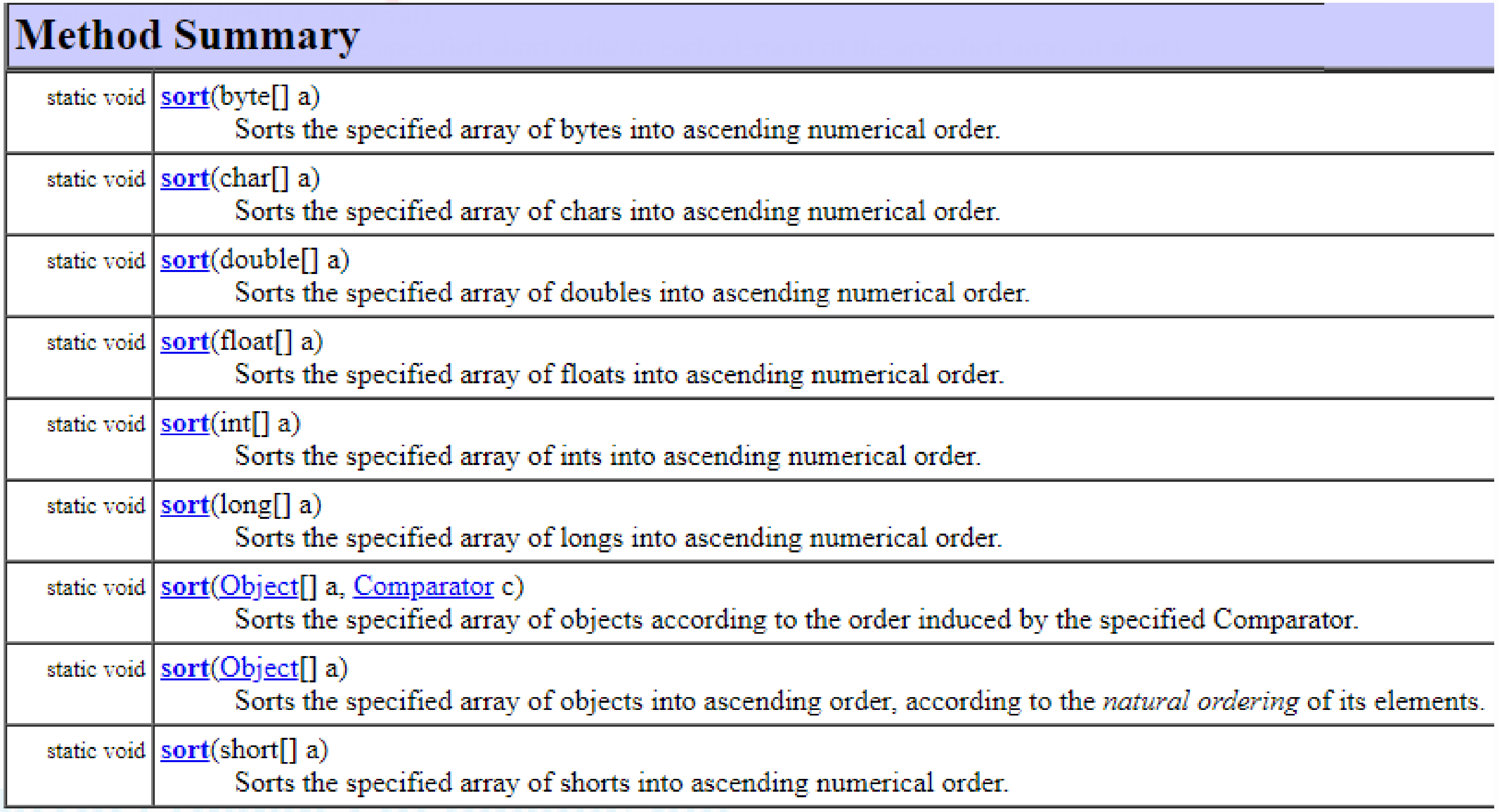
Class java.util.Arrays

- This class contains various methods for manipulating arrays (such as sorting and searching)



Method Summary	
static List	asList(Object[] a) Returns a fixed-size List backed by the specified array.
static int	binarySearch(byte[] a, byte key) Searches the specified array of bytes for the specified value using the binary search algorithm.
static int	binarySearch(char[] a, char key) Searches the specified array of chars for the specified value using the binary search algorithm.
static int	binarySearch(double[] a, double key) Searches the specified array of doubles for the specified value using the binary search algorithm.
static int	binarySearch(float[] a, float key) Searches the specified array of floats for the specified value using the binary search algorithm.
static int	binarySearch(int[] a, int key) Searches the specified array of ints for the specified value using the binary search algorithm.
static int	binarySearch(long[] a, long key) Searches the specified array of longs for the specified value using the binary search algorithm.
static int	binarySearch(Object[] a, Object key, Comparator c) Searches the specified array for the specified Object using the binary search algorithm.
static int	binarySearch(Object[] a, Object key) Searches the specified array for the specified Object using the binary search algorithm.
static int	binarySearch(short[] a, short key) Searches the specified array of shorts for the specified value using the binary search algorithm.

Class java.util.Arrays

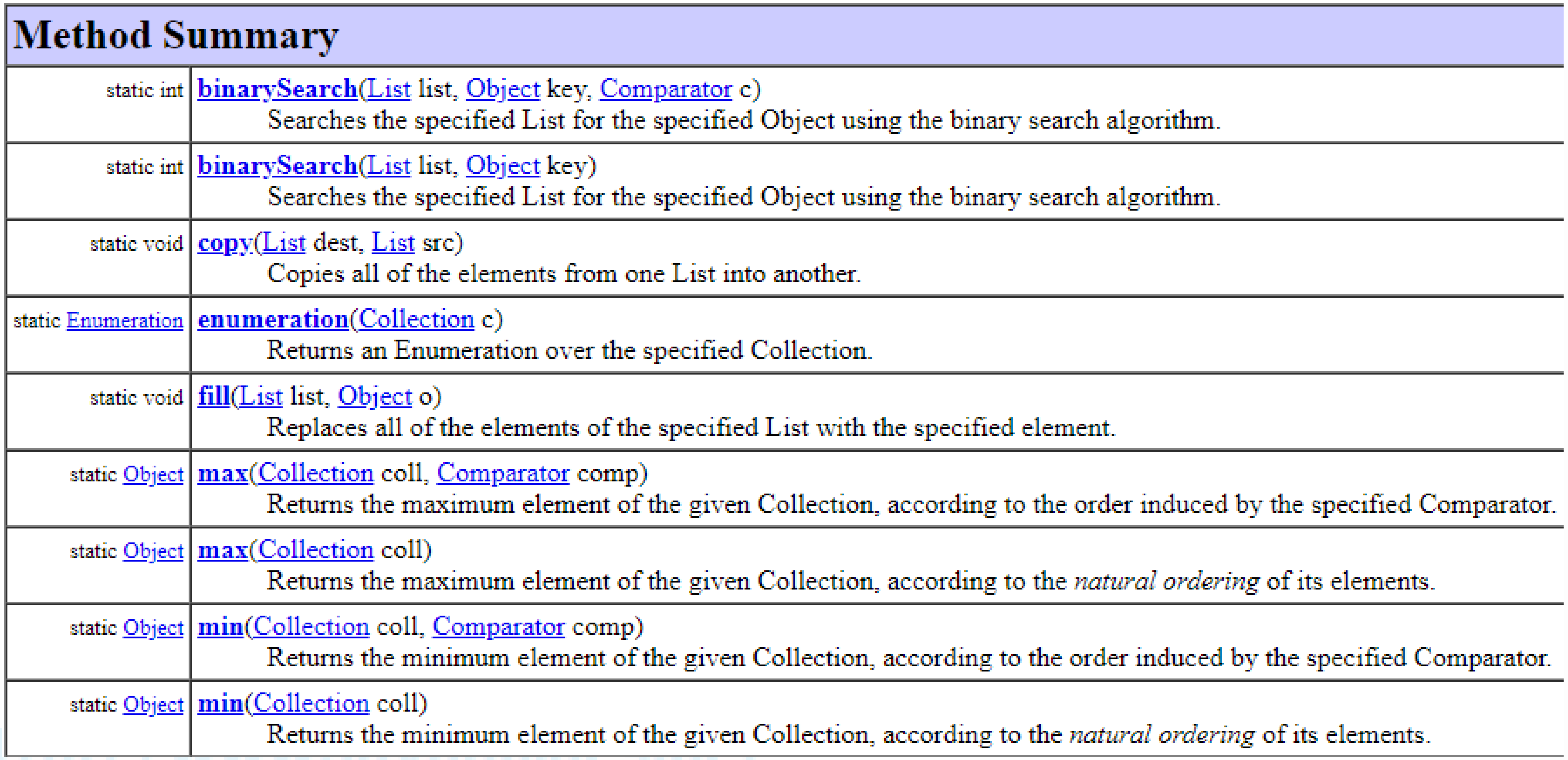


Method Summary	
static void	sort(byte[] a) Sorts the specified array of bytes into ascending numerical order.
static void	sort(char[] a) Sorts the specified array of chars into ascending numerical order.
static void	sort(double[] a) Sorts the specified array of doubles into ascending numerical order.
static void	sort(float[] a) Sorts the specified array of floats into ascending numerical order.
static void	sort(int[] a) Sorts the specified array of ints into ascending numerical order.
static void	sort(long[] a) Sorts the specified array of longs into ascending numerical order.
static void	sort(Object[] a, Comparator c) Sorts the specified array of objects according to the order induced by the specified Comparator.
static void	sort(Object[] a) Sorts the specified array of objects into ascending order, according to the *natural ordering* of its elements.
static void	sort(short[] a) Sorts the specified array of shorts into ascending numerical order.

http://www.javaguides.net

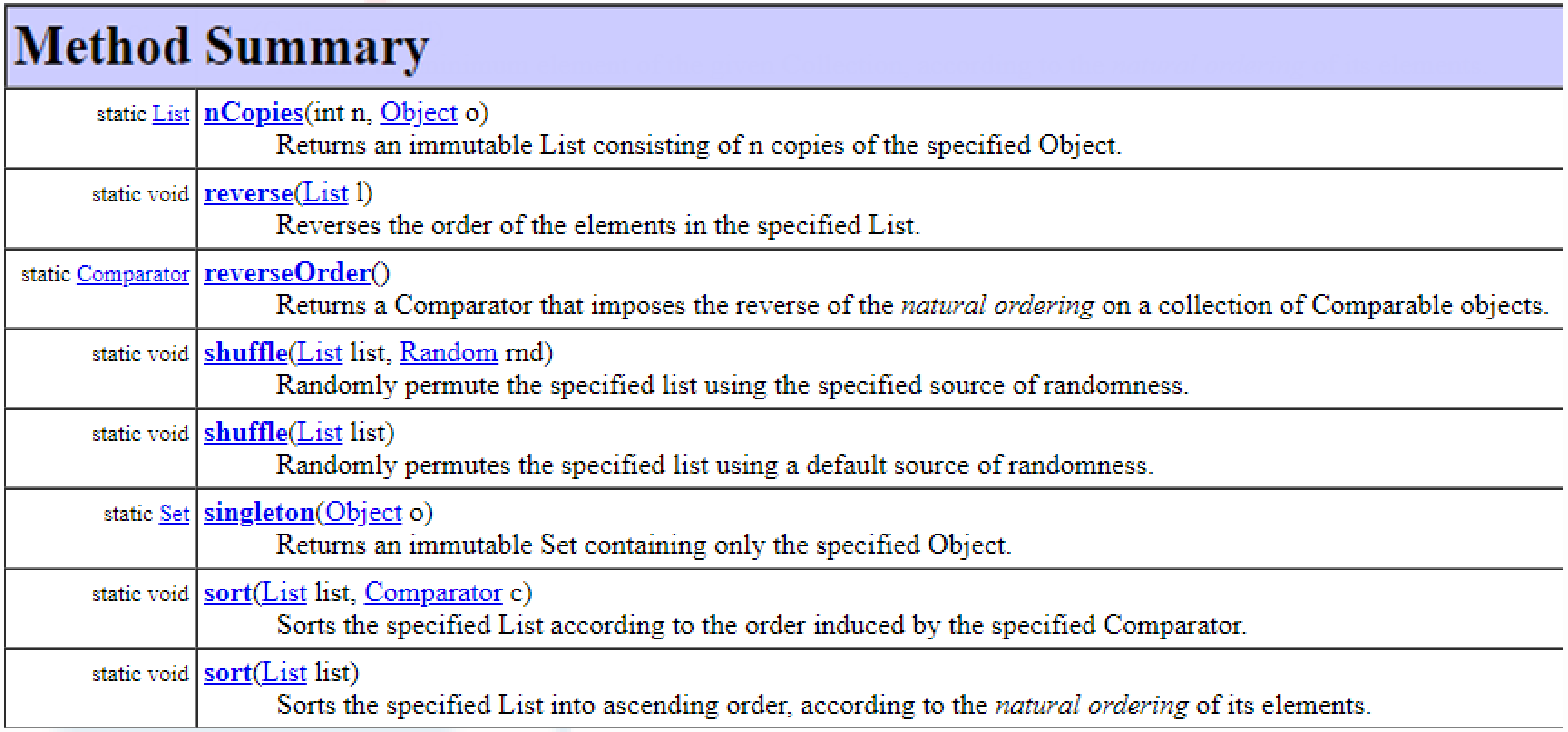
Class java.util.Collections

- This class consists exclusively of static methods that operate on or return Collections.



Method Summary	
static int	**binarySearch**(List list, Object key, Comparator c) Searches the specified List for the specified Object using the binary search algorithm.
static int	**binarySearch**(List list, Object key) Searches the specified List for the specified Object using the binary search algorithm.
static void	**copy**(List dest, List src) Copies all of the elements from one List into another.
static Enumeration	**enumeration**(Collection c) Returns an Enumeration over the specified Collection.
static void	**fill**(List list, Object o) Replaces all of the elements of the specified List with the specified element.
static Object	**max**(Collection coll, Comparator comp) Returns the maximum element of the given Collection, according to the order induced by the specified Comparator.
static Object	**max**(Collection coll) Returns the maximum element of the given Collection, according to the *natural ordering* of its elements.
static Object	**min**(Collection coll, Comparator comp) Returns the minimum element of the given Collection, according to the order induced by the specified Comparator.
static Object	**min**(Collection coll) Returns the minimum element of the given Collection, according to the *natural ordering* of its elements.

Class java.util.Collections



Method Summary	
static List	**nCopies**(int n, Object o) Returns an immutable List consisting of n copies of the specified Object.
static void	**reverse**(List l) Reverses the order of the elements in the specified List.
static Comparator	**reverseOrder**() Returns a Comparator that imposes the reverse of the *natural ordering* on a collection of Comparable objects.
static void	**shuffle**(List list, Random rnd) Randomly permute the specified list using the specified source of randomness.
static void	**shuffle**(List list) Randomly permutes the specified list using a default source of randomness.
static Set	**singleton**(Object o) Returns an immutable Set containing only the specified Object.
static void	**sort**(List list, Comparator c) Sorts the specified List according to the order induced by the specified Comparator.
static void	**sort**(List list) Sorts the specified List into ascending order, according to the *natural ordering* of its elements.

and Algorithms
http://www.javaguides.net

Objects Comparison

http://www.javaguides

Comparing objects

- Operators like < and > do not work with objects in Java.
  - But we do think of some types as having an ordering (e.g. `Date`s).

- **natural ordering**: Rules governing the relative placement of all values of a given type.
  - Implies a notion of equality (like `equals`) but also < and > .
  - **total ordering**: All elements can be arranged in $A \leq B \leq C \leq \ldots$ order.

- **comparison function**: Code that, when given two objects *A* and *B* of a given type, decides their relative ordering:
  - A < B,   A == B,   A > B

http://www.javaguides.net
Java Data Structures and Algorithms

Objects Comparison

» Comparable interface

Java
Data Structures and Algorithms
http://www.javaguides.net

The Comparable interface

- The standard way for a Java class to define a comparison function for its objects is to implement the `Comparable` interface.

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

- A call of **A**.`compareTo`(**B**) should return:

  a value < 0 if A comes "before" B in the ordering,

  a value > 0 if A comes "after" B in the ordering,

  or exactly 0 if A and B are considered "equal" in the ordering. ➔Consider implementing `Comparable`.

Data Structures and Algorithms
http://www.javaguides.net

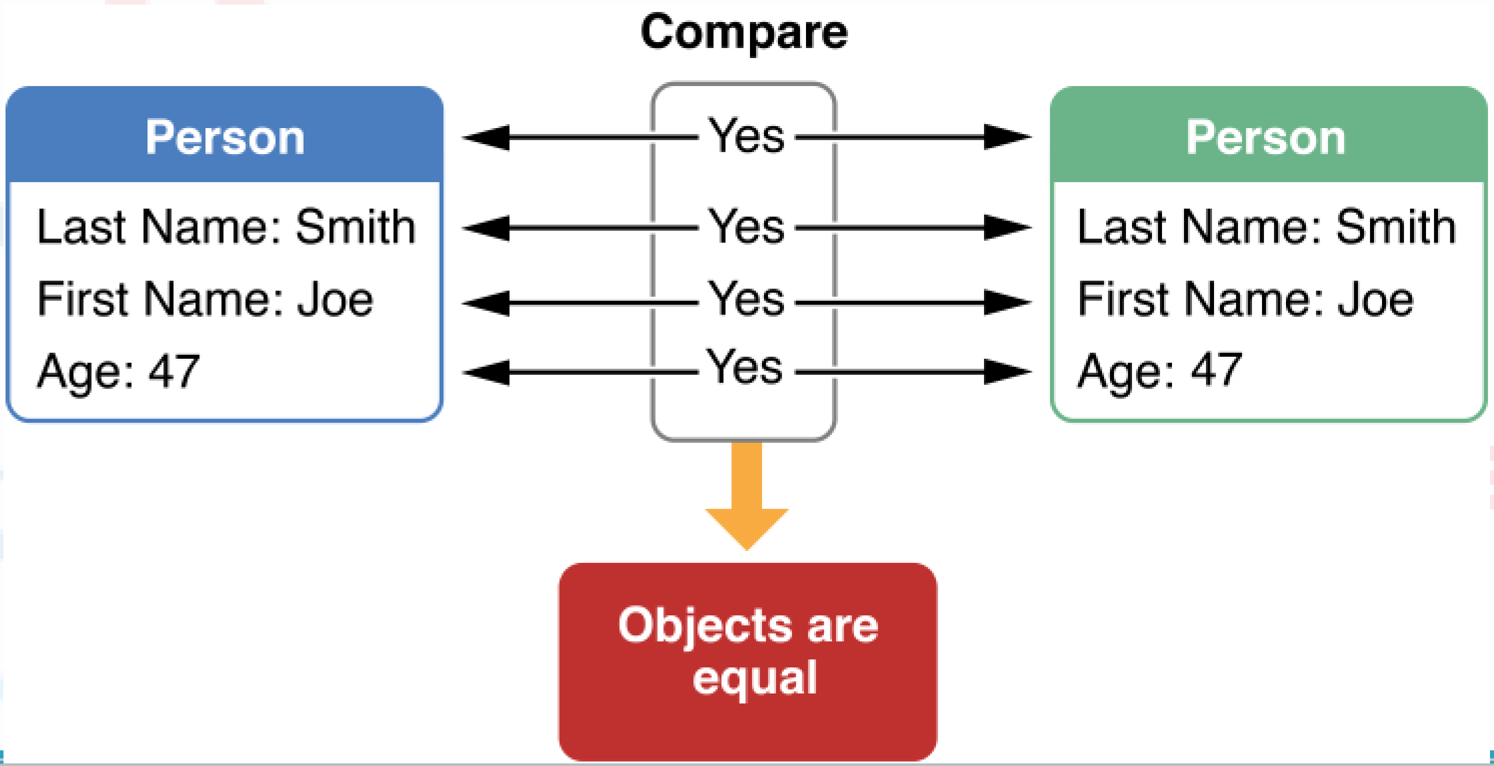
compareTo example

```
public class Point implements Comparable<Point> {
    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;    // same x, smaller y
        } else if (y > other.y) {
            return 1;     // same x, larger y
        } else {
            return 0;     // same x and same y
        }
    }

    // subtraction trick:
    // return (x != other.x) ? (x - other.x) : (y - other.y);
}
```

Java Data Structures and Algorithms http://www.javaguides.net

compareTo and equals

- `compareTo` should generally be consistent with `equals`.
  - `a.compareTo(b) == 0` should imply that `a.equals(b)`.



compareTo and equals

- Employee e = new Employee(...);
- e.compareTo(null);//???
- e.equals(null);//???

- Note that null is not an instance of any class, thus:
  - e.compareTo(null) should throw a NullPointerException,
  - even though e.equals(null) returns false.

Java Data Structures and Algorithms http://www.javaguides.net

What's the "natural" order?

```
public class Rectangle implements Comparable<Rectangle> {
    private int x, y, width, height;

    public int compareTo(Rectangle other) {
        // ...?
    }
}
```

- What is the "natural ordering" of rectangles?
  - By x, breaking ties by y?
  - By width, breaking ties by height?
  - By area? By perimeter?

- Do rectangles have any "natural" ordering?
  - Might we ever want to sort rectangles into some order anyway?

java Data Structures and Algorithms http://www.javaguides.net

Objects Comparison

- Comparator interface

Java Data Structures and Algorithms http://www.javaguides.net

Comparator interface

```
public interface Comparator<T> {
    public int compare(T first, T
  second);
}
```

- Interface `Comparator` is an external object that specifies a comparison function over some other type of objects.
  - Allows you to define multiple orderings for the same type.
  - Allows you to define a specific ordering for a type even if there is no obvious "natural" ordering for that type.

Java Data Structures and Algorithms http://www.javaguides.net

Comparator examples

```
public class RectangleAreaComparator
        implements Comparator<Rectangle> {
// compare in ascending order by area (WxH)

public int compare(Rectangle r1, Rectangle r2) {
      return r1.getArea() - r2.getArea();
    }
}
```

Java Data Structures and Algorithms http://www.javaguides.net

Comparator examples (cont.)

```
public class RectangleXYComparator
        implements Comparator<Rectangle> {
    // compare by ascending x, break ties by y
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getX() != r2.getX()) {
            return r1.getX() - r2.getX();
        } else {
            return r1.getY() - r2.getY();
        }
    }
}
```

Data Structures and Algorithms
http://www.javaguides.net

Using Comparators

- **TreeSet** and **TreeMap** can accept a `Comparator` parameter.

```
Comparator<Rectangle> comp = new
  RectangleAreaComparator();
Set<Rectangle> set = new TreeSet<Rectangle>(comp);
```

- Methods are provided to reverse a `Comparator`'s ordering:

```
Collections.reverseOrder()
Collections.reverseOrder(comparator)
```

Java Data Structures and Algorithms http://www.javaguides.net

Using Comparators (cont.)

- Searching and sorting methods can accept `Comparator`s.

```
Arrays.binarySearch(array, value, comparator)
Arrays.sort(array, comparator)
Collections.binarySearch(list, comparator)
Collections.max(collection, comparator)
Collections.min(collection, comparator)
Collections.sort(list, comparator)
```

Java Data Structures and Algorithms http://www.javaguides.net

Using `compareTo`

- `compareTo` can be used as a test in an `if` statement.

```
String a = "alice";
String b = "bob";
if (a.compareTo(b) < 0) {   // true
    ...
}
```

Java Data Structures and Algorithms http://www.javaguides.net

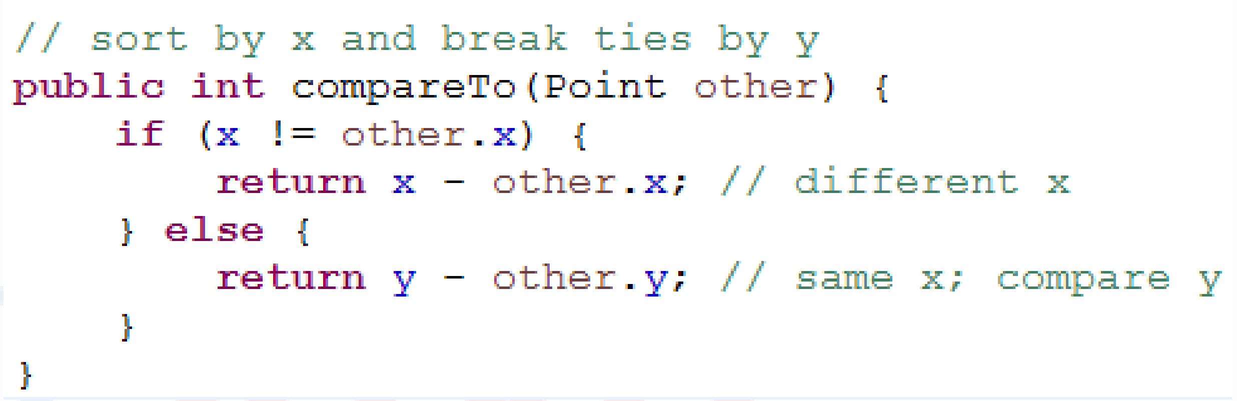
Using `compareTo`

Primitives	Objects
`if (a < b) { ...`	`if (a.compareTo(b) < 0) { ...`
`if (a <= b) { ...`	`if (a.compareTo(b) <= 0) { ...`
`if (a == b) { ...`	`if (a.compareTo(b) == 0) { ...`
`if (a != b) { ...`	`if (a.compareTo(b) != 0) { ...`
`if (a >= b) { ...`	`if (a.compareTo(b) >= 0) { ...`
`if (a > b) { ...`	`if (a.compareTo(b) > 0) { ...`

Java Data Structures and Algorithms http://www.javaguides.net

`compareTo` tricks

- *subtraction trick* – Subtracting related numeric values produces the right result for what you want `compareTo` to return:



```
// sort by x and break ties by y
public int compareTo(Point other) {
    if (x != other.x) {
        return x - other.x;  // different x
    } else {
        return y - other.y;  // same x; compare y
    }
}
```

- The idea:
  - if `x > other.x`, then `x - other.x > 0`
  - if `x < other.x`, then `x - other.x < 0`
  - if `x == other.x`, then `x - other.x == 0`

Data Structures and Algorithms
http://www.javaguides.net

NOTE: This trick doesn't work for `doubles`

compareTo tricks 2

- *delegation trick* – If your object's fields are comparable (such as strings), use their `compareTo` results to help you:

```
// sort by employee name, e.g. "Jim" < "Susan"
public int compareTo(Employee other) {
    return name.compareTo(other.getName());
}
```

- *`toString` trick* – If your object's `toString` representation is related to the ordering, use that to help you:

```
// sort by date, e.g. "09/19" > "04/01"
public int compareTo(Date other) {
    return toString().compareTo(other.toString());
}
```

Java Data Structures and Algorithms http://www.javaguides.net

Section complete

Next Section ►►

Java
Data Structures and Algorithms
http://www.javaguides.net

Count unique words?

- How to write a program that counts the number of unique words in a large text file?

Java
Data Structures and Algorit
http://www.javaguides.net

Count unique words? (cont.)

- How to write a program that counts the number of unique words in a large text file:
  - Store the words in a collection
  - Report the number of unique words
  - Allow the user to search for a word, and report whether or not the word occurred

➔What **<u>collection</u>** would you use?

Java Data Structures and Algorithms http://www.javaguides.net

Set

A List Which Each Element just Occurs One Time

Java Data Structures and Algorithms http://www.javaguides.net

Definition

- A *set*:
  - A collection of elements, without duplicates.
  - Allowed operations: add, remove, search (contains)
  - No indexes – *we cannot get element at index i*
  - Order is unimportant
- Examples:
  - {“PINE”,“APPLE”, “PEN”, “PINE”} => LIST
  - {“PINE”, “APPLE”, “PEN”} => SET
  - {1,2,3,4,5,2,3} => LIST
  - {1,2,3,4,5} => SET

Java Data Structures and Algorithms http://www.javaguides.net

Set methods

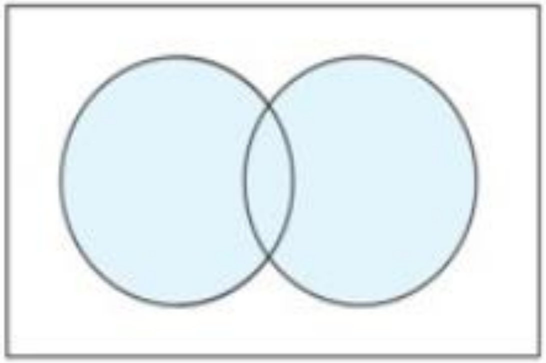
- Fundamental methods:
  - add(e): Adds the element e to S (if not already present).
  - remove(e): Removes the element e from S (if it is present).
  - contains(e): Returns whether e is an element of S.
  - iterator( ): Returns an iterator of the elements of S.

Java Data Structures and Algorithms http://www.javaguides.net

Set operations

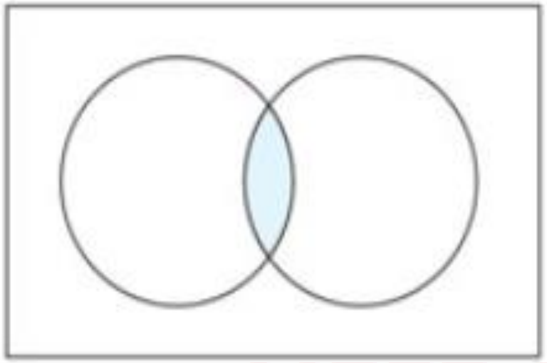
- The traditional mathematical set operations of union, intersection, and subtraction of two sets S and T:

union



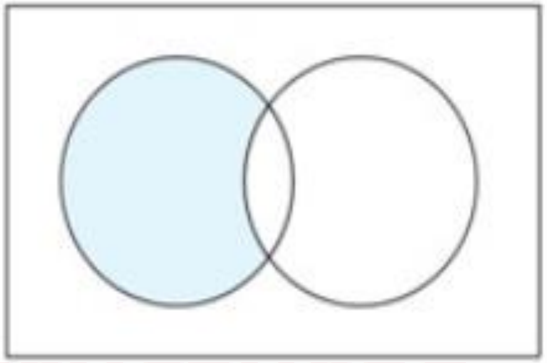
{e: e is in S or e is in T}

intersection



{e: e is in S and e is in T}

subtraction

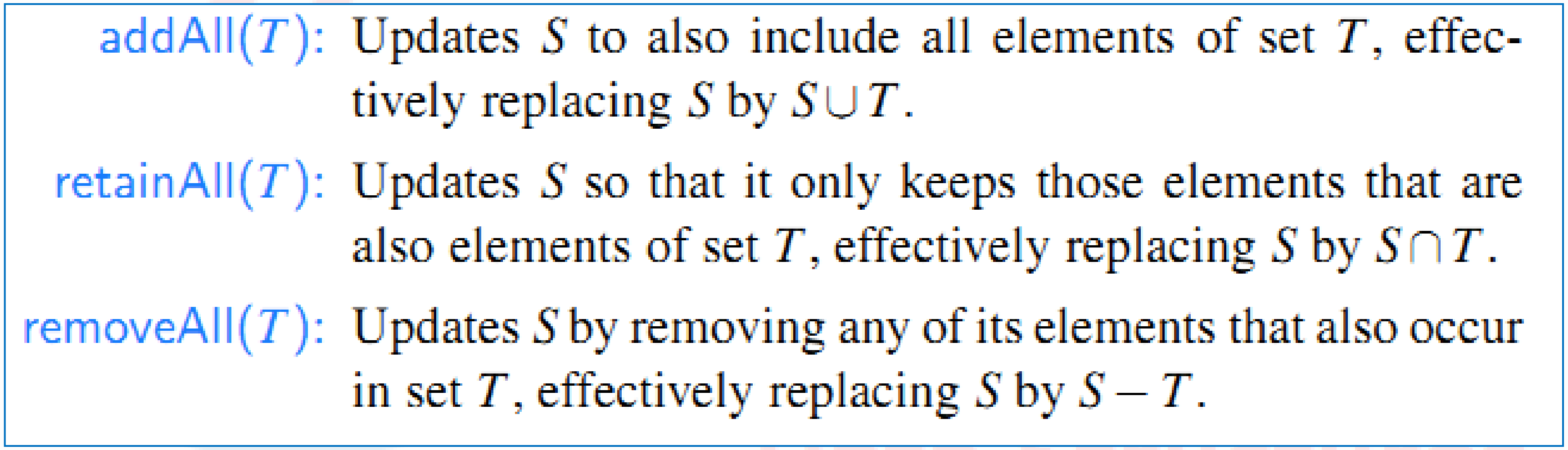


{e: e is in S and e is not in T}

Data Structures and Algorithms
http://www.javaguides.net

Set operations (cont.)

- These operations are provided through the following methods (Set interface):



addAll(T):	Updates S to also include all elements of set T, effectively replacing S by $S \cup T$.
retainAll(T):	Updates S so that it only keeps those elements that are also elements of set T, effectively replacing S by $S \cap T$.
removeAll(T):	Updates S by removing any of its elements that also occur in set T, effectively replacing S by $S - T$.

Data Structures and Algorithms
http://www.javaguides.net

Example

- Testing if s2 is a *subset* of s1
  s1.containsAll(s2)

- Setting s1 to the *union* of s1 and s2
  s1.addAll(s2)

- Setting s1 to the *intersection* of s1 and s2
  s1.retainAll(s2)

- Setting s1 to the *set difference* of s1 and s2
  s1.removeAll(s2)

Java Data Structures and Algorithms http://www.javaguides.net

Iterators for sets

- A set has a method Iterator iterator( ) to create an iterator over the set
- The iterator has the usual methods:
  - boolean hasNext()
  - Object next()
  - void remove()
- remove() allows you to remove elements as you iterate over the set
- If you change the set in any other way during iteration, the iterator will throw a ConcurrentModificationException

Java Data Structures and Algorithms http://www.javaguides.net

Set implementation

- In Java, Sets implement the Set interface in java.util package:
  - HashSet implemented using a "hash table" array very fast elements are stored in unpredictable order
  - TreeSet implemented using a "binary search tree“ pretty fast elements are stored in sorted order
  - LinkedHashSet implemented as a hash table with a linked list running through it, it provides *insertion-ordered* iteration (least recently inserted to most recently) and runs nearly as fast as HashSet.

Java Data Structures and Algorithms
http://www.javaguides.net

What is hashing?

- Technique for finding elements without making a linear search through all elements
- Hash function is a function that computes an integer value (called a hash code) from an object (different objects should have different hash codes)
- Object class has a hashCode method
- int h = x.hashCode();

Java Data Structures and Algorithms http://www.javaguides.net

Hash functions

- Should avoid collisions (two or more different objects with the same hash code)
- If you have your own objects you write:
  - public int hashCode( )
  - When adding x.equals(y) => x.hashCode() == y.hashCode (avoid duplicates)
- E.g. for a circle so that circles of different radii are stored separately.
- Forgetting to implement hashCode means the one assoc with Object is used – not a good idea.

Java Data Structures and Algorithms http://www.javaguides.net

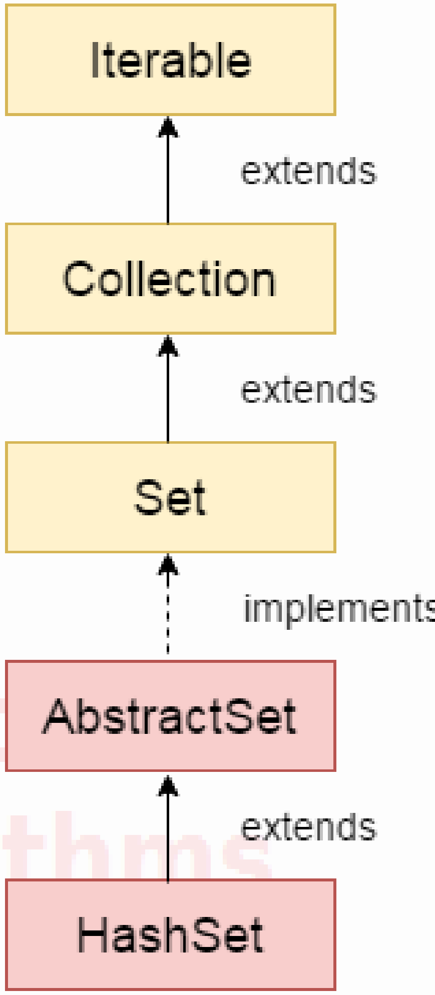
HashSet

Java
Data Structures
and Algorithms
http://www.javaguides.net

HashSet

- HashSet implements the interface Set.
  - Implemented using a hash table.
  - Storing the elements by using a mechanism called **hashing**.
  - No ordering of elements.
  - add, remove, and contains methods constant time complexity O(c).

→HashSet is the best approach for search operations.



Java Data Structures and Algorithms
http://www.javaguides.net

HashSet (cont.)

- **Constructors**:
  - HashSet():
    - construct a default HashSet (capacity: 16, loadFactor: 0.75).
  - HashSet(int capacity):
    - initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
  - HashSet(int capacity, float loadFactor):
    - initialize the capacity of the hash set to the given integer value capacity and the specified load factor.
  - HashSet(Collection<? extends E> c):
    - initialize the hash set by using the elements of the collection c.

Java Data Structures and Algorithms http://www.javaguides.net

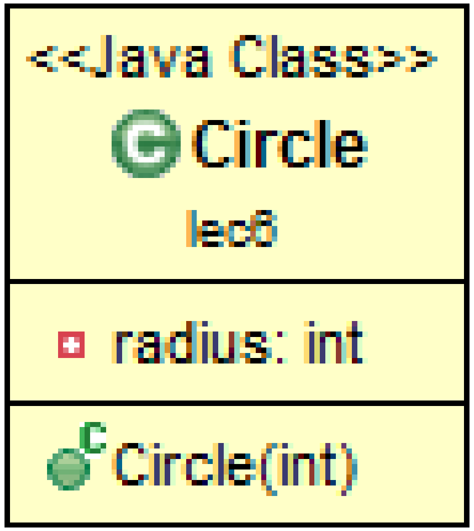
Membership testing in HashSets

- When testing whether a HashSet contains a given object, Java does this:
  - Java computes the hash code for the given object
    - Hash codes are discussed in a separate lecture
    - Java compares the given object, using equals, *only* with elements in the set that have the *same* hash code
- Hence, an object will be considered to be in the set only if *both*:
  - It has the same hash code as an element in the set, *and*
  - The equals comparison returns true
- **Moral:** to use a HashSet properly, you must have a good public boolean equals(Object) and a good public int hashCode() defined for the *elements* of the set

Java Data Structures and Algorithms http://www.javaguides.net

Example

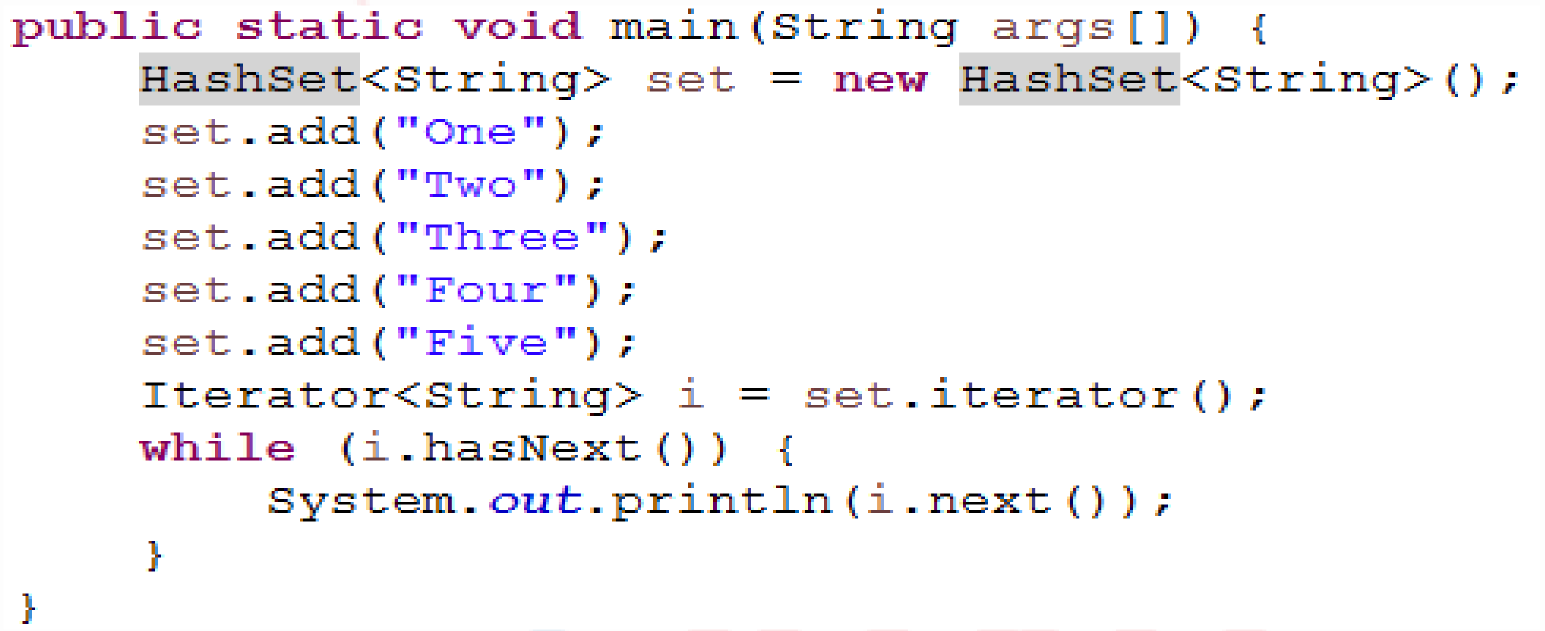
- Consider:



```
Set<Circle> circles = new HashSet<Circle>();
circles.add(new Circle(5));
if (circles.contains(new Circle(5)))
    System.out.println("Circle of radius 5 exists");
else
    System.out.println("Circle of radius 5 does not exists");
```

Data Structures
and Algorithms
http://www.javaguides.net

Using HashSet



```
public static void main(String args[]) {
    HashSet<String> set = new HashSet<String>();
    set.add("One");
    set.add("Two");
    set.add("Three");
    set.add("Four");
    set.add("Five");
    Iterator<String> i = set.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
    }
}
```

- **Output**:

```
Five
One
Four
Two
Three
```

Data Structures and Algorithms
http://www.javaguides.net

Notes on foreach/iterator

- For a given set of circles as follows:

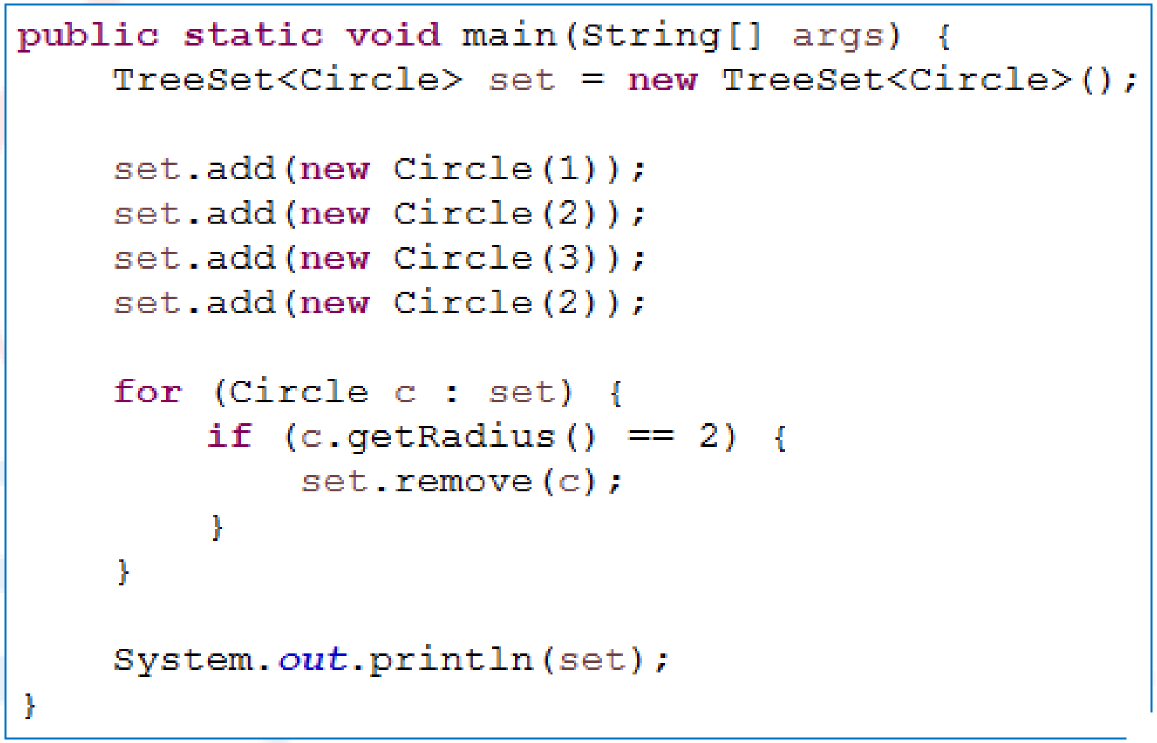
```
TreeSet<Circle> set = new TreeSet<Circle>();

set.add(new Circle(1));
set.add(new Circle(2));
set.add(new Circle(3));
set.add(new Circle(2));
```

- How to remove a specific cricle while iterating a set of circles?

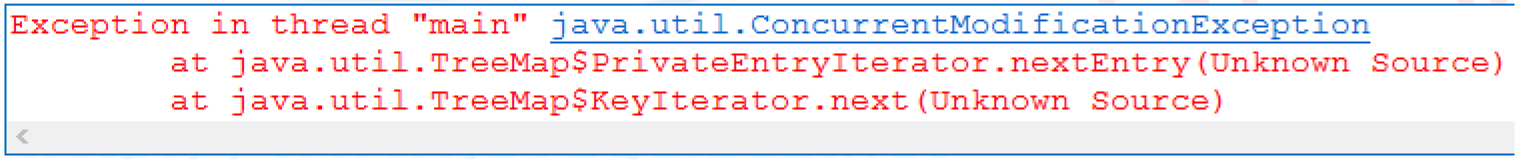
java
Data Structures and Algorit
http://www.javaguides.net

Notes on foreach/iterator

- It’s easy, using foreach approach:



```
public static void main(String[] args) {
    TreeSet<Circle> set = new TreeSet<Circle>();

    set.add(new Circle(1));
    set.add(new Circle(2));
    set.add(new Circle(3));
    set.add(new Circle(2));

    for (Circle c : set) {
        if (c.getRadius() == 2) {
            set.remove(c);
        }
    }

    System.out.println(set);
}
```



```
Exception in thread "main" java.util.ConcurrentModificationException
	at java.util.TreeMap$PrivateEntryIterator.nextEntry(Unknown Source)
	at java.util.TreeMap$KeyIterator.next(Unknown Source)
```

Notes on foreach/iterator

- How about using iterator approach:

```
public static void main(String[] args) {
    TreeSet<Circle> set = new TreeSet<Circle>();

    set.add(new Circle(1));//yes, add ok
    set.add(new Circle(2));//yes, add ok
    set.add(new Circle(3));//yes, add ok
    set.add(new Circle(2));//no, add fail (because of duplication)

    Iterator<Circle> iter = set.iterator();
    while (iter.hasNext()) {
        Circle next = iter.next();
        if (next.getRadius() == 2)
            iter.remove();
    }
    System.out.println(set);
}
```

```
[[Circle: 1], [Circle: 3]]
```

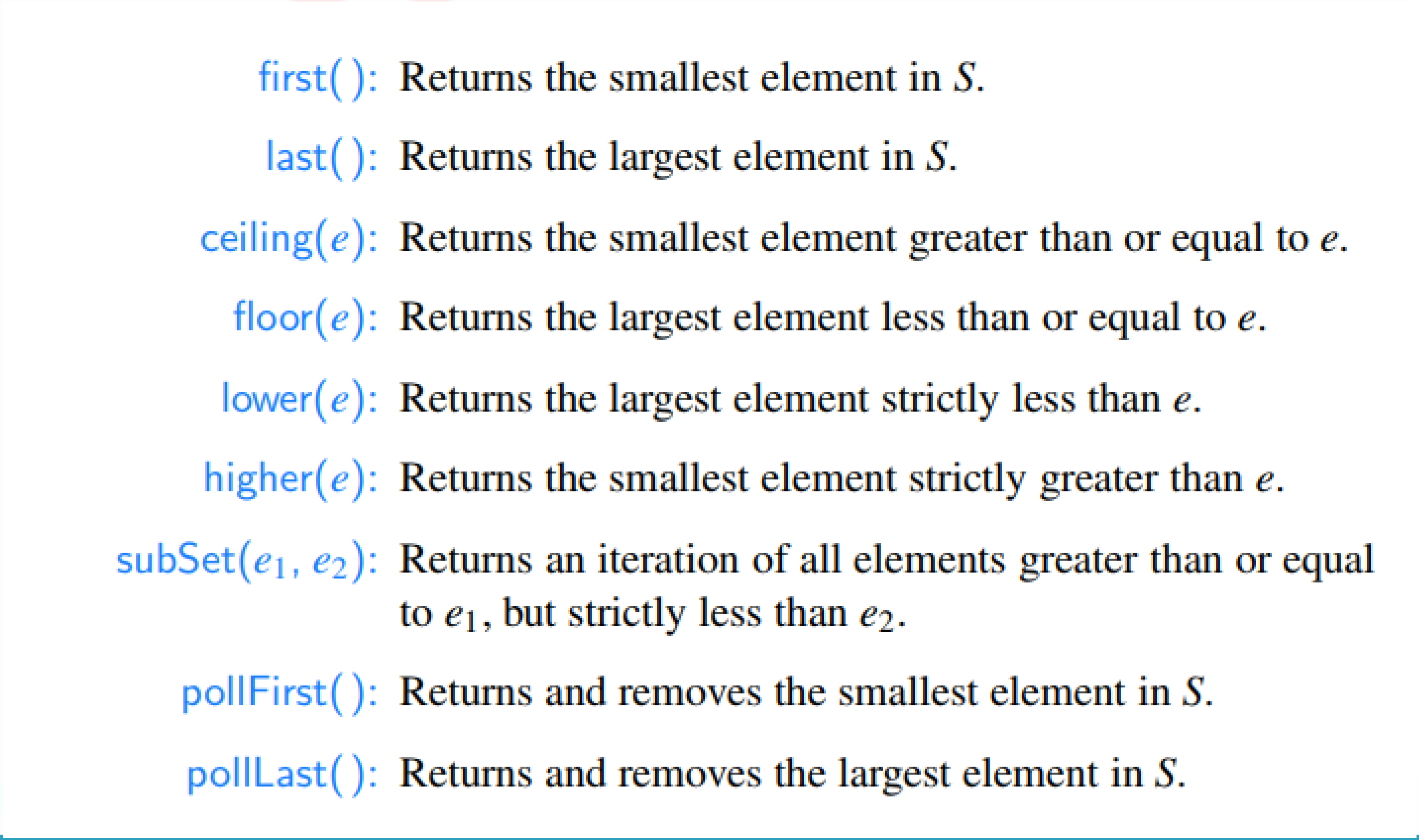
Data Structures and Algorit http://www.javaguides.net

SortedSet

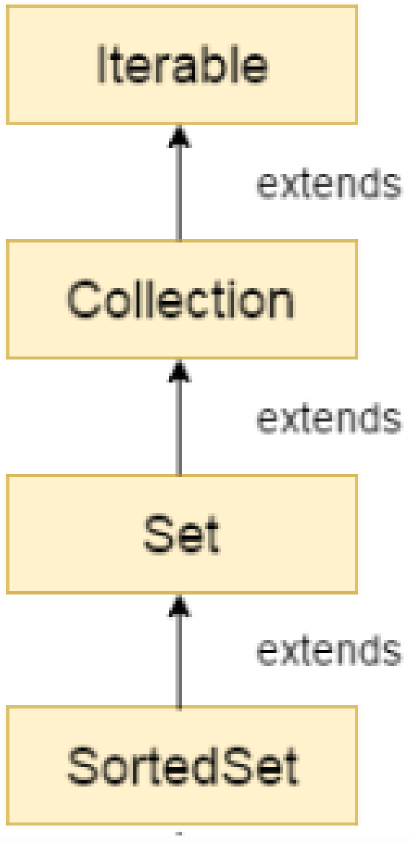
Java
Data Structures and Algorithms
http://www.javaguides.net

SortedSet

- A SortedSet is just like a Set, except that an Iterator will go through it in **ascending order**
- SortedSet is implemented by TreeSet

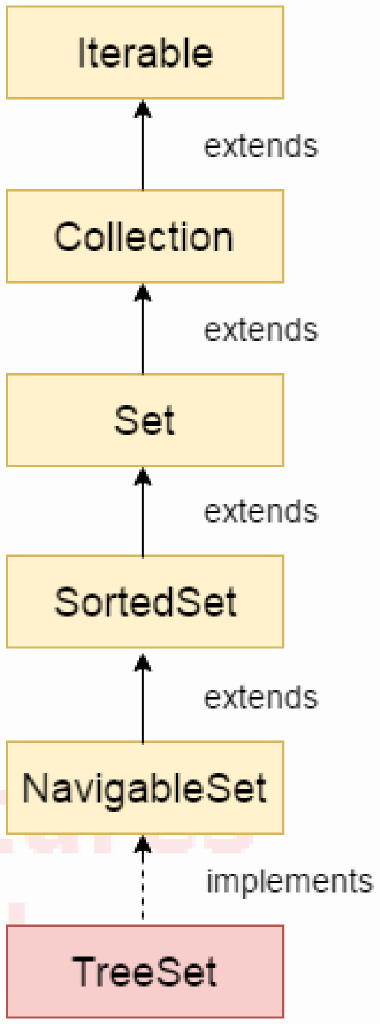


first( ):	Returns the smallest element in S.
last( ):	Returns the largest element in S.
ceiling(e):	Returns the smallest element greater than or equal to e.
floor(e):	Returns the largest element less than or equal to e.
lower(e):	Returns the largest element strictly less than e.
higher(e):	Returns the smallest element strictly greater than e.
subSet(e_1, e_2):	Returns an iteration of all elements greater than or equal to e_1, but strictly less than e_2.
pollFirst( ):	Returns and removes the smallest element in S.
pollLast( ):	Returns and removes the largest element in S.



TreeSet

- TreeSet implements the interface **Set.**
  - Implemented using a tree structure.
  - Guarantees ordering of elements.
  - add, remove, and contains methods logarithmic time complexity O(log (n)), where n is the number of elements in the set.



http://www.javaguides.net

TreeSet (cont.)

- TreeSet():
  - construct an empty tree set that will be sorted in ascending order according to the natural order of the tree set.
- TreeSet(Collection<? extends E> c):
  - build a new tree set that contains the elements of the collection c.
- TreeSet(Comparator<? super E> comparator):
  - construct an empty tree set that will be sorted according to given comparator.
- TreeSet(SortedSet<E> s):
  - build a TreeSet that contains the elements of the given SortedSet.

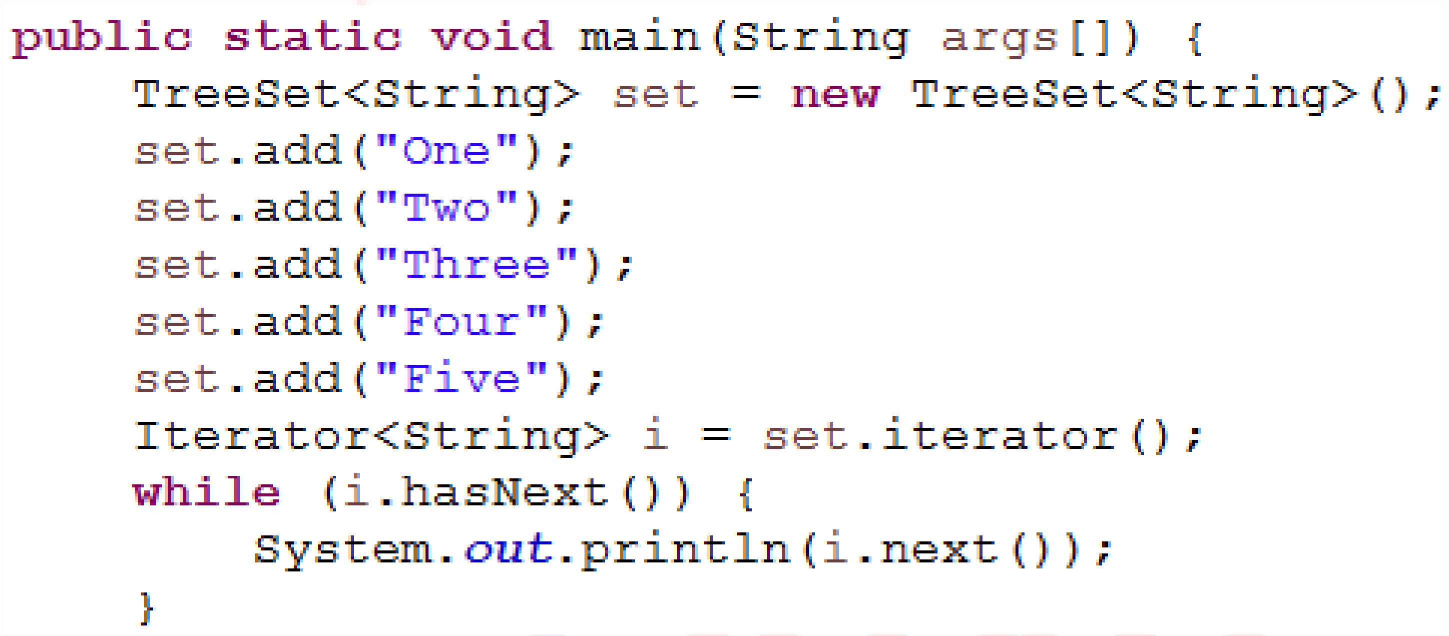
Java Data Structures and Algorithms http://www.javaguides.net

Membership testing in TreeSets

- In a TreeSet, elements are kept in order
- That means Java must have some means of comparing elements to decide which is “larger” and which is “smaller”
- Java does this by using either:
  - The int compareTo(Object) method of the Comparable interface, or
  - The int compare(Object, Object) method of the Comparator interface
- Which method to use is determined *when the TreeSet is constructed*

Java Data Structures and Algorithms http://www.javaguides.net

Using TreeSet



```
public static void main(String args[]) {
    TreeSet<String> set = new TreeSet<String>();
    set.add("One");
    set.add("Two");
    set.add("Three");
    set.add("Four");
    set.add("Five");
    Iterator<String> i = set.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
    }
```

- **Output**:

```
Five
Four
One
Three
Two
```

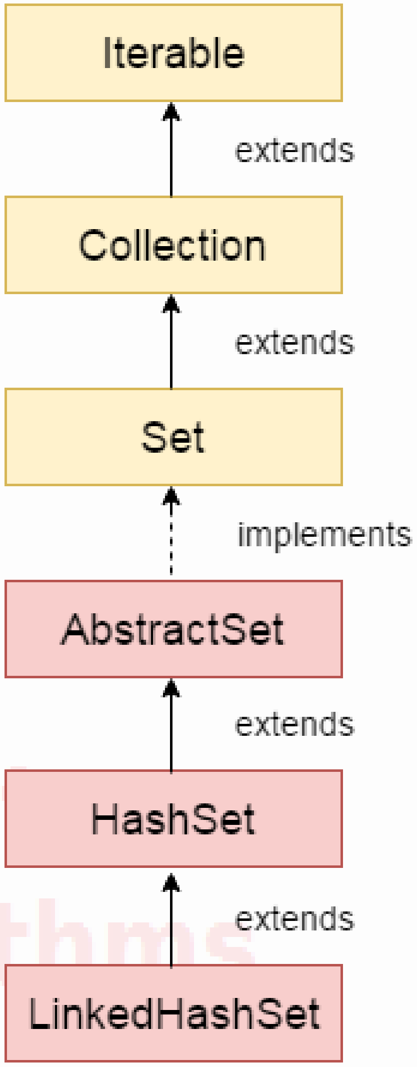
Data Structures and Algorithms
http://www.javaguides.net

LinkedHashSet

Java
Data Structures and Algorithms
http://www.javaguides.net

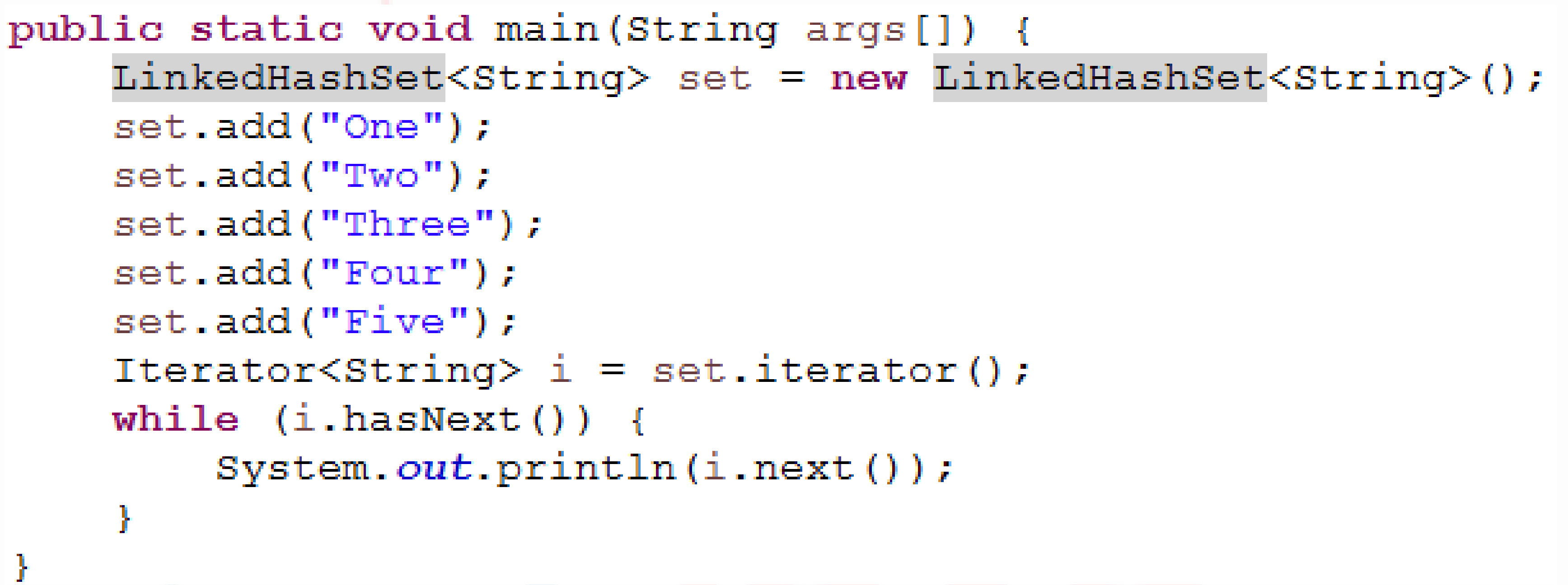
LinkedHashSet

- LinkedHashSet:
  - contains unique elements only like HashSet.
  - provides all optional set operation and permits null elements.
  - is non synchronized.
  - maintains insertion order.



Java Data Structures and Algorithms http://www.javaguides.net

Using LinkedHashSet



```
public static void main(String args[]) {
    LinkedHashSet<String> set = new LinkedHashSet<String>();
    set.add("One");
    set.add("Two");
    set.add("Three");
    set.add("Four");
    set.add("Five");
    Iterator<String> i = set.iterator();
    while (i.hasNext()) {
        System.out.println(i.next());
    }
}
```

- **Output**:

```
One
Two
Three
Four
Five
```

Data Structures and Algorithms
http://www.javaguides.net

Flawed compareTo method

```
public class BankAccount implements Comparable<BankAccount> {
    private String name;
    private double balance;
    private int id;

    // ...
    public int compareTo(BankAccount other) {
        return name.compareTo(other.name); // order by name
    }

    public boolean equals(Object o) {
        if (o != null && getClass() == o.getClass()) {
            BankAccount ba = (BankAccount) o;
            return name.equals(ba.name) && balance == ba.balance && id == ba.id;
        } else {
            return false;
        }
    }
}
```

- What's bad about the above?

Data Structures and Algorithms
http://www.javaguides.net

Flawed compareTo method (cont.)

```
BankAccount ba1 = new BankAccount("Jim", 123, 20.00);
BankAccount ba2 = new BankAccount("Jim", 456, 984.00);


Set<BankAccount> accounts = new
  TreeSet<BankAccount>();
accounts.add(ba1);
accounts.add(ba2);
System.out.println(accounts);      // [Jim($20.00)]
```

- Where did the other account go?
  - Since the two accounts are "equal" by the ordering of `compareTo`, the set thought they were duplicates and didn't store the second.

Data Structures and Algorithms
http://www.javaguides.net

Q
A
Java
Data Structures