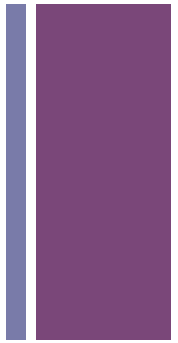
Sorting…



- Why sort?
  - To make searching faster!
    - How?
      - Binary Search gives log(n) performance.
- There are many algorithms for sorting: bubble sort, selection sort, insertion sort, quick sort, heap sort, …
- Why so many?
  - First we will learn some of them and perhaps we will be able to answer this question.
    [Hint: While performance has a lot to do with it, it isn't always about that!]

Using Java Sorting Methods

- Java has built in methods for sorting
  - All arrays can be sorted as long as:
    - the items in the array have a natural ordering
      - e.g. numeric basic data types
      - any object that implements Comparable
  - or
    - there is a Class that can compare the items as if they had a natural ordering.

+ Using Java Sorting Methods

Method sort in Class Arrays
`public static void sort(int[] items)`

Behavior
Sorts the array `items` in ascending order.

Using Java Sorting Methods

Method sort in Class Arrays

```
public static void sort(Object[] items)
```

Behavior

Sorts the array `items` in ascending order.

Using Java Sorting Methods

Method sort in Class Arrays

```
public static <T> void sort(T[] items,
Comparator<? super T> comp)
```

Behavior

Sorts the objects in `items` in ascending order as defined by method `comp.compare`. All objects in `items` must be mutually comparable using method `comp.compare`.

Using Java Sorting Methods

Method sort in Class Collections	Behavior
`public static <T extends Comparable<T>> void sort(List<T> list)`	Sorts the objects in `list` in ascending order using their natural ordering (defined by method `compareTo`). All objects in `list` must implement the `Comparable` interface and must be mutually comparable.
`public static <T> void sort (List<T> list, Comparator<? super T> comp)`	Sorts the objects in `list` in ascending order as defined by method `comp.compare`. All objects must be mutually comparable.

Declaring a Generic Method

SYNTAX **Declaring a Generic Method**

FORM:

methodModifiers <*genericParameters*> *returnType* *methodName*(*methodParameters*)

EXAMPLE:

```
public static <T extends Comparable<T>> int binarySearch(T[] items,
                                                          T target)
```

MEANING:

To declare a generic method, list the *genericParameters* inside the symbol pair <> and between the *methodModifiers* (e.g., `public static`) and the return type. The *genericParameters* can then be used in the specification of the *methodParameters*.

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

`T` represents the generic parameter for the sort method

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

`T` should also appear in the method parameter list

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

The second method parameter means that `comp` must be an object that implements the `Comparator` interface for type `T` or for a superclass of type `T`

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T> void sort(T[] items, Comparator<? super T> comp)
```

For example, you can define a class that implements `Comparator<Number>` and use it to sort an array of `Integer` objects or an array of `Double` objects

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T extends Comparable<T>> void sort(List<T> list)
```

<T extends Comparable<T>> **means that generic parameter** T **must implement the interface** Comparable<T>

Declaring a Generic Method (cont.)

- Sample declarations:

```
public static <T extends Comparable<T>> void sort(List<T> list)
```

The method parameter `list` (the object being sorted) is of type `List<T>`

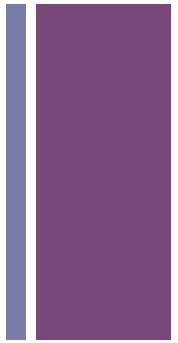
Selection sort

- Basic idea:
  - step forward on each item of the array starting with the first item, if there is a smaller item in front of the item being stepped on, then swap the two items. Repeat until you've stepped on every item.
- Implementation:
  - nested loop
    - first loop marks the current item
      - inner loop finds the smallest item between the current item and the last item inclusively, then swaps the items
- Time Complexity?

Bubble sort

- Basic idea:
  - start with the first item in the array compare adjacent items if they are not sorted, swap them, go to the next item and repeat until you get to the end.
  - repeat the above process until sorted
- Implementation:
  - nested loop
    - first loop checks if the array is sorted
      - inner compares and swaps
- Time Complexity?

Insertion Sort



- Basic idea:
  - start with a sorted subarray, insert the next item from your unsorted list into the right position of the sorted list.
  - When you get to the end of the unsorted list, you are done
- Implementation:
  - nested loop
    - first loop gets next item to insert
      - inner compares, copies and makes space
      - inserts into space
- Time Complexity?