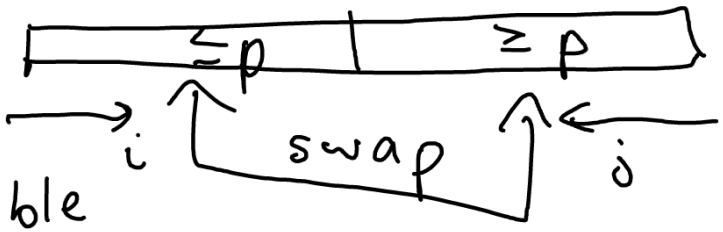
CS 2110 Last time: 2 recursive sorting algorithms

- Mergesort: always $O(n \lg n)$, stable
- Quicksort. partitions array around pivot p.



- Swaps ⇒ not stable
- Running time? Depends on p.

Best case: median ⇒ even split

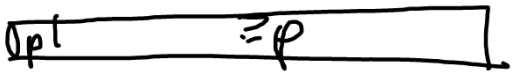
Worst case: smallest / largest element

⇒ $O(n)$ recursive calls

× $O(n)$ time (partition)

= $O(n^2)$ time.

1 vs. n-1 split



sorted array ⇒ $O(n^2)$

Choose random element, swap with a[l]

$O(n \lg n)$ expected time (~2x slower than median)

Still: usu. faster than mergesort

Design patterns

coding idioms
- help organize code
- often: loose coupling
- names, e.g. Iterator

Concept design pattern / Strategy

Generic sort algorithm:

```
<T> void sort (T [] a, Comparator<T> c);
```

c provides the T operations

code does not need to know about T

Generic code on T (GameState, Move)

- interface (e.g. Comparator) describes operations of parameter types (T)
- separate model object (not a T) defines operations

```
interface Comparator<T> {
    /** Returns: 0 if x and y are equal (equivalent)
                 >0 if x is > y
                 <0 if x < y */
    int compare(T x, T y);
}
```

A : A 65
B : B
C : a 97
D : b

- Can override default behavior.
- sort strings while ignoring case.

{"A", "C", "b", "B", "a"} —sort→ {"A", "B", "C", "a", "b"}

—?→ ? {"A", "a", "b", "B", "C"}

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

T extends Comparable<T>
⇒ T knows how to compare to another T

Comparator<T>: object knows how to compare 2 T's.

Lambda expressions ("lambda calculus") — looser coupling ✓

Special syntax for implementing interface w/ 1 method.

```
(x, y) -> x.toLowerCase().compareTo(y.toLowerCase())
```

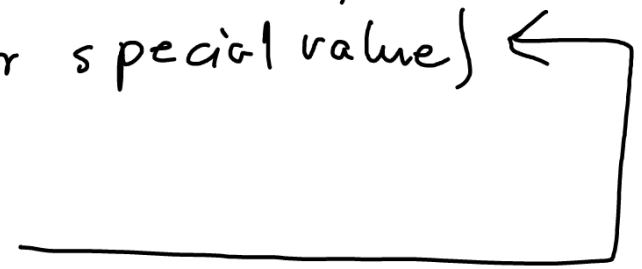
parameters

method body: { ...statements... }
or: returned expression

A4: `advisees = new TreeSet<PhDTree>((x,y) -> x.prof.compareTo(y.prof));`

What if a method has no reasonable value for arguments?

A: Add precondition to rule out arguments
B: Throw error
C: Throw exception (checked)
D: Return null (or special value)
E: It depends
F: Option pattern



Programming without null

Java: mostly type-safe except every value could be null

$x.f = x.f;$ Exception A: Yes → x could be null
B: No
C: ?? ⇒ NullPointerException (NPE)

Alternative: Option pattern.

Java builtin: Optional<T>
- either a T (present) or empty
- if empty Optional used as T throws NoSuchElementExc. unchecked

Our alternative (A5): Maybe<T>
either some(T) or none()

v:T ⇒ Maybe.some(v) : Maybe<T>
Maybe.none() : Maybe<T>

Methods (Optional similar)

boolean isPresent();

T get() throws NoMaybeValue ← fast

returns contained value or

```
Maybe<String> m = Maybe.some("hi");
                  Maybe.none()
try {
println("m = " + m.get());
} catch (NoMaybeValue exc) { println("no"); }
```

⇒ m = hi

no

```
Maybe<T> m = ...

m.then(v -> ...).orElse(...)
```

Optional:map

value if v is present

value if not

```
.orElseGet(() -> ...)
```

lazy