Type Inference

```
interface I1 {}
class A imp I1 {}
class B ext A {}
class C<T> {}.
```

```
public <T ext. A > T run(C <? extends T> c)
```

↳ T <: A

T <: I1.

A <: T

```
I1 i = run(new C<A>());
```

$\boxed{A} <: T <: A$ ✓

<: I1

```
public <T ext. A > T run(C <? super T> c)
```

T <: A

T <: I1

T <: A.

```
I1 i = run(new C<A>());
```

choose B?

No!

$\boxed{T <: A}$

take this one.

no lower bound.

Look at the bound

Then pick the most specific* one!

that is related

Immutability.

problem of aliasing.

shared

```
Point p = new Point (0,0);
Circle c1 = new Circle(p,1);
Circle c2 = new Circle (p,4);

c1.moveTo (1,1);
  ↳ c2 is affected
```

fix with immutability.

↳ keyword – 'final'

  ∟ only be set once.

↳ create a new obj instead of change some state.

↳ take up less memory. ?

ImmutableArray<T>.

```
↳ private final T[] array;

// only items of type T goes into the array.
@SafeVarargs (for type safety)
public static <T> ImmutableArray<T> of (T... items) {
      return new ImmutableArray<> (~~items~~);
                                  (items, 0, items.length - 1);
}.

private ImmutableArray(T[] a) {
      this.array = a.
}
public T get (int index) {
      return this.array [~~index~~];
                        this.start + index
}

@Override
public String toString() {
      return Arrays.toString();
}
```

no set method to pollute array with wrong type.

array of arbitrary size. also its generic!

problem:

subarray?

↳ Immutable array class

  ↳ elems aren't going to change

    ↳ pass around the same Immutable and just change the indexes.

↳ [0 1 2 3 4]
SubArr (1,4)
↳ [1 2 3 4]

```
public ImmutableArray <T> subArray ( int start, int end) {
      return new ImmutableArray <> ( items, ~~start~~, ~~end~~);
}.
```

this.start + start

keep track in class.

this.start + end.

```
private final int start;
private final int end;
private ImmutableArray ( T[] a , int start, int end) {
      this. array = a;
      this. start = start;
      this. end = end;
}.
```

take advantage of immutability

↳ cache / store the object.

↳ share objects around, not worry about them modifing.

↳ Safely share internals of the class.

  ↳ can't modify the array.

↳ Safe concurrent execution.

Nested Classes.

encapsulation · common data & methods.

↳ that belongs / relate to the class.

```
class A {
    private    class B {
    }
    private static class C {
    }.
}
```

this.x. ↓ class B

A.this.x. ↳ class A

class B { ⇒ acts like a field.

inner class ⇒ not static

↳ allow us to access fields from the outer class.

⇒ Static nested class.

↘ Cannot use non-static variables from outer class

interface Comparator <T>.

↳ smth that does the comparison.

↳ Compare (T o1, T o2) method.

↳ returns int.

o1 bef. o2 if -ve.

```
class Sorting {
    static void sortNames (List <String> names) {
        class NameComparator implements Comparator <String> {
            @Override
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
        }
        names.sort ( new NameComparator ( ));
    }.
}
```

y

x

local class

↳ only defined within the method

→ can reference the x within scope of the method.

↘ can ref. y in outer class.

enclosing class & method.

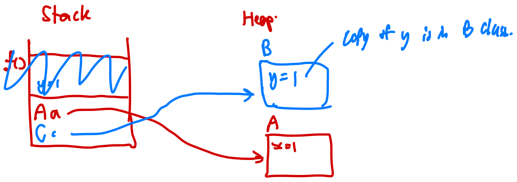
```
interface C { void g(); }
class A {
    int x=1;
    C f() {
        int y=1;
        class B implements C {
            void g() {
                x=y ;
            }
        }
        B b = new B();
        return b;
    }
}.
```

y is in f, but f is removed from stack...

A a = new A();

C c = a.f(); → ref. obj of type B.

c.g();



c.g() → y??

Variable Capture

↳ copy of var. req. is kept. in the local class.

```
class Sorting {

    static void sortNames(List<String> names) {
                        names.sort
        ~~Comparator<String> comp~~ = (new Comparator<String>() {
            @Override
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
        });
        ~~names.sort(comp);~~

    }.
}.
```

anonymous class

↳ no name.

class implementation within curly brackets.

similar to the arrow method thingy

```
class Sorting {

    static void sortNames(List<String> names) {
        boolean ascending = false;
        class NameComparator implements Comparator<String> {
            @Override
            public int compare(String s1, String s2) {
                if (ascending) {
                    return s1.length() - s2.length();
                } else {
                    return s2.length() - s1.length();
                }
            }
        }
        ascending = true;
        names.sort(new NameComparator());

    }.

}
```

← need keep track of this variable

must be final / effectively final

↳ does not change after initialization

← disallowed!

use in lab.

Inner / static nested ⇒ good for OOP.

functional programming

✓ local class → in a method

✓ anonymous → not defined with a name; no class keyword.

Variable Capture

↳ not being able to access things when they are thrown away in the stack, java will capture some.

good practice to make immutable class that