Generics

- new Pair < String , Integer > [2]

  new Pair <S, T>

  new T [2].

  error! - can't mix array and generics.

  arrays are covariant,

  issue caused by type erasure.

- Generics are invariant
  - they are complex type.

    T <: S

    does not imply :

    Array <T> <: Array <S>.

    or >:

    Why?

Circle <: Shape

says nothing

about

Array<Circle>

and Array<Shape>.

What about 2 diff. type?.

< T extends S, S > boolean contains (Array<T> arr , S obj)

↳ works!

arrayShape. copy from (arrayCircle) since Circle <: Shape

\ incompatible type.

Array <Circle> $\not<:$ Array <Shape>.

to compiler, they are no relationship.

Wildcard

upper-bounded wildcard.

Array < ? extends Shape>.

(can be substituted by a subtype of Shape

A < Shape > <: A < ? extends Shape >.

A < Circle > <: ⤴

A < Square > <: ⤴

⋮

Is. Covariance — If S <: T

then A< ? extends S > <: A< ? extends T>.

∴ For any type S , A<S> <: A < ? extends S > <: A < ? extends T >.

lower-bounded wildcard.

Array <? super Shape>.

? can be substituted by a supertype of Shape.
eg. Object

Is Contravariance : if Circle <: Shape
S <: T then

A <? super T> <: A<? super S>.
Shape Circle

match w Shape
⋮
Shape
Circle

Shape and Shape's supertype

For any type S, A<S> <: A<? super S>

Array<Shape> <: Array<? super Circle>.

PECS.

Producer extends; Consumer super.
upper. lower.

eg. src.get(i).
the variable passed in produces a value

eg dest. set(i, this.get(i))
the variable passed in consuming a value

can we use just one type param?

<S> boolean contains (Array <? extends S> array , S obj)

∴ allow us to declare relationships between generic types.

Unbounded wildcard

Array<?>
can be substituted by a type. (any type).

For any type S , A<S> <: A<?>
, A<? super S> <: A<?>
A<? extends S> <: A<?>

super type of all.
generic class A

A <Circle> <: A<?>
A <? extends _> <: A<?>

∴ No need to use raw types any more.

do not use → new A()
use → a instanceof A<?> - trying to a certain runtime type of object
→ new Comparable<?>[10];

Type inference:

Diamond operator <>

Pair< String , Integer > p = new Pair<>();

String, Integer.

not raw type.

infer from here

A.<String>contains(new Array<String>(0), "Hello")

==> using type inference:

diamond operator disappears!

A.contains(new Array<String>(0), "Hello")

Javac infers the type

Type Inference

- Infer type arg. automatically.
- picks the most specific one
- What is it inferring?

  What is the type?

A.contains(circleArray, shape)

Array<Circle>   Shape

<S> boolean contains(Array<? extends S> array, S obj)

① Identify all possibilities

Shape?
GetAreable?
Object?

Supertype of Shape.

'Constraint'
~~Circle?~~
Shape?
GetAreable?
Object?

Supertype of Circle

② Figure out the intersection of the possibilities //

③ Pick most specific one

∴ will infer that the type parameter = Shape.

B <: A <: I1

<T extends A> T m(C<? ~~extends~~ super T> c).

A
B.

I1
A
B

A
I1
Object.

A
B.

A.contains(new String[]{"Hello"}, "Hello")

A.contains(new String[]{"Hello"}, 123).

both would.

<Object>

<T> boolean contains(T[] array, T obj)

String.
S[] <: O[].
Object

Integer.
Number
Object

Shape s = A.findLargest(new Array<Circle>(0));

A<Circle> <: A<? extends Shape>.

<T extends GetAreable> T findLargest(Array<? extends T> array).

bounded!
T <: GetAreable

~~GetAreable~~
~~Country~~
Shape (implements GA)
Circle etc.

GetAreable or any of its subtype

Shape
Circle.
~~Square~~
~~Triangle~~
any subtype of Shape.

Circle
Shape.
any supertype of Circle
~~i.e. Object~~.

∴ most specific is Circle //.