Types

S is a *subtype* of T (`S <: T`) if a piece of code written for T can be used for variables of S

1. **Reflexive** - $S <: S$
2. **Transitive** - if $S <: T$ and $T <: U$, then $S <: U$
3. **Anti-symmetry** - if $S <: T$ and $T <: S$, then $S = T$

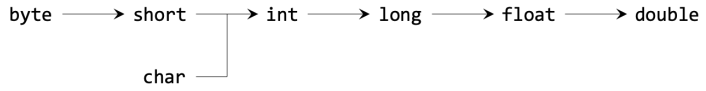
`S instanceof T` returns true if $S <: T$

Type Conversions

Narrowing	Widening
$S <: T$, we can type cast a variable of type T to type S	$S <: T$, we can type cast a variable of type S to type T

`Shape s = new Circle()`
`Circle c = (Circle) s`
*requires **explicit typecasting** and **validation** at runtime
	`Shape s = new Circle()`

Primitive Types



Variance of Types

$C(S)$ stands for complex types - arrays
Covariant - $S <: T$ implies $C(S) <: C(T)$
Contravariant - $S <: T$ implies $C(T) <: C(S)$
Invariant - neither *covariant* nor *contravariant*

* Java **array** is *covariant*
* Java **generics** are *invariant*
* Java arrays and generics **cannot be mixed**

Run-time VS Compile-Time

`Circle c = new ColouredCirlce(...)`
Compile-time type - `Circle`
Run-time type - `ColouredCircle`

OOP Principles

Encapsulation

Hides internal representation and implementation
Exposes just the required method interface for use

Noun	Properties	Associated Verbs
Class	Fields	Methods

Composition - *HAS-A* relationship

Inheritance

`extends` - *IS-A* relationship

Polymorphism

Using same method signatures in different subclasses to determine which method should be executed

Dynamic Binding

1. Determine compile-time type of target
2. Look for all available methods
3. Choose the most specific method
4. Determine run-time type of target

Method Signatures and Descriptors

Method Signature - method name, number of parameters, type of each parameter, order of parameters
`C::foo(B1, B2)`

Method Descriptor - method signature + return type
`A C::foo(B1, B2)`

Method Overriding and Overloading

Overriding - Same method descriptor
Overloading - Same method name, but different method signatures

Information Hiding (Abstraction)

`private` fields, `public` methods

Tell-Don't-Ask

Should get the class to perform a task instead of retrieving the *internal* values and performing the task ourselves

Liskov Substitution Principle

If $S <: T$:

- S should be able to pass all test cases of T
- S should not break any expectations and expected attributes of T - i.e. S should contain all fields and methods of T
- S should be able to replace T and not break any logic

Violation of LSP - if S does not fulfill any of the above conditions

Preventing Inheritance (classes) and Overriding (methods) - `final`

Abstract Classes

`abstract class A {}`
Used when one or more of its instance methods require further details to implement
* may or may not contain any `abstract` methods

Concrete class
A class that is not abstract - overrides any `abstract` methods in its parent class

Interfaces

`interface GetAreable {}`
Models what an entity can do

`class A implements X, Y, Z`
Classes can implement multiple interfaces
If `class A implements X`, $A <: X$

`interface X extends Y, Z`
Interfaces can extend multiple interfaces, but **cannot implement** (interfaces are abstract!)

`interface C {}, class D {}`
`C c = (C) new D()` compiles but subjected to runtime check since there maybe a future class that is a subtype of both `C` and `D` (e.g. `class E extends D implements C`)

Wrapper Class

Primitive	Wrapper
int	Integer
double	Double
char	Character
boolean	Boolean

Enables flexible programs at the cost of performance since primitive wrapper class objects are **immutable**
Leads to **overhead** for memory allocation and garbage collection

Auto-(un)boxing

```
Integer i = 4
// auto-boxing: int 4 converted to an instance of Integer
int j = i
// auto-unboxing: converts instance of Integer back to int
```

Exceptions

Overridden methods in subclasses can throw the same exceptions or any of its subtypes

```
class NewCheckedExceptions extends Exception {
        public NewCheckedExceptions() {
                super("message")
        }
}
class NewUncheckedExceptions extends RunTimeException

class C throws NewCheckedException {
        throw new NewCheckedException("message")
}
...
try {
    // do something
} catch (NewCheckedException e | exception 2) {
    System.out.println(e.getMessage());
    // handle exception 2
} finally {
    // runs regardless of exception occurrence
}
```

Generics

Allow classes/methods to be defined without using `Object` type

- Enforces **type safety** → binds a generic type to specific type at compile time
- errors will be at *compile time* instead of *runtime*

```
class Pair<S, T> {...]

// DictEntry follows the T type in Pair
class DictEntry<T> extends Pair<String, T> {...}

// bounded parameters
class Pair<S extends Comparable<S>, T> {...}
```

```
// arrays and generics dont mix
// need to declare before assigning
class Seq<T> {
  private T[] array;

  public Seq(int size) {
    // The only way we can put an object into array
    // is through the method set() and we only put
    // object of type T inside. So it is safe to
    // cast `Object[]` to `T[]`.
    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
  }
}
```

Generic Methods

```
// type parameter must appear before the return type
public static <T> boolean contains(T[] arr, T obj) {}

// to call a generic method
A.<Circle>contains(...)

// bounded class generics method
public static <T extends GetAreable> T
    findLargest(T[] arr)

// instance method
public <U> void printSomething(U value)
```

Notes

```
B implements Comparable<B> {...}
A extends B {...}
A <: B <: Comparable<B> <: Comparable<?>
Comparable<A> INVARIANT Comparable<B>
Comparable<A> <: Comparable<? extends B>
Comparable<B> <: Comparable<? super A>
```

Type Erasure

During compilation, type parameters are erased and replaced with the most specific reference type
* `Object` for *unbounded* and the **bound** for *bounded* parameters

Heap pollution - situation where a parameterized type expects to hold an object of type X but stores an object of type Y instead

Suppress warnings

- only suppress warnings if it causes a type error
- `@SuppressWarnings` can only be applied to declarations and not assignments

Raw Types

A generic type used without type arguments
Only use together with `instanceof` - `a instanceof A`

Wildcards eliminate this need since we can now do "`a instanceof A<?>`"
Leads to **unchecked warnings**

`Seq<?>` - sequence of specific, but unknown types
`Seq` - sequence of Object instances, no type checking
`Seq<Object>` - sequence of Object instances

Wildcards

PECS - Producer *Extends*, Consumer *Super*

Upper-bounded - `V<? extends A>`

When we want to **retrieve from** instead of adding to

- Covariance
- if `S <: T`, then `A<? extends S> <: A<? extends T>`
- `A<S> <: A<? extends S>`

```
public void copyFrom(Seq<? extends T> src) {
    int len = Math.min(this.array.length,
        src.array.length);
    for (int i = 0; i < len; i++) {
        this.set(i, src.get(i));
    }
}
```

Lower-bounded - `V<? super A>`

When we want to **add to** a collection instead of retrieving from

- Contravariance
- if `S <: T`, then `A<? super T> <: A<? super S>`
- `A<S> <: A<? super S>`

```
public void copyTo(Seq<? super T> dest) {
    int len = Math.min(this.array.length,
        dest.array.length);
    for (int i = 0; i < len; i++) {
        dest.set(i, this.get(i));
    }
}
```

Unbounded - `V<?>`

The parent class of all wildcards
* `Array<?>` is the supertype of all `Array<T>`

Type Inference

Ensures **type safety** - compiler can ensure that `List<myObj>` holds objects of type `myObj` at *compile time* instead of *runtime*

* `<? super Integer>` - inferred as `Object`
* `<? extends Integer>` - inferred as `Integer`

Diamond Operator

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

p would be inferred as an instance of `Pair<String,Integer>`

Constraints

1. Target typing - type of target variable
→ `Shape o = statement()`, so `T <: Shape`
2. Type parameter bound
→ `public static <T extends GetAreable>`, so `T <: GetAreable`
3. Argument parsed
→ `Seq<Circle> <: Seq<? extends T>`, so `T <: Circle`
→ `Shape <: S`, so `Shape <: S <: Object`

```
public static <T extends GetAreable> T
    findLargest(Seq<? extends T> seq) {...}
Shape o = A.findLargest(new Seq<Circle>(0));
public .. contains(..., S obj) {...}
A.contains(circleSeq, shape)
```

Most specific

`Type1 <: T <: Type2`	→ `T = Type1`
`Type1 <: T`	→ `T = Type1`
`T <: Type2`	→ `T = Type2`

Java

Access modifiers

- `public`/`private` - fields accessible/inaccessible from outside the class
- `static` - associate field with class
- `final` - value will not change

`this` - refers to current instance

Stack and Heap

```
public A multiply(A b) {
    A a = new A(this.x * b.x, new B());
    return a;
}
public static void main(String[] args) {
    B b = new B();
    A a1 = new A(2, b);
    A a2 = new A(3, b);
    A c = a2.multiply(a1);
}
```