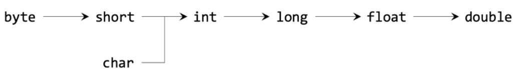
1 Types

Primitive Types

```
byte <: short <: int <: long <: float <: double
```



Widening

```
int i = 5;
double d = i;
```

Subtype can be assigned to super type

Narrowing

```
Circle circle = (Circle)
obj;
```

Supertype can be assigned to subtype through **casting**

Reference Types

- Reference types as **instance or static variables** are initialized to null by default.
- **Local reference** variables are not defaulted and must be explicitly initialized. (variable localVar might not have been initialized)

2 Classes and Abstraction Barrier

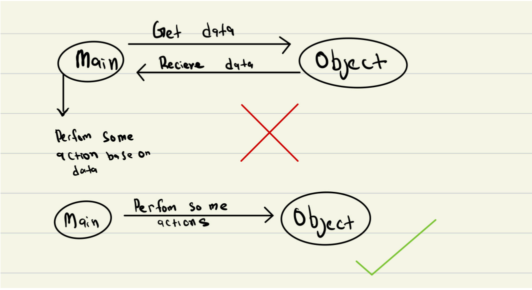
Encapsulation is the bundling of data (fields) and methods (functions that operate on the data) into a single unit, called a class.

- Verbs = functions/methods
- Noun = Object
- Fields = State of the Object

Fields

- Private cannot be accessed from outside the class, and can only be accessed **within** the class
- Public field or method can be accessed, modified, or invoked from outside the class.

Tell don't ask principle



3 Heap and Stack

- Heap for storing dynamically allocated objects;
- Stack for local variables and call frames.
- Metaspace for storing meta information about classes;
- Arrows

Parse the code to identify:

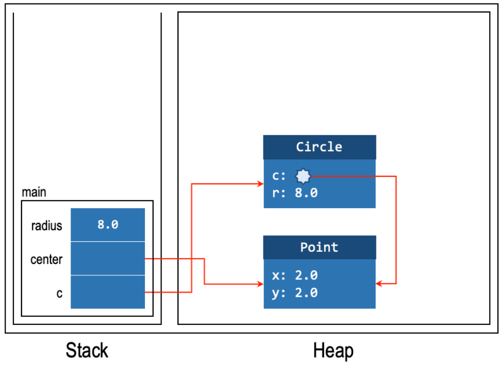
- Local variables: Variables declared in methods or as parameters.
- Instance variables: Variables declared in the class but not marked static.
- Static variables: Variables declared with the static keyword.

For each variable or object:

- If declared inside a method or as a parameter, it's stored on the stack.
- If declared using new, allocate the object in the heap and store its reference in the stack.
- If declared as static, allocate it in the method area.

For each method call:

- Allocate a stack frame.
- Store local variables and method parameters in this stack frame.
- Push the frame onto the stack.
- When the method returns, pop the frame from the stack.



4 Inheritance

The constructor of a subclass automatically includes an implicit call to the no-argument constructor of its superclass if a specific super() call is not explicitly made.

Use composition to model a **has-a relationship**; inheritance for a **is-a relationship**. Make sure inheritance preserves the meaning of subtyping.

5 Polymorphism

Feature	Method Signature	Method Descriptor
Method Name	✅ Included	✅ Included
Number of Parameters	✅ Included	✅ Included
Types of Parameters	✅ Included	✅ Included
Order of Parameters	✅ Included	✅ Included
Return Type	❌ Not Included	✅ Included
Class Name (Optional)	✅ May be included	✅ May be included

Method Signature	`C::foo(B1, B2)`
Method Descriptor	`A C::foo(B1, B2)`

- **Overriding:** Same method descriptor. The method can be a subtype of the overriding class
- **Overloading:** Same name but a differing method signature

Equals Method Make sure it's a equivalence relationship

6 Dynamic Binding

A method is considered more specific than another if it can handle a narrower range of arguments compared to the other method. The closer to CTT(C) as possible.

Complie time-step

- Determine the compile-time type of obj (i.e., CTT(obj)).
- Determine the compile-time type of arg (i.e., CTT(arg)).
- Determine all the methods with the name foo that are accessible in CTT(obj):
  - This includes the parent of CTT(obj), grandparent of CTT(obj), and so on.
  - The access modifiers are appropriate.
- Determine all the methods from Step 3 that are compatible with CTT(arg):
  - Correct number of parameters.
  - Correct parameter types (i.e., supertype of CTT(arg)).
- Determine the most specific method from Step 4:
  - If there is no most specific method, fail with a compilation error.
  - Otherwise, record the method descriptor.

Run Time Step

- Retrieve the method descriptor obtained from the compile-time step.
- Determine the runtime type of obj (i.e., RTT(obj)).
- Starting from RTT(obj), find the first method that matches the method descriptor as retrieved from Step 1 **Exact Match**:
  - If not found, check in the parent of RTT(obj).
  - If not found, check in the grandparent of RTT(obj).
  - ...
  - If not found, check in the root Object.
  - If not found, fail with a runtime error.

7 Liskov Substitution Principle (LSP)

A subclass should not break the expectations set by the superclass. If a class B is substitutable for a parent class A then it should be able to pass all test cases of the parent class A. If it does not, then it is not substitutable and the LSP is violated.

8 Abstract and Interface

Abstract

```
abstract class Shape {
  private int numOfAxesOfSymmetry ;

  public boolean isSymmetric() {
    return numOfAxesOfSymmetry > 0;
  }

  abstract public double getArea();
}
```

- Cannot be init
- A class with at least one abstract method must be declared abstract.
- An abstract class may have no abstract method
- Instantiate an anonymous class extending an abstract class.
- An abstract class does not need to have an abstract method but an abstract method need an abstract class
- Abstract class cannot be final

Interface

- A single class can implement two interfaces that have the same method signature, and the class only needs to provide one implementation of the method to satisfy both interfaces.
- By default, all methods in an interface are public and abstract (prior to Java 8)
- Interfaces can have concrete methods using the default keyword. (8+)
- Interfaces can have static methods. (8+)
- Interfaces can define private methods for code reuse within the interface (9+)
- All fields in an interface are public, static, and final by default.
- No Constructors
- No Instance Fields

Functional Interface

- An interface with exactly one abstract method is a functional interface (used in lambda expressions).

```
package cs2030s.fp;

public interface  Transformer<T, U> {
  public U transform(T t);
}
```

9 Run time mismatch

1. Find the compile-time type of variable b (denoted CTT(b)).
2. Check if there is a "possibility" that the run-time type of b (denoted RTT(b)) is a subtype of C (i.e., RTT(b) <: C). We will explain the possibilities more later.
   - If it is impossible, then exit with compilation error.
   - Otherwise, continue to step 3.
3. Find the compile-time type of variable a (denoted CTT(a)).
4. Check if C is a subtype of CTT(a) (i.e., C <: CTT(a)).
   - If it is not, then exit with compilation error.
   - Otherwise, add run-time check for RTT(b) <: C.

Possibilities

- if CTT(B) <: Casting (✅)
- if Casting <: CTT(B) (✅)
  - Run time checked needed
- If casting is an interface (✅)
- if casting </: CTT(B) (❌)
- if casting is an Interface and CTT(B) is a final class (❌)

Run-Time Check

1. Find the run-time type of variable b (denoted RTT(b)).
2. Check if RTT(b) <: C.

10 Variance

- **Covariant**: Allows reading, where subtyping is preserved.
- **Contravariant**: Allows writing, where subtyping is reversed.
- **Invariant**: Enforces exact type matching, no subtyping allowed.

Kind	As Producer	As Consumer
Array	X x = arr[n];	arr[n] = value;
Function	X x = f(arg);	f(value);

11 Exception

- The first catch block that will be caught
- If a supertype is placed above it's respective subtype it will cause an unreachable error
- `throws supertype` it will accept all it's subtype
- **Checked Exception:**
- **Unchecked Exception:**

12 Generics

- Generic array declaration is fine but generic array instantiation is not
- Generics help reduce ClassCastException

Generic Classes

```
class Box<T> { T value; void set(T value) { this.value =
value; } }
```

Generic Interfaces

```
interface Pair<K, V> { K getKey(); V getValue(); }
```

Generic Methods

```
public static <T> void printArray(T[] array) { ... }
```

Bounded Types Parameters

```
<T extends Number> void method(T num) { ... }
```

Wild Cards

- Unbounded (?): Accepts any type.
- Upper-bounded (? extends T): Accepts subtypes of T.
- Lower-bounded (? super T): Accepts supertypes of T

13 Type Erasure

Method-level override

```
interface I {
  <T extends T1> int foo(T t);
}
```

Different number of parameter

```
  class C implements I {
  @Override
  public <T extends T1, S> int foo(T t) {
    return 0;
  }
}
```

FAIL: Not Renaming

```
class C<T extends T1> implements I {
  @Override
  public <S extends T> int foo(S t) {
    return 0;
  }
}
```

FAIL: Method-Level Type parameter

```
class C implements I {
  @Override
  public <S> int foo(T1 t) {
    return 0;
  }
}
```

FAIL: Class-Level Type parameter

```
class C<S extends T1> implements I {
  @Override
  public int foo(S t) {
    return 0;
  }
}
```

Class-Level Overriding

```java
interface I<T> {
  int foo(T t);
}
```

FAIL: Not Using Type Argument

```java
class C implements I<String> {
  @Override
  public int foo(Object t) {
    return 0;
  }
}
```

- Unless the argument was not given since String was given we cannot use Object

Overloading: Make sure the type erased code has a different method signature

14 Warning

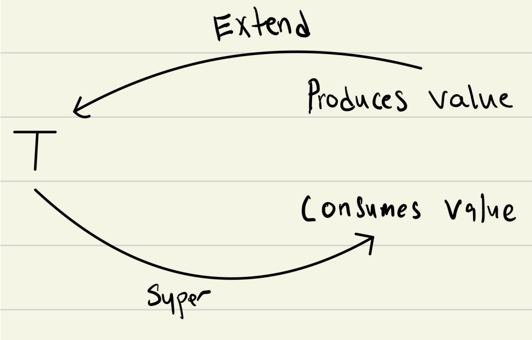
Unchecked Warnings An unchecked warning in Java occurs when the compiler encounters a situation involving generics where it cannot verify type safety at compile time due to type erasure.

When it is allowed Since array is declared as private, the only way someone can put something into the array is through the Seq::set method1. Seq::set only put items of type T into array. So the only type of objects we can get out of array must be of type T. So we, as humans, can see that casting Object[] to T[] is type-safe.

Rawtype assigning a parameterized value to a raw type is considered a raw type usage. **ITS NOT ALLOWED**

15 Wildcards

`List<?> cannot confirm the type when retrieved unless we cast it explicitly`



Type Inference

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

Rules of inference

1. **Argument Typing:** Type of argument is passed to parameter.
2. **Target Typing:** Return type is passed to variable.
3. **Type Parameter:** The declared type, especially for bounded type parameter.

16 Immutability

An immutable class is a class for which there cannot be any visible changes outside of its abstraction barrier.

- Ease of understanding
- Enabling safe sharing of objects
- Enabling safe sharing of internals
- `@SafeVarargs` annotation for passing an array of items (T ...)
- Enabling safe concurrent execution

Checklist

1. Ensure that all fields have the final modifier (not necessary but good to have).
2. Ensure that the types of all the fields are immutable classes.
3. Ensure that arrays are copied before assigning to a field.
4. Ensure that there is no mutator.
   - If there was a mutator and you are modifying the class to be immutable, then you need to return a new instance instead.
5. Ensure that the class has the final modifier to prevent inheritance.

17 Nested Class

- A variable is considered **effectively final** if (**Only to local Variable**):
  1. It is assigned a value only once, and
  2. It is not reassigned (even if the final keyword is not explicitly used).
- If such a variable is captured (used) in a lambda expression, anonymous class, or nested class, its immutability is enforced by the compiler to ensure safety.

Static Nested Class:

- Associated with the containing class.
- Can only access static fields and static methods of the containing class.

Non-static Nested Class (Inner Class):

- Associated with the instance.
- Can access all fields and methods of the containing class.

Local Classes:

- Declared within functions or methods.
- Declared in a block of code between { }.
- Has access to variables of the enclosing class through the this reference.
- Has access to local variables of the enclosing method if they are effectively final.

Annoymous class

```java
new ClassName (ConstructorArguments) { body of a normal class}
```

18 Lambda

```java
Maybe::of      // x -> Maybe.of(x)         (i) static method (Maybe is a class)
x::compareTo   // y -> x.compareTo(y)    (ii) instance method (x is a variable)
Some::new     // x -> new Some(x)     (iii) constructor    (Some is a class)

// Other possibilites
A::h // 2 parameters (x, y) -> x.h(y) or (x, y) -> A.h(x, y)
// 3 parameters (x, y, z) -> x.h(y, z) or (x, y, z) -> A.h(x, y, z)
```

Curried Function

```java
Function<Integer, Function<Integer, Integer>> curriedAdd = a -> b -> a + b;

// Use the curried function
Function<Integer, Integer> add2 = curriedAdd.apply(2); // Partially apply the first argument
System.out.println(add2.apply(3)); // Output: 5

// Or in one line
System.out.println(curriedAdd.apply(2).apply(3)); // Output: 5
```

19 Pure functions

- A pure function will always return the same result given the same inputs.
- The function does not perform actions like:
  - Writing to or reading from files.
  - Modifying global variables or external state.
  - Printing to the console
- Pure functions often work with immutable data structures, avoiding changes to input data.

20 Streams

- **Stateless Operations:** Operate on each element independently (e.g., map, filter).
- **Stateful Operations:** Require knowledge of the entire stream or parts of it to produce results (e.g., distinct, sorted).

Terminal Operations:

- Operations that trigger the evaluation of the stream.
- Examples:
  - Stream::forEach
- Typical workflow: Chain a series of intermediate operations, ending with a terminal operation.
- Reduce (fold/accumulate): Applies a lambda repeatedly on elements to reduce into a single value.
- Element Matching: Returns booleans.
  - Examples: noneMatch, allMatch, anyMatch.

Intermediate Operations:

- Operations that return another stream.
- Examples:
  - Stream::map, Stream::filter
- Types:
  - Stateful: Needs to keep track of some states to operate (e.g., sorted: Returns a stream with elements sorted).
  - Bounded: Should only be called on a finite stream.
    - Examples: sorted, distinct are bounded operations.
  - Truncation: Converts infinite stream to finite stream.
  - Peeking: Takes in a consumer, applying a lambda on a "fork" of the stream.

Consumed Once:

- Streams can only be operated on once. IllegalStateException will be thrown otherwise

21 Monad

- Left Identity Law
  - `Monad.of(x).flatMap(x -> f(x))` must be the same as `f(x)`
- Right Identity Law
  - `monad.flatMap(x -> Monad.of(x))` must be the same as `monad`
- Associative Law
  - `monad.flatMap(x -> f(x)).flatMap(x -> g(x))` must be the same as `monad.flatMap(x -> f(x).flatMap(y -> g(y)))`

22 Functor

Two Laws:

- preserving identity: `functor.map(x -> x)` is the same as functor
- preserving composition: `functor.map(x -> f(x)).map(x -> g(x))` is the same as `functor.map(x -> g(f(x)))`.

23 Parallel Streams

The stream cannot be parallelized if:

- **Interference:** Happens when one of the stream operations modifies the source of the stream (e.g., the original list) during the execution of the stream's pipeline.
- **Stateful:** Parallelizing stateful lambdas can cause inconsistent results because multiple threads might access and modify the shared state simultaneously. **Stateless:** Stateless operations can run in parallel safely because they do not depend on shared mutable state.
- **Side-effect:** ArrayList is what we call a non-thread-safe data structure. If two threads manipulate it at the same time, an incorrect result may result. we can use a thread-safe data structure. Java provides several in java.util.concurrent package, including CopyOnWriteArrayList.

Associativity:

- `combiner.apply(identity, i)` must be equal to `i`.
- The combiner and the accumulator must be associative – the order of applying must not matter.
- The combiner and the accumulator must be compatible – `combiner.apply(u, accumulator.apply(identity, t))` must equal to `accumulator.apply(u, t)`

To make something parallel it relies on satisfying four key conditions during stream operations: identity, purity, associativity, and compatibility.

The parallel version of findFirst, limit, and skip can be expensive on an ordered stream,

24 CompatableFuture

`.join` needs to be called to perform computation

Creation:

- completedFuture: Represents an already completed task.
- runAsync: Accepts a Runnable lambda expression.

Creation:

- **`completedFuture`**: Represents an already completed task.
- **`runAsync`**: Accepts a `Runnable` lambda expression and completes when the lambda finishes execution.
- **`supplyAsync`**: Accepts a `Supplier<T>` that produces a result and completes when the supplier generates the result.

Chaining Operations:

- **`thenApply`**: Transforms the result of the current stage (map operation).
- **`thenCompose`**: Chains dependent tasks, returning a new `CompletableFuture` (flatMap operation).
- **`thenCombine`**: Combines the results of two independent tasks.
- **`thenRun`**: Executes a `Runnable` after the current stage completes.
- **`runAfterBoth`**: Executes after the current stage and another `CompletableFuture` both complete.
- **`runAfterEither`**: Executes after either the current stage or another `CompletableFuture` completes.

Getting Results:

- **`get()`**: Blocks until the `CompletableFuture` completes. Throws:
  - `InterruptedException`: If the thread is interrupted.
  - `ExecutionException`: If the computation throws an exception.
- **`join()`**: Similar to `get()`, but throws unchecked exceptions.

Exception Handling:

- **`handle`**: Handles exceptions and returns a fallback value or computed result.
  - Example: `.handle((result, ex) -> ex == null ? result : "Fallback")`
- **`exceptionally`**: Handles exceptions by returning a fallback value.
- **`whenComplete`**: Executes a callback after completion, providing access to the result and exception.

25 Final

- **Final Class:** A final class cannot be subclassed (extended). This is useful when you want to prevent inheritance for security, immutability, or design reasons.
- **Final method:** A final method cannot be overridden in subclasses. This ensures that the method's behavior remains unchanged in any subclass.
- **Final variable:** A final variable can be assigned only once. Once assigned, its value cannot be changed.

26 Fork and Join

Thread Pool:

- Consists of:
  - Collection of threads, each waiting for a task to execute.
  - Collection of tasks to be executed.
- Tasks are put in a shared queue, and an idle thread picks up a task from the shared queue to execute.

Fork and Join:

- ForkJoinPool: Fine-tuned for the fork-join model of recursive parallel execution.
  - Parallel divide-and-conquer model of computation.
  - Solves a problem by breaking it up into smaller (but identical) problems and then combining the results.
- RecursiveTask supports methods fork() and join(), as well as compute():
  - left.fork(): Adds tasks to the thread pool (one of the threads will call its compute() method).
  - right.compute(): Normal method call.
  - left.join(): Blocks until the computation of the recursive sum is completed and returned.
- ForkJoinPool:
  - Each thread has a deque of tasks.
    - A deque is a double-ended queue, and behaves like both stack and queue.
  - When a thread is idle, it checks its deque:
    - If deque is empty, it picks up a task at the head of the deque.
    - If deque is still empty, it picks up a task from the tail of the deque of another thread to run (work stealing).
  - When fork() is called:
    - The caller adds itself to the head of the deque of the executing thread.
    - Most recently forked task gets executed next (similar to how normal recursive calls work).
  - When join() is called:
    - If the subtask to be joined hasn't been executed, compute() is called and the subtask is executed.
    - If the subtask to be joined has been completed (by this thread or another thread), the result is read and join() returns.
    - If the subtask to be joined has been stolen and is being executed, the current thread finds some other tasks to work on either in its local deque or steals another task from another deque.

Order of fork() and join():

- Most recently forked task is likely to be executed next.
  - join() the most recent fork() task first.
- Order of forking should be the reverse of the order of joining:
  - left.fork();
  - right.fork();
  - return right.join() + left.join();
- Should only be at most a single compute (in the middle of the palindrome).