0 Compiled, Typed Languages

java $\xrightarrow{\text{compiled}}$ bytecode $\xrightarrow[\text{interpreted}]{\text{compiled}}$ machine code

Why Compile? Able to debug while program is still being developed

Why not? Source code is analysed without running, cannot always ensure correctness ⇒ Conservative / Permissive Checking

eg. CTT(c) = Shape, RTT(c) = Circle

```
Shape c = new Circle();
c.getRadius();
```

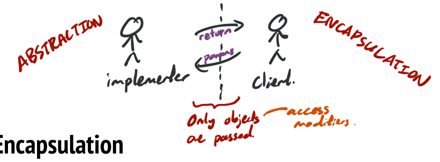
Type Conversions

Suppose S <: T. Then implicit type conversion from S to T (narrowing) is allowed. Type conversion from T to S (widening) requires type casting.

1 Abstraction

The *separation of concerns* between implementor and client.

- Compartmentalise computation & effects
- Hide implementation of tasks ⇒ able to change
- Reduce code complexity through code reuse



2 Encapsulation

Abstraction over types/functions, exposing *ONLY* a certain interface

- Ability to create complex data types
- Creates an abstraction barrier

Composition

Models a *HAS-A* relationship. Further abstraction and encapsulation.

Dangers of Aliasing: Sharing references between objects

Information Hiding

To enforce the abstraction barrier using access modifiers, to avoid direct access to info and data

	Same class	Same Package Sub	Same Package Non-Sub	Different Package Sub	Different Package Non-Sub
private	✓	✗	✗	✗	✗
default	✓	✓	✓	✗	✗
protected	✓	✓	✓	✓	✗
public	✓	✓	✓	✓	✓

Tell, Don't Ask

Telling the object what to do, instead of Asking for state + Performing task on behalf.

- Prevent leakage of class implementation details
- Keep encapsulation intact
- Reduce coupling between client and class

Tasks performed only on fields should be implemented in the class

3 Inheritance

`Subclass <: Superclass`

Models the *IS-A* relationship. To extend certain properties of classes. Technically breaks the abstraction barrier

`final` Keyword in declarations to prevent class inheritance, method overriding and field re-assignment

Method Overriding & Overloading

Method Signature: Number, type, order of params., and name

Method Descriptor: Method Signature + Return type

`@Override`: When subclass defines an *instance* method with the same method signature, a return sub-type, and throws sub-type checked exception, or doesn't throw an exception

Overloading: When class defines two or more methods with the same name but different method signatures.

Liskov Substitution Principle

Let $\phi(x)$ be a property provable $\forall x$ of type T. Then $\phi(y)$ should be true $\forall y$ of type S where S <: T, ie.

⇒ Subclass is substitutable, and pass all test cases of superclass

⇒ Ensures inheritance with method overriding is safe

Needs to be enforced by programmer, not compiler

Inheriting

Classes can extend ≤ 1 superclasses, implement ≥ 0 interfaces. Interfaces can extend ≥ 0 interfaces.

- Abstract Class
  - Allow general & extensible code writing for other classes
  - To demonstrate *IS-A* relationships, without specific implementation
  - May have no abstract method
  - Cannot be instantiated
- Interface
  - Modelling behaviour across class hierachies
  - Methods are `public abstract`
  - Fields are `public static final`

Note: *ANY* object can be casted to *ANY* interface

4 Polymorphism

Difference in code behaviour without changing existing code. Allows succinct, future-proof code.

Dynamic Binding

Determining which method to be invoked from RTT of object.

`x.invokedMethod(params...)`

1. During compile time
   (a) Determine method descriptor of invoked method with `CTT(x)`
   (b) Search *ALL* methods in `CTT(x)` that can be invoked
   (c) Choose the *most specific* method. If none exists, error.
   (d) Store method descriptor in the bytecode
2. During run-time
   (a) Retrieve method descriptor from bytecode
   (b) Search for first matching method:
   ⇒ Starting from `RTT(x)`, iterating to parent class

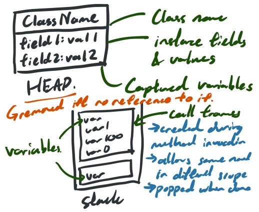
Most Specific Method: If arguments to M can be passed into N without compilation error, M is more specific than N

Class Methods: No Dynamic Binding. Step 1 taken and CTT's method is executed.

5 Heap and Stack

The Heap stores dynamically allocated objects. Persists across method invocations. Dealloc $\iff$!∃ reference

The Stack (actual stack) stores variables, and call frames. ∅ for uninitialised variables. Dealloc when method returns



Primitive Types

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

Note:

- Uninitialised fields have default values, but not variables
- Methods call by value for primitives, call by reference for objects

Wrapper Classes

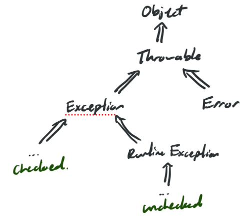
Encapsulates the primitive types to write general code for all reference types. Immutable ⇒ Much more expensive than primitives.

```
Integer i = Integer.valueOf(4);
int j = i.intValue();
```

Type conversion between primitive and wrapper done by auto-boxing and unboxing ONLY with the equivalent wrappers.

6 Exceptions

```
try { // do something }
catch (Exception e) { // handle exception }
finally { // clean up
   // no matter exception occurred or not
}
```



Checked exceptions might still happen with perfect code, programmer has no control. Must be handled to compile.

Unchecked exceptions are caused by programmer errors that cause Runtime Exceptions.

Catch

```
} catch (ExceptionX e) {
   ...
} catch (ExceptionY e) {
```

Exceptions are checked in order. If `ExceptionY <: ExceptionX`, the second catch will never run ⇒ Exception!

Design principles

1. Catch exceptions to clean up, even if just "passing the buck"
2. Do NOT catch all (Exception e)
3. Do not overreact (Exiting the program)
4. Do not break abstraction (Leaking abstraction throwing `FileNotFound` etc.)
5. Do not use as flow control mechanism

7 Complex Types

Variance of Types

Let $C(S)$ be some complex type based on S.

C is Covariant $\iff (S <: T \to C(S) <: C(T))$

C is Contravariant $\iff (S <: T \to C(T) <: C(S))$

C is Invariant $\iff C$ is not covariant $\wedge$ C is not contravariant

Arrays

Covariant for reference types ONLY.

Technically violates the LSP. Unable to put `Object` in `Integer[]`, even though `Integer[] <: Object[]`

8 Generics

A generic type takes other types as type parameters.

Generic Class Pair<S,T> with Type Parameters `S, T` ← SCOPED

Type Arguments `String, Integer` are passed in, to create Parametrised Type `Pair<String, Integer>` Generic Methods are methods that are parametrised with type parameters

Type Erasure

Implementation of Generics in Java: Code Sharing. After type checking, type parameters and arguments are erased.

Alternative: Code Specialisation–Code for every instantiated type.

We denote $|T|$ for the erasure of type T.

- The erasure of a parameterized type `G<T1,\ldots,Tn>` is `|G|`.
- The erasure of a nested type `T.C` is `|T|.C`.
- The erasure of an array type `T[]` is `|T|[]`.
- The erasure of a type variable is the erasure of its leftmost bound.
- The erasure of every other type is the type itself.

Bridging Methods are generated when the class extends a parameterised type, to match the type erased signature of the parent class's method.

```
public void foo(Object arg) {
   foo((Param-edType) arg);
} // Bridging Method
public void foo(Param-edType arg) { ... }
```

Reifiable Types

Where type information is fully available at runtime–none erased during compilation. A type is *reifiable* $\iff$ any of the following:

- It refers to a non-generic class or interface type declaration.
- It is a parameterized type in which all type arguments are unbounded wildcards.
- It is a raw type: Generic Class or Interface without type arguments. DO NOT USE. No type checking at all.
- It is a primitive type.
- It is an array type whose element type is reifiable.
- It is a nested type, for each type T separated by a `.`, T is reifiable.

Arrays' Reifiability allows covariance: When a variable of the wrong type is put in the array, it throws a `RuntimeException`.

Type reifiability prevents **Heap Pollution**–where a variable of a parameterised type refers to an object not of that parameterised type.

Resolved by: Invariance of Generics, and Checking type safety during compile time

To create Generic Type Arrays: Ensure no heap pollution ⇒ `@SuppressWarnings("unchecked")` on array *declaration*

Bounding Type Parameters

Constraining valid type arguments, to allow assert certain properties or method usages: `T extends U, R super S`

Wildcards To write general code despite generic invariance: Enforce subtyping relationships on type arguments.

Producer Extends, Consumer Extends

Whether the variable "*produces*"–returns variables of type T–or "*consumes*"–accepts variables of type T

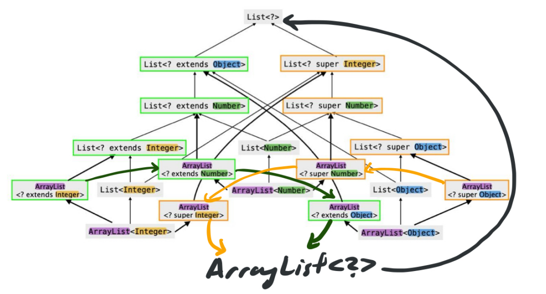
Unbounded Wildcards `T<?>` is the maximum element of the partial order <: . BUT `CTT(T<?>.produce())=Object`, `T<?>.consume(null)` ONLY

Type Inference

When type witness is unspecified: Calling a generic method, or using the diamond operator.

1. A bound set is generated from a list of type parameter declarations $P_1, \ldots, P_p$ and associated inference variables $\alpha_1, \ldots, \alpha_p$ to be determined. For each $l \in \{1..p\}$:
   - If P_l has no bound, $\alpha <: Object$ is added.
   - Otherwise, for each T delimited by $\&$ in the bound, $\alpha_1 <: T$ is added.
   - If no proper upper bound for α_l is added, $\alpha_l <: Object$ is added.

2. Reduction: Simplifying a set of constraint formulas into a bound set. Constraint formulas are given by:
   - Type of return variable: Target Typing
   - Type of Method Arguments
3. Incorporation: To infer new bounds based on assertions of original bounds.
4. Resolution: For a bound set that does not contain the bound $false$, a subset of the inference variables may be resolved.
   - If α_i has one or more proper lower bounds $L_1, \ldots, L_k$, $T_i = \text{lub}(L_1, \ldots, L_k)$
   - Otherwise, if α_i has one or more proper upper bounds $U_1, \ldots, U_k$, $T_i = \text{glb}(U_1, \ldots, U_k)$



9 Functional Programming

Avoiding change altogether

- Ease of reasoning and understanding
- Enable safe sharing of objects: Cached single copy of origin
- Enable safe sharing of internals: Return new object with same cached array with different start/end indices etc.
- Enable safe concurrent execution $\iff$ no mutation within abstraction barrier

9.1 Referential Transparency

Suppose `s.get(0)` returns `k`.

Deterministic and Immutable: We are then able to replace all invocations of `s.get(0)` with `k`, and

No Side Effects: usages of `T t = s.get(0)` with `s.get(0)`.

Immutability

Instances with no visible changes outside abstraction barrier $\Rightarrow$ Every call of the instance's method behaves the same way throughout the lifetime of the instance. How:

1. Prevent Inheritance: IMPT! Ensure no change in behaviour
2. Prevent field reassigning: Ensure no mutation of fields
3. Prevent field mutation: Fields have to be immutable

No Side Effects

- Printing to screen
- Writing to files
- Throwing Exceptions
- Changing other variables
- Modifying arguments

void functions, and division operations are not pure. Overflow is not an error.

10 Nested Classes

Defining a class within another class/method, to group logically relevant classes within the same encapsulation.

NOTE: Container class CAN access private fields of nested class!

```
public/[default] class OuterClass {
   static class StaticNestedClass { ... }
   class InnerClass { ... }
}
```

Access Modifiers

Nested classes can be `private`, `protected`, `[default]`, `public`. Private nested classes' instances can still be returned and exist outside of the class, but their methods and fields cannot be used, even if they are `public`.

Should only be exposed $\iff$ no implementation details leaked, and is an essential part of interface of outer class.

10.1 Static Nested Class

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

```
Outer.Inner iObj = new Outer.Inner();
```

10.2 Inner Class

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

```
Outer oObj = new Outer();
Outer.Inner iObj = oObj.new Inner();
```

Local Class

Class defined within a function, scoped within the method. Can access fields and methods of containing class, and local variables of enclosing method. A local class in a static method can only access static members of the containing class.

Anonymous Class

Declaration AND instantiation of a local class in a single statement.

```
new SuperClass(args) { body }
```

Fields, extra methods, instance initialisers and local classes are allowed in `body`, like a normal class without constructors.

10.3 Variable Capture

Creating a copy of local variables inside the local class. *Language design decision* to only allow variables that are `final` or "effectively final". Mutation not by reassignment is still allowed.

Shadowing

When the inner class declares a variable with the same name as one of the enclosing class, it shadows the variable.

`this.x` refers to the inner class' member, while `Outer.this.x` refers to the outer class' member.

Stack and Heap

On the stack and heap, a new instance of this class will include a copy of the variables required in the inner class, `Outer.this`, and the inner class' variables.

10.4 Lambda Expression

```
@FunctionalInterface
```

for interfaces with exactly **one** abstract method. Able to use lambda expressions as syntactic sugar to simplify *usage of functions as first-class citizens.*

Similar to anonymous classes without introducing a new level of scoping. Declarations, and variable references are *interpreted just as they are* in the enclosing environment. No shadowing issues :)

Not allowed to redeclare variables, even as a parameter.

Curried Functions

$$(X, Y) \to Z \equiv X \to Y \to Z$$

Translating a n-ary function to a sequence of n unary functions–a higher-order function. Each lambda expression is a closure, *storing data from the environment* where it is defined.

Method Referencing

Referencing an existing method rather than specifying a new lambda. Can be used for:

- Static method `A::foo` $\Rightarrow$ `(args) -> A.foo(args)`
- Instance method
  `aObj::foo` $\Rightarrow$ `args -> aObj.foo(args)`
  `A::foo` $\Rightarrow$ `(args[1..])->args[0].foo(args[1..])`
- Constructors `A::new` $\Rightarrow$ `args -> new A(args)`

During compilation, type inference is used to match the given reference to a matching method. Compilation Error if multiple matches or ambiguity.

11 Monads

Equipped with two functions `of` and `flatMap` satisfying the laws:

1. Left Identity Law
   $\texttt{Monad.of}(x)\texttt{.flatMap}(x \to f(x)) \equiv f(x)$
2. Right Identity Law
   $\texttt{m.flatMap}(x \to \texttt{Monad.of}(x)) \equiv \texttt{m}$
3. Associative Law
   $\texttt{m.flatMap}(x \to f(x))\texttt{.flatMap}(y \to g(y)) \equiv$
   $\texttt{m.flatMap}(x \to f(x)\texttt{.flatMap}(y \to g(y)))$

Functor

$$\texttt{Monad} \Rightarrow \texttt{Functor} \quad \texttt{Functor} \not\Rightarrow \texttt{Monad}$$

Sufficient for lambdas to be applied sequentially to a value.

1. Identity Preservation
   $\texttt{f.map}(x \to x) \equiv \texttt{f}$
2. Composition Preservation
   $\texttt{f.map}(x \to f(x))\texttt{.map}(x \to g(x)) \equiv$
   $\texttt{f.map}(x \to g(f(x)))$

12 Concurrency and Parallelelisation

Parallelism: multiple subtasks run at the same time, with a processor that runs multiple instructions, or with multiple processors.

Concurrency: the "illusion" of parallelism.

$$Parallel \Rightarrow Concurrent, Concurrent \not\Rightarrow Parallel$$

Parallelisation is non-deterministic: no order of which processes start and complete.

Parallelisability

- No Interference: No modification of the stream during execution of terminal operation. Throws ConcurrentModificationException
- Stateless: The result does not depend on any state that might change during executiion of the stream Previous elements, Input
- No Side-effects: No concurrent modification of non-thread safe objects that may result in incorrect behaviour

Operations that fulfil these conditions are known as embarrassingly parallel.

$$\text{Parallelising} \neq \text{Performance}$$

Creating too many threads will outweigh benefits of parallelisation.

For **streams**, particularly for

```
stream.reduce(U i,
    BiFunc<U, ? super T, U> acc,
    BinOp<U> comb>)
```

which accumulates each substream and combines the result, parallelisation further requires:

- $\texttt{combiner.apply(i, x)} \equiv \texttt{x}, \forall x$
- `combiner` and `accumulator` are associative.
- `combiner` and `accumulator` are compatible.
  $\texttt{comb.apply(u, acc.apply(i, v))} \equiv$
  $\texttt{acc.apply(u,v)}, \forall u, v$

Streams themselves may define encounter order:

Ordered	Unordered
`Stream::iterate`	
`Stream::of`	`Stream::generate`
Ordered Collections	Unordered Collections

Operations that respect encounter order: `distinct`, `sorted`, `findFirst`, `limit`, `skip` are expensive. `stream.unordered()` if original order is not important.

Threads

In Java, threads divide computation into subtasks, to improve utilisation of processor.

```
new Thread(RUNNABLE).start();
```

The execution flow of each thread can be separately programmed with `Runnable.run()`. Program exits only after *ALL* created threads run to their completion.

CompletableFuture<T>

A MONAD. A CF is either completed or not complete.

To **Create a CF**:

- `CF<U> CF.completedFuture(U value)`
- `CF<Void> runAsync(Runnable r)`
- `CF<T> supplyAsync(Supplier<T> s)`

To **Chain CFs**:

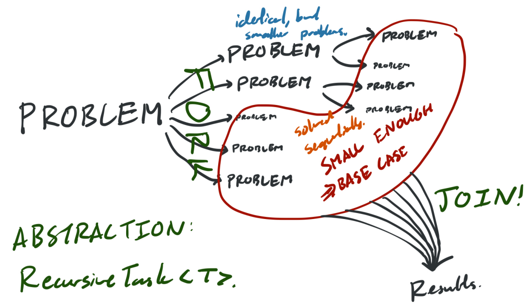
- `CF<Void> thenAccept(Consumer<T> c)`
- `CF<U> thenApply(Function<T, U> map)`
- `CF<U> thenCompose(Function<T, CF<U>> map)`
- `CF<U> thenCombine(CF<V> other, BiFunc<T, V, U> fn)`
- `CF<Void> thenRun(Runnable r)`

To **get CF result**:

- `T get()` throws InterruptedException, ExecutionException
- `T join()` throws any Exception that occurs

Handling Exceptions:

- `CF<T> exceptionally(Func<Throwable, T> f)`
- `whenComplete(BiCons<T, Throwable> cons)`



1. Idle Thread: Take head of own deque. If empty, take tail of another deque, and compute.
2. `fork()` called: Invoker adds itself to the head of current deque.
3. `join()` called: If complete, read result and join returns. If not stolen, compute on current thread. If stolen, do another task while waiting.

Order of fork join: "Palindrome", with only ONE compute in the middle