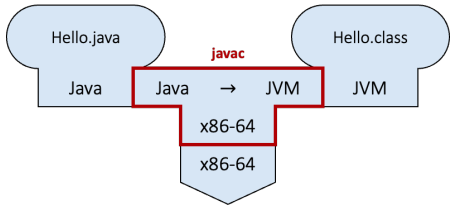
CS2030S Programming Methodology II

AY 24/25 Sem 1 — github/omgeta

1. Java Language

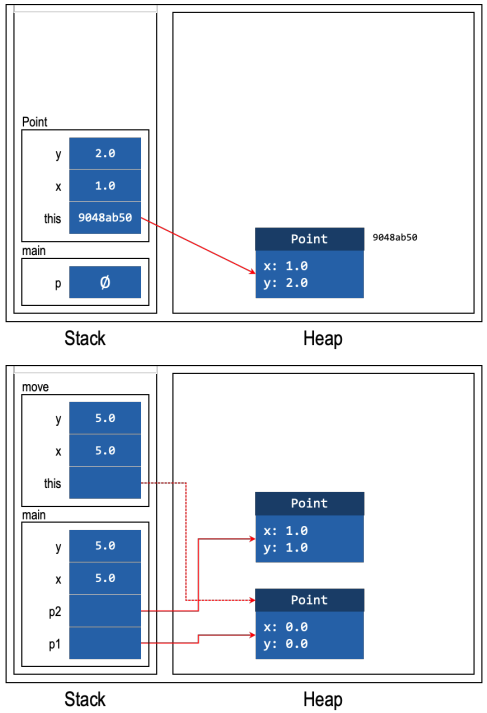
Java is a high-level programming language which compiles into JVM bytecode.



Types in Java have the two general properties:

i. CTT of variables cannot be changed (Static)
ii. Strict compiler type safety checks (Strong)

Stack and Heap Diagram



2. Types

Types determine the meaning and operations defined on variables.

For types `S`, `T` where `S` is a subtype of `T`, `S <: T`:

i. `S <: S` (Reflexive)
ii. $\texttt{S} <: \texttt{T} \land \texttt{T} <: \texttt{S} \rightarrow \texttt{S} = \texttt{T}$ (Anti-symmetric)
iii. $\texttt{S} <: \texttt{T} \land \texttt{T} <: \texttt{U} \rightarrow \texttt{S} <: \texttt{U}$ (Transitive)

For complex type `C(X)` with component types `S`, `T`, there exists a mutually exclusive subtype relationship:

i. $\texttt{S} <: \texttt{T} \rightarrow \texttt{C(S)} <: \texttt{C(T)}$ (Covariant)
ii. $\texttt{S} <: \texttt{T} \rightarrow \texttt{C(T)} <: \texttt{C(S)}$ (Contravariant)
iii. Neither (Invariant)

Type Conversion

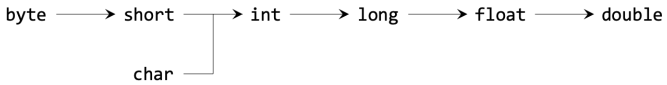
For types `S`, `T` where `S <: T` and the CTT are related, there exist possible conversions:

i. `S`→`T` (E.g. `T t = s`) (Widening - implicit)
ii. `T`→`S` (E.g. `S s = (S) t`) (Narrowing - casting)

Narrowing throws `ClassCastException` if RTT $\not<:$ CTT.

Primitives

Primitives are predefined types for numbers and booleans. They store copies of their values.



Reference Types

Reference types are types for holding references to objects. They can refer to the same underlying object.
Classes are used to define new reference types.
All reference types are a subtype of `Object`.

3. Object Oriented Programming

Encapsulation is the bundling of data with methods that act on it within a class.

Abstraction is the separation of concerns by hiding implementation details from clients.

Inheritance allows a class to inherit fields and methods from a superclass, enabling code reuse and extensibility.

Polymorphism allows clients to use a single interface to represent multiple underlying implementations, enhancing flexibility.

Abstract Classes

Abstract classes cannot be instantiated, but can have fields and methods (abstract or concrete) to be inherited. Abstract classes cannot be final, and abstract methods cannot be private.

```
abstract class Shape {
  private Point c;
  public abstract int distTo(Point p);
}
```

Interfaces

Interfaces cannot be instantiated, but can have abstract methods to be implemented.

```
interface GetAreable {
  int getArea();
}
```

Concrete Classes

Concrete classes can be instantiated if and only if there are no abstract methods after overriding.

```
class Circle extends Shape implements
    GetAreable {
  private int r;
  public int distTo(Point p) { ... };
  public int getArea(){ ... };
}
```

Modifiers

Access modifiers:

i. `public`: accessible from anywhere
ii. `protected`: accessible within the same package and subclasses
iii. `default`: accessible within the same package
iv. `private`: accessible within the defining class

Non-Access Modifiers:

i. `static`: belongs to the class, shared by instances
ii. `final`: cannot be overriden or modified

Method Overloading

Method overloading allows classes to have methods with the same name but a different method signature, i.e., a different set of parameters.

Method Overriding

Method overriding allows subclasses to override inherited methods with the same method descriptor using the `@Override` decorator.

Dynamic Binding

For objects `obj`, `arg` and method call `obj.foo(arg)`, the process of dynamic binding is given by:

1. Find all matching method descriptors in supertypes of CTT(`obj`) that accept parameters of CTT(`arg`).
2. Determine the most specific method descriptor, else throw a compilation error.
3. Starting from RTT(`obj`) and going up to superclasses, search for the method descriptor and run the method.

Static methods do not support dynamic binding, and the method to invoke is resolved entirely using CTT.

Method Specificity

For methods `M`,`N`, `M` is more specific than `N` if its arguments can be passed to `N` without compilation error.

4. Generic Types

Generic types are complex types which operate on types specified at runtime. All generic types are invariant.

For type parameters `S`, `T` we can restrict type parameter:

i. `<S extends T>`
(Upper bounded)

Wildcards

Wildcards allow for generic types to have variance in subtyping relations.

For type parameters `S`,`T` where `S <: T`, we can restrict type parameter:

i. `<? extends S> <: <? extends T>`
(Upper bounded - covariant)
ii. `<? super T> <: <? super S>`
(Lower bounded - contravariant)
iii. `<?>`
(Unbounded - invariant)

Producer Extends, Consumer Super (PECS)

PECS is a guideline for using wildcards:

i. Use `<? extends T>` when methods read values of `T`
ii. Use `<? super T>` when methods set values of `T`

Type Erasure

During compilation, type parameters are arguments are replaced by their bounds (or `Object` if unbounded), limiting runtime access to generic type information.

Type Inference

RTT can be inferred from the CTT using `<>`.

```
public <T extends Circle> T bar(Array<? super T> array)
```

- Type Parameter (Blue)
- Target Typing (Red)
- Argument Typing (Green)

If `S1 <: T <: S2`, Java infers `S1`

5. Exceptions

Checked exceptions are exceptions which the programmer has no control over, and must be declared or caught.
E.g.: `IOException`, `FileNotFoundException`.

Unchecked exceptions are exceptions caused by programmer errors. All unchecked exceptions are subclasses of `RuntimeException`.
E.g.: `IllegalArgumentException`, `NullPointerException`, `ClassCastException`.

Unchecked Warnings

Unchecked warnings are thrown by potential type safety runtime errors due to type erasures.

`@SuppressWarnings("unchecked")` is annotated with a descriptive comment to suppress unchecked warnings for provably type-safe code.

`@SafeVarargs` is annotated with a descriptive comment to suppress unchecked warnings for varargs methods with generic types.

6. Design Principles

Liskov Substitution Principle (LSP)

LSP is the principle that a supertype should be able to be replaced by a subtype without breaking the supertype's properties.

Information Hiding

Information hiding is the principle that access to an object's internal implementation must be restricted, i.e., there must be an abstraction barrier separating the concerns of the client and implementer.

Tell Don't Ask

Tell Don't Ask is the principle that the client should tell an object what to do instead of asking to get the value of a field to perform the computation on the object's behalf.

7. Immutability

Immutable classes have no visible changes outside of its abstraction barrier.

Checklist:

i. All fields are `final` (or effectively final)
ii. All types in fields are immutable
iii. Arrays are copied before assignment
iv. No mutator (or return a new instance)
v. Class is `final`

8. Nested Classes

Nested classes are used to group logically relevant classes together:

i. Inner classes are non-static with access to both contexts.
ii. Static nested classes are static with access to only static contexts.
iii. Local classes are non-static classes defined in methods with access to both contexts.

Non-static classes use qualified `this`, e.g. `A.this.x` to access non-static fields and methods.

Private Nested Classes

Public methods in private nested classes cannot be accessed outside the outer class unless they are implementated methods of an interface.

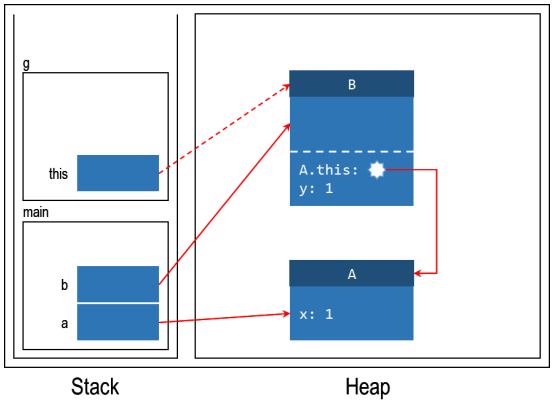
Anonymous Classes

Anonymous classes are unnamed local classes, typically instantiated for single-use cases. Arguments are passed into the constructor.

```
names.sort(new Comparator<String>() {
  public int compare(String s1, String
   s2) {
    return s1.length() - s2.length();
  }
});
```

Variable Capture

Variable capture is the copying of local variables of the enclosing method which are used by the local class. Such captured variables must be `final` (or effectively final).



9. Functional Design

Pure functions have no side-effects, and referentially transparent such that it can be replaced by the return value without affecting correctness.

Functional interfaces are interfaces with exactly one abstract method, which should be annotated with `@FunctionalInterface`.

Lambda Expressions

Lambda expressions are syntactic sugar for instantiating local classes for functional interfaces.

Method Reference

Method references can be used to refer to static methods, instance methods and constructors of a class or instance.

However, ambiguity when using class names to refer to instance methods may cause compilation error:

```
A::new // x -> new A(x)
a::g   // x -> a.g(x)
A::f   // x -> A.f(x) or x.f()
```

Monads and Functors

Monad Laws:

i. `Monad.of(x).flatMap(x -> f(x))` ≡ `f(x)` (Left Identity)
ii. `mon.flatMap(x -> Monad.of(x))` ≡ `mon` (Right Identity)
iii. `mon.flatMap(x -> f(x)).flatMap(x -> g(x))` ≡ `mon.flatMap(x -> f(x).flatMap(y -> g(y)))` (Associative)

Functor Laws:

i. `functor.map(x -> x)` ≡ `functor` (Identity)
ii. `functor.map(x -> f(x)).map(x -> g(x))` ≡ `functor.map(x -> g(f(x))` (Composition)

10. Parallel Programming

Concurrency is the division of computation into independent subtasks which are rapidly switched between.

Parallelism is concurrency with true simultaneous execution. Streams are parallelised with `.parallel()` or `.parallelStream()`. Conditions for parallelisation:

i. Non-interference with data source
ii. Effectively stateless
iii. Avoid side-effects (or collect)

Unordered streams (`generate`, `Set`) are generally more performant for expensive operations which require thread coordination (`.findFirst()`, `.skip()`, `.limit()`) than ordered streams (`iterate`, `List`). Streams can be made unordered with `.unordered()`.

Associativity of `.reduce`

```
.reduce(BinaryOperator<T> acc)
// Optional<T>
.reduce(T id, BinaryOperator<T> acc)
// T
.reduce(T id,
        BiFunction<U, ? super T, U> acc
        BinaryOperator<U> combiner)
// U (parallelizable)
```

Threads

Threads enable parallelization by directly allowing the creation of threads with a lambda.

```
new Thread(() -> {
  System.out.print(Thread.currentThread
    ().getName());
  for (int i = 1; i < 100; i += 1) {
    System.out.print("_");
  }
}).start();
```

CompletableFuture

`CompletableFuture` is a higher-level monad which manages threading and allows programmers to focus on the dependency relations between tasks.

Creation:

i. `completedFuture(T)` (Wrapping completed tasks)
ii. `supplyAsync(Supplier<T>)` (Returns `CompletableFuture<T>`)
iii. `runAsync(Runnable)` (Returns `CompletableFuture<Void>`)

Intermediate Synchronous (same thread as caller):

i. `thenApply` $\iff$ `map`
ii. `thenCompose` $\iff$ `flatMap`
iii. `thenCombine` $\iff$ `combine`
iv. `thenRun(Runnable)` (Returns `CompletableFuture<Void>`)
v. `runAfterBoth(CompletableFuture, Runnable)` (Run after both CFs complete)
vi. `runAfterEither(CompletableFuture, Runnable)` (Run after either CF complete)

*asynchronous versions (e.g. `thenApplyAsync`) may pass the lambda to a different thread.

Terminal Blocking:

i. `get()` (throws checked `InterruptedException`, `ExecutionException`)
ii. `join()` (no checked exceptions)

Fork and Join

Fork and Join is a divide-and-conquer thread pooling strategy which works on instances of abstract class `RecursiveTask<T>` with `.fork()` (submit a smaller version of the task), `.join()` (wait for smaller tasks to complete and return) predefined, and abstract `compute()` implemented by the client.

```
class Summer extends RecursiveTask<
    Integer> {
  private static final int
    FORK_THRESHOLD = 2;
  private int low;
  private int high;
  private int[] array;

  public Summer(int low, int high, int[]
     array) {
    this.low = low;
    this.high = high;
    this.array = array;
  }

  @Override
  protected Integer compute() {
    // stop splitting into subtask if
   array is already small.
    if (high - low < FORK_THRESHOLD) {
      int sum = 0;
      for (int i = low; i < high; i++) {
        sum += array[i];
      }
      return sum;
    }

    int mid = (low + high) / 2;
    Summer left = new Summer(low, mid,
   array);
    Summer right = new Summer(mid, high,
    array);
    left.fork();
    return right.compute() + left.join()
   ;
  }
}
```

Order of `fork()`,`join()`,`compute()`

Order of calls should form a palindrome with no crossing. There is at most a single middle `compute()`.

```
left.fork();  // >---------+
right.fork(); // >------+  | should have
return right.join() // <+  | no crossing
     + left.join(); // <---+
```

ForkJoinPool

Each thread has a deque of tasks. When idle, it picks tasks from its deque's head. If the deque is empty, it steals tasks from another thread's deque tail (work stealing).

Calling `fork()` adds the task to the deque's head, ensuring most recent task is executed next, mimicking recursion.

When `join()` is called:

i. If the task isn't executed, its `compute()` runs.
ii. If completed, the result is returned.
iii. If being executed by another thread, the current thread works on other tasks or steals one.

Calling `compute()` runs the task in the same thread.