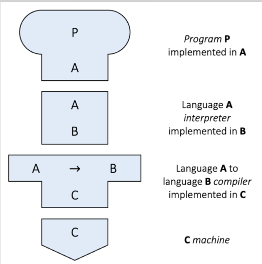
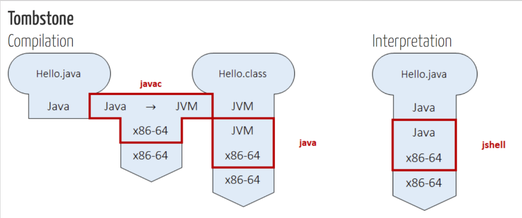
CS2030S
AY24/25 Sem 1
by ngmh

Programming Languages

- Dynamic v/s Static Typing: Dynamic languages have variables that can hold values of multiple different unrelated types. Static languages have variable types that must be declared and cannot be changed
- Strong v/s Weak Typing: Stronger languages have greater rules in their type system to ensure type safety

Tombstone Diagram





Java Subtyping and Types

- T is a subtype of S (T <: S) if code written for S can be safely used for T
- T has a narrower scope, while S has a wider scope
  - T can be put into S (widening)
  - S can be explicitly typecast into T (narrowing)
- Primitive Types
  - byte <: short <: int <: long <: float <: double
  - char <: int
- Reference Types
  - Anything that is not a Primitive Type such as classes
  - Stores only a reference to the value, like a pointer
  - Two reference variables can share the same value; == compares references not actual values
  - Default null value, which could cause errors

Class Fields and Methods

- Use the `static` keyword to specify
- Access using `Class.field` or `Class.method()`
- `this` keyword will NOT work in Class Methods.

Pillars of OOP

- Encapsulation: Bundle of variables (fields) and functions (methods), by making a `class`, that can be represented in a class diagram
- Abstraction: Writing reusable code by grouping sets of instructions
- (Not a Pillar) Composition: Creating wrapper class around an object with additional fields, models a **Has-A** relationship
- Inheritance: Preserve methods and fields of the original object, models a **Is-A** relationship
- Polymorphism: Allowing one variable to take on different run-time types, or different methods to be called

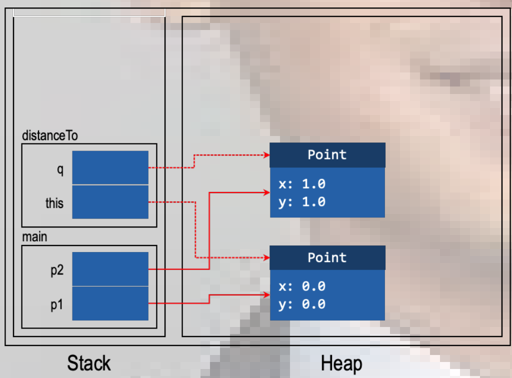
OOP Style

- Information Hiding:
  - Mark fields as `private` where possible. Objects of same class can still access them.
  - Avoid having a **getter** or **setter** when possible
- Tell Don't Ask: Client should tell the class what to do, rather than gathering information to manually calculate

Stack and Heap

- Stack:
  - Where all variables are allocated and stored
  - Contains **Call Frames**, which are created and destroyed when methods are called
- Heap:
  - Where objects are allocated and stored
  - **new** is used: Object created on heap
  - Objects are stored as Class Name, Instance Fields and Values, and Captured Variables
- Example Diagram when `distanceTo()` is called

```
Point p1 = new Point(0, 0);
Point p2 = new Point(1, 1);
p1.distanceTo(p2);
```



Inheritance

- Use the `extends` keyword
- Parent Constructor: `super()`, **ONLY FIRST LINE**
- Overriding
  - Use the @Override annotation
  - Match **Method Descriptor** (Name of method, Type of Parameters, Return Type), can't throw new exceptions
- Overloading
  - Method with same name but different **Method Signature** (Name of method, Type of Parameters)
- Liskov Substitution Principle
  - Any property of objects of type T should be true for any object of type S where S <: T
  - Don't break expectations of the parent class

Dynamic Binding

- Compile Time Type v/s Run Time Type
  - `Circle c = new ColouredCircle()`
  - CTT(c) = Circle, RTT(c) = ColouredCircle
- Example: `obj.foo(arg)`
- Compile Time
  - Check CTT(obj) and CTT(arg)
  - Find all accessible methods named `foo`, including those in supertypes of CTT(obj)
  - Find most specific method (narrowest) that fits with CTT(arg)
  - Record the method descriptor
- Run Time
  - Retrieve method descriptor
  - Determine RTT(obj), recursing upwards to find first method fitting descriptor
- Does not apply to Class methods

Abstract Classes

- A general class, that should **NEVER** be instantiated
- Use the `abstract` keyword
- Abstract arrays can still be defined
- Abstract methods do not have any body
- A class with abstract methods must be made abstract

Interface

- Models requirements of classes
- Use the `implements` keyword for inheritance, methods are public abstract by default
- A class can implement multiple interfaces
- Solves Diamond Problem:
  - If inheritance from multiple classes were allowed, there could be conflicting method declarations
  - Can satisfy both at once with pure interfaces
- Casting to interfaces is allowed, since class could satisfy without explicitly implementing
- Impure interface: Has method body with `default` keyword

Wrapper Classes

- Encapsulate primitive types
- Auto-boxing: Automatically put primitive types into wrapper class (line 1); Auto-unboxing: Automatically put value of wrapper class into primitive variable (line 2)

```
Integer i = 4;
int j = i;
Double d = 2;
```

- However, 2 step boxing is not allowed, (line 3: 2 is not automatically casted to a double before auto-boxing)
- Wrapper classes **DO NOT** share the same subtyping relationship as primitives
- Using wrapper classes come at a performance cost, as objects have to be stored on the heap
- Wrapper classes are immutable, changing value involves auto-boxing and auto-unboxing to make a new object

Type Checking

- `a = (C) b`
- Compile Time
  - Find CTT(b), then check if it is possible for RTT(b) <: C, if not Compile Error
  - Possibility:
    - CTT(b) <: C: This widening cast is allowed
    - C <: CTT(b): This narrowing cast requires runtime checks for RTT(b)
    - C is an interface: There could be a subclass of B implementing C, so a runtime check is needed
  - Impossibility:
    - B and C are unrelated
    - C is an interface and B is final
  - Find CTT(a), then check if C <: CTT(a), if not Compile Error
  - Add run-time check for RTT(b) <: C
- Run Time
  - Find RTT(b), then check if RTT(b) <: C

Variance

- Java Arrays are Covariant: S <: T implies S[] <: T[]
- Contravariant: S <: T implies (T) <: (S)
- Invariant: None of the above, not comparable

Exceptions

- Use `try catch finally` keywords
- `throw` immediately suspends execution of the `try` block
- `catch` can catch all exceptions which are subtypes
- `finally` **ALWAYS RUNS**, unless computer explodes
- Exceptions can be thrown upwards using `throws`
- Unchecked Exceptions: Caused by programming errors, not explicitly caught or thrown, subclasses of RTE
- Checked Exceptions: Beyond programmer's control, must be actively anticipated and handled if not program will not compile

Generics

- Generic Types can take in Type Parameters
- Can be bounded by classes and interfaces using `extends`
- Do not use 2 parameters of the same name together
- Note that generics are **invariant**
- Example Usage

```
class Pair<S, T>{} // Definition
<T> boolean contains(T[] array, T obj){}
A.<String>contains(strArr, str) // Usage
```

Type Erasure

- Java thing for backwards compatibility
- Generic types are replaced by their upper bound, e.g. Comparable (default Object)
- When a Generic is instantiated and used, a typecast is added to the code

```
Int i = new Pair<Str,Int>("a", 4).sec();
Int i = (Int) new Pair("a", 4).sec();
```

Generic Array Problems

- Due to type erasure, we cannot tell if putting Generics into an array will cause errors
- See code below:

```
Pr<Str, Int>[] pArr = new Pr<Str, Int>[2];
Object[] oArr = pArr;
oArr[0] = new Pr<Dbl, Bool>(3.14, true);
```

- becomes

```
Pr pArr = new Pr[2];
Object[] oArr = pArr;
oArr[0] = new Pr(3.14, true);
```

- and this looks okay to us now
- But if we do `Str str = pArr[0].first()`, we get a ClassCastException
- Arrays are reifiable, with full type information available at run-time, unlike generics, so Java does not allow this
- Array declaration is ok, but instantiation with `new` is not

Generic Type Rules

- Generic method signature includes type parameters, with them being equal up to renaming
- Type checking uses type argument for class-level type parameters where possible
- Method descriptor stored during dynamic binding at CTT is type erased

Generic Arrays and Warnings

- Since we cannot instantiate generic arrays, we typecast it instead, since arrays are covariant

```
T[] tmp = (T[]) new Object[sz];
this.arr = tmp;
```

- Generates warning, can view with `-Xlint:unchecked`
- Compiler cannot guarantee absence of RTE, due to possible modification
- Can suppress with `@SuppressWarning("unchecked")`, but need to justify it
- Another warning is for raw types, where compiler cannot guarantee type safety due to usage of raw types

Wildcards

- Solves problem with generics being invariant
- Upper-Bounded: Covariant; S <: T implies <? extends S> <: <? extends T>, <S> <: <? extends S>
- Lower-Bounded: Contravariant; S <: T implies <? super T> <: <? super S>, <S> <: <? super S>
- PECS: Produce Extends, Consumer Super
  - Think from perspective of the wildcard
  - In copyFrom, the wildcard is producing a value
  - In copyTo, the wildcard is consuming a value
- Unbounded Wildcards: "Supertype" of all wildcards
  - Use to replace raw types, does not produce any warnings as no (nonexistent) type information is lost

Type Inference

- Can use an empty diamond operator on RHS to instantiate generic types; useful if generic type is really long
- 3 Step Method:
  - Argument Typing: Type of parameters passed in
  - Target Typing: Type of variable where return is stored
  - Type Parameter: Self explanatory
- Only consider types explicitly involved then choose the most specific type
- Example

```
<T extends Circle> T f(Seq<? super T> s)
GetAreable c = f(new Seq<Circle>());
```

- Type Inference
  - Argument Typing: Circle super T, T <: Circle
  - Target Typing: GetAreable c = T, T <: GetAreable
  - Type Parameter: T extends Circle, T <: Circle
- Resolved as T <: Circle, so T is Circle

Factory Methods

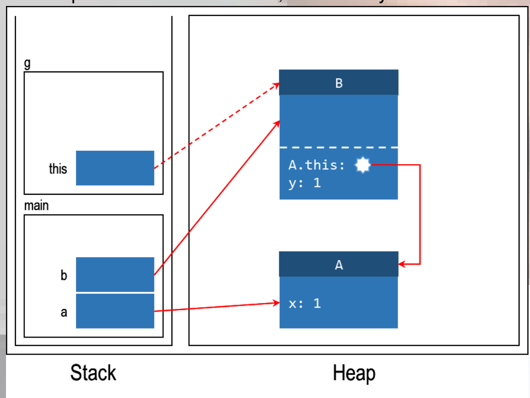
- Create classes through factory rather than constructor
- Static method, hides constructor information
- Useful when consumer does not know exactly what subtype should be created, acts as auxiliary
- Capture generic types to correctly parameterise

Immutability

- Use the `final` keyword to prevent inheritance of classes, or modification of methods, or re-assignment of fields.
- Ensure types of fields are immutable.
- Ensure arrays are copied before assignment.
- Ensure there no mutators, return new instance instead.
- **@SafeVarargs**: Allow different kinds of parameters in constructor such as multiple numbers of number array.

Nested Classes

- Capture super class if not static
- Example: B and C are nested, but C and y are static



- Local Class: Class within a function, access enclosing class fields with this
- Effectively final for primitive types (cannot be reassigned), makes copies of local variables when capturing
- Anonymous Class: Declare and instantiate local class in the same statement

Side Effect-Free Programming and Lambdas

- Side Effect: Printing, Modifying value of field, Mutating arguments, throwing exceptions, etc.
- Referential Transparency: Can replace every $f(x)$ with y in code. Function must be deterministic.
- Pure Function: Side-effect free, Referentially Transparent.
- Inline Functions in Java are implemented as methods in anonymous classes that implements an interface, and will show in the Stack and Heap.
- **@FunctionalInterface** has exactly one abstract method.
- Lambdas using Method References for Static Method, Instance method, and Constructor:

```
Maybe::of      // x -> Maybe.of(x)
x::compareTo   // y -> x.compareTo(y)
Some::new      // x -> new Some(x)
```

- This can be confusing when there are multiple parameters for a `A::h`, as it could be either class or instance method:

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

- When using lambdas, captured variables are effectively final, unlike when using method references.
- Currying: Returning a lambda from a function. Instance methods can be seen as partially applied curried functions.

Maybe and Lazy

- Internalise null checks to prevent bugs.
- Mutate content using transformers, not instance methods.
- Using lambdas, we can define functions for later execution.
- Memoization: Cache the result after evaluation.

Monads and Functors

- Monad: Class with a value and additional side information. Has the `of`, `map` and `flatMap` methods.
- Monadic Laws
  - Left Identity Law: `Monad.of(x).flatMap(x -> f(x)` is the same as `f(x)`.
  - Right Identity Law: `x.flatMap(y -> Monad.of(y))` is the same as `x`.
  - Associative Law: `x.flatMap(y -> f(y)).flatMap(y -> g(y))` is the same as `x.flatMap(y -> f(y).flatMap(z -> g(z)))`.
- Functor: Ensures lambdas can be applied sequentially without worrying about side information.
  - Identity Law: `x.map(y -> y)` is the same as `x`.
  - Composition Law: `x.map(y -> f(y)).map(y -> g(y))` is the same as `x.map(y -> g(f(y))`.

InfiniteList and Streams

- Simple linked list: Store content as head, next node as tail.
- For infinite length, need to evaluate lazily.
- Java Streams are infinite lists with more functionalities.
- Terminal Operation: Triggers evaluation of stream.
- Intermediate Operation: Returns another Stream.
- Bounded Operations: Operations that should only be called on finite streams. They can be stateful, requiring some form of keeping track of states to operate.
- Truncation: Use functions like `limit` or `takeWhile`.
- A Stream can only be operated on once.

Parallelism and Concurrency

- Concurrency: Switching rapidly between multiple tasks to making it seem like they are running at the same time.
- Parallelism: Running subtasks on multiple cores so that they are actually executed at the same time.
- Stream: Use `.parallel()`. Order is not preserved.
- Embarrassingly Parallel: Only involve one element at once.
- Stateful operations or those with side effects might produce the wrong result when parallelised.
- Parallelising `reduce(id, acc, comb)`:
  - `comb.apply(id, i)` must equal `i`.
  - Combiner must be associative. Some non-associative accumulators also work, depending on usage.
  - Accumulator and Combiner must be compatible: `comb.apply(u, acc.apply(id, t))` must equal `acc.apply(u, t)`.

Threads and Asynchronous Programming

- Normally, operations are blocking; code execution only resumes once the operation is finished.
- Threads are a simple flow of execution in a program. We can create them programmatically to run two things at the same time. The `.start()` method returns instantly.
- Using and managing Threads is complicated.
- `CompletableFuture<>` is a monad that allows us to perform tasks concurrently. It has methods like `allOf` or `thenApply` to make composition and mapping easy.
- Multiple `thenApplys`, are evaluated as a stack (LIFO).
- To get the result, we have to use `.get()`, or `.join()` (doesn't throw checked exceptions).
- Java uses thread pools to manage threads through `ForkJoinPool`.
  - Each thread has a deque of tasks.
  - When a thread is idle, it checks its deque of tasks. If not empty, it takes the head task to execute using `.compute()`. Otherwise, it takes the tail task of another deque to run (work stealing).
  - When `.fork()` is called, the caller adds itself to the head of the deque of the executing thread. Subsequent forks also add to the head of the deque, like recursion.
  - When `.join()` is called, if the subtask to be joined hasn't been executed, `.compute()` is called and it is executed. If it has been completed, the result is read and `.join()` returns. If the subtask has been stolen, the current thread finds other tasks to work on.
  - Forks and joins should be called in a palindromic manner. The most recently forked task should be the first to be joined. There should also be at most one compute in the middle. This is because if a task is joined when it is not at the head, we have to search through the deque to find and execute it.

pls give me A+ prof thanks