CS2030S

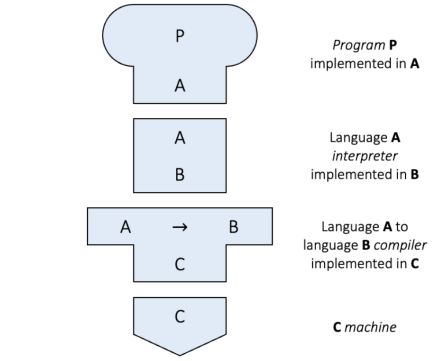
AY24/25 Sem 1

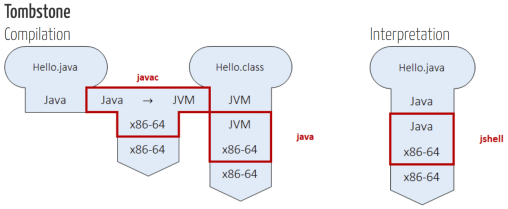
by ngmh

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





Subtyping

- 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)
- Properties
  - Reflexive: T <: T
  - Transitive: T <: S, S <: U, T <: U
  - Anti-Symmetric: If T <: S, S <: T, then S = T

Primitive Types

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

Reference Types

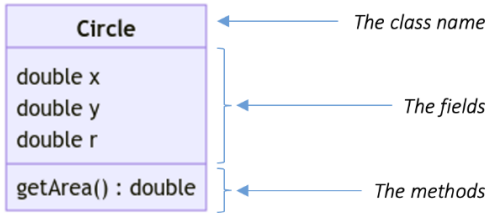
- Anything that is not a Primitive Type such as classes
- Stores only a reference to the value
- Two reference variables can share the same value like pointers, equality by default compares references not actual value
- If uninitialized, have null values and throw errors

Class Fields and Methods

- Use the **static** keyword to specify
- Fields can be accessed using Class.field, common value shared across the entire class
- Methods can also be accessed using Class.method(), but the **this** keyword will **NOT** work

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

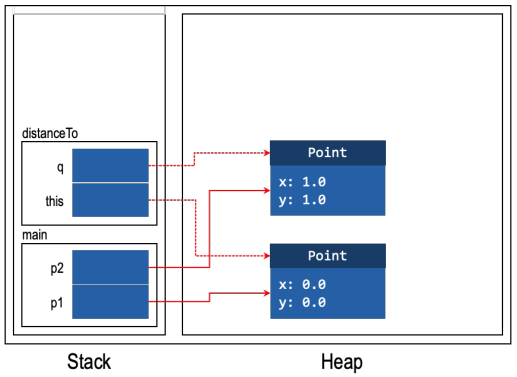
OOP Style

- Information Hiding:
  - Prefer marking fields as **private**, unless necessary, to keep abstraction barrier
  - Private fields can still be accessed by objects of the same class
  - If absolutely necessary, use a **getter** or **setter** to access
- Tell Don't Ask: Client should tell the class what to do, rather than gathering information from the class for actions

Stack and Heap

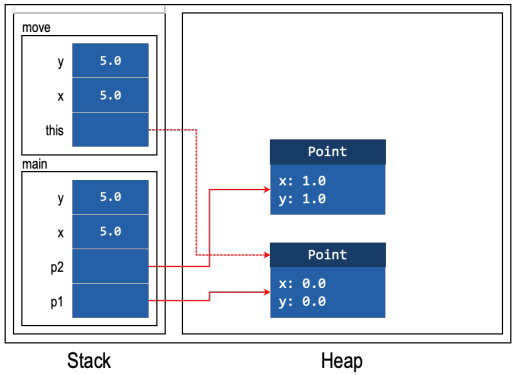
- Stack:
  - Where all variables are allocated and stored
  - Contains **Call Frames**, which are created when methods are invoked and destroyed when methods are returned from
- Heap:
  - Where objects are allocated and stored
  - Whenever **new** is used, a new object is created on the heap
  - Objects are stored as Class Name, Instance Fields and Values, and Captured Variables
- Example Diagram when move is called

```
class Point {
  private double x;
  private double y;
  public Point(double x, double y) {}
  public double distanceTo(Point q) {}
}
Point p1 = new Point(0, 0);
Point p2 = new Point(1, 1);
p1.distanceTo(p2);
```



- Example Diagram when distanceTo is called

```
class Point {
  // ...
  public void move(double x, double y) {}
}
Point p1 = new Point(0, 0);
Point p2 = new Point(1, 1);
double x = 5;
double y = 5;
p1.move(x, y);
```



Inheritance

- Use the **extends** keyword
- Use **super**() to call the parent constructor, **MUST BE FIRST LINE**
- Overriding
  - Use the @Override annotation
  - Must match **Method Descriptor** (Name of method, Type of Parameters, Return Type)
- 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
  - Can use the **final** keyword to prevent inheritance of classes, or modification of methods, or re-assignment of fields

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)
  - Check 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)
  - Start from RTT(obj) and find first method fitting descriptor
  - Recurse upwards until found
- 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
  - Java does not know which method to call
  - Using pure interfaces, they can both be satisfied at once
- Casting to interfaces is allowed, since there is a possibility for the class to satisfy the interface even without implementing it
- Can be impure, when methods have method bodies using the **default** keyword

Wrapper Classes

- Java provides wrapper classes that 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)
  - 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)
  - Check if C <: CTT(a), if not Compile Error
  - Add run-time check for RTT(b) <: C
- Run Time
  - Find RTT(b)
  - 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 run-time errors
- Checked Exceptions: Something programmer cannot control, should be actively anticipated and handled, if not program will not compile

Generics

- Java allows us to define Generic Types that take in Type Parameters
- We can bound type parameters by classes and interfaces using **extends**
- Do not use 2 parameters of the same name in the same context as it confuses the compiler
- 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

- After type checking, Java performs type erasure to ensure 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 us to make them
- Generic array declaration is ok, but instantiation using **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;
```

- However, this generates the unchecked warning, which can be seen in detail if we compile with –Xlint:unchecked
- At compile time, compiler adds typecasts when generics are used
- Unchecked Warning happens when compiler cannot guarantee absence of RTE, due to possible modification of the array
- If we want to suppress it using @SuppressWarning("unchecked"), we need to justify it:
  - array is **private** so it can only be accessed by Seq
  - array can only be modified by Seq::set(**int**, T), which is of type T
  - The only retrievable objects from array are subtypes of T, using Seq::get(**int**)
  - It is safe to cast Object[] to T[]
- Another possible warning is raw types, for the same reason where the compiler cannot guarantee type safety due to usage of raw types

Wildcards

- Solves problem with generics being invariant
- Upper-Bounded
  - copyFrom(Seq<? **extends** T> src) allows src to be a subtype of T
  - Is **Covariant**: S <: T implies <? extends S> <: <? extends T>
  - <S> <: <? extends S>
- Lower-Bounded
  - copyTo(Seq<? **extends** T> dest) allows dest to be a supertype of T
  - Is **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
  - Similar to Object being supertype of all objects, but object as a generic type parameter does not work as generics are invariant
  - Use to replace raw types, does not produce any warnings as no (nonexistent) type information is lost

Type Inference

- We 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
- Java only considers types explicitly involved in the inequalities, then chooses 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: T assigned to GetAreable, 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
- Hides constructor information
- Useful when consumer does not know exactly what subtype should be created, acts as auxiliary
- Static method
- Capture generic types to correctly parameterise

pls give me A+ prof thanks