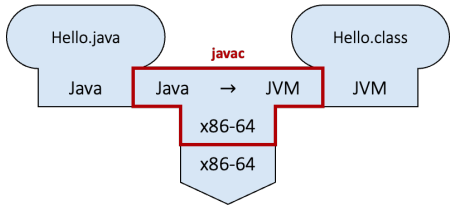
CS2030S Programming Methodology II

AY 24/25 Sem 1 — github/omgeta

1. Java Language

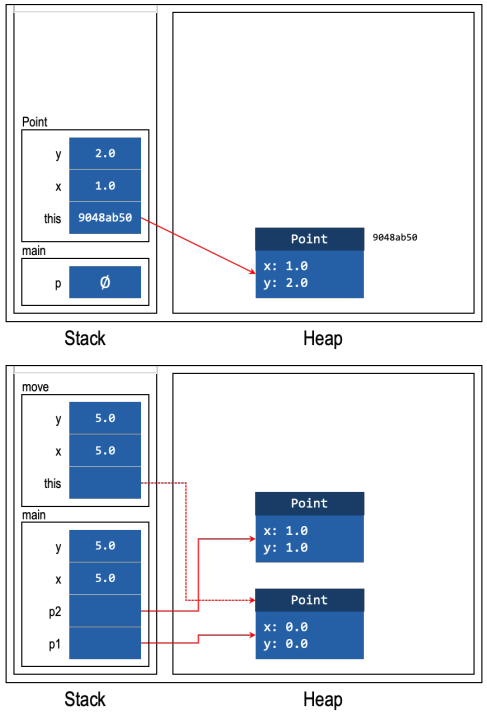
Java is a high-level programming language which compiles into JVM bytecode.



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. `S <: T` ∧ `T <: S` → `S = T` (Anti-symmetric)
iii. `S <: T` ∧ `T <: U` → `S <: U` (Transitive)

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

i. `S <: T` → `C(S) <: C(T)` (Covariant)
ii. `S <: T` → `C(T) <: C(S)` (Contravariant)
iii. Neither (Invariant)

Type Conversion

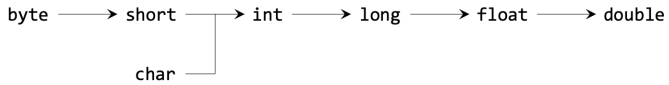
For types `S`, `T` where `S <: T` and the CTT are related, there exist possible conversions:

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 ⊀: 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 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;
  @Override
  public int distTo(Point p) { ... };
  @Override
  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 the type parameter to:

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 the type parameter to:

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`
(Producer Extends)
ii. Use `<? super T>` when methods set values of `T`
(Consumer Super)

Type Erasure

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

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.

If the code is provably type-safe, we can suppress warnings with `@SuppressWarning("unchecked")` annotated with a descriptive comment.

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.