CS2030S
AY25/26 Sem 1

1. Program and Compiler

- Java program is compiled then executed:
  1. **Compile**: `javac Hello.java`
  2. **Execute**: `java Hello`
- Java program interpreted using `jshell` interpreter

2. Variables and Types

- **Statically typed**: variable can only hold values of the same type (assigned at compile time)

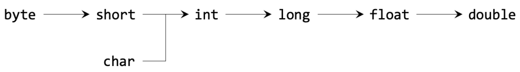
  Compile time type: type the variable is assigned with when declaring the variable

  Type checking during compile time
- **Strongly typed**: enforce strict rules in type system → catch type errors during compile time
- S is **subtype** of T, denoted by $S <: T$, if a piece of code written for variables of type T can also be safely used on variables of type S (S is **supertype** of T)

  (i) **Reflexive**: $S <: S$

  (ii) **Transitive**: $S <: T$ and $T <: U \Rightarrow S <: U$

  (iii) **Anti-symmetric**: $S <: T$ and $T <: S \Rightarrow S = T$



- **Widening type casting**: if $S <: T$, variable of type T can automatically hold value from variable of type S

```
T x = S y;     // ok
```

Narrowing type casting: if $S <: T$, explicit typecasting from T to S (else code won't compile)

```
S x = (S) y;
```

If runtime type of target is not the same as the cast type, then runtime error will occur

3. Functions

- **Method**: Java terminology for function
- **Abstraction barrier**: separates role of programmer into implementer & client
  - Above barrier: **implementer** provides implementation
  - Below barrier: **client** uses abstraction to perform task

4. Encapsulation

- **Composite data type**: group primitive types together using a name
- **Class**: 1. **fields** 2. **methods**
- **Encapsulation**: keep data and functions related to a composite data type together
- **Object**: instance of a class

5. Information Hiding

- **Access modifiers**:

Accessed from	`private`	`public`
Inside the class	✓	✓
Outside the class	×	✓

- **Information hiding**: protect abstraction barrier from being broken by explicitly specifying if a field/method can be accessed from outside abstraction barrier
  - Private fields: prevent arbitrary changes
  - Public methods

6. Tell, Don't Ask

"Tell, Don't Ask" principle:

- Don't ask an object for its state (using getters/setters), then perform the task on its behalf
- Tell an object what to do – a task that is performed only on the fields of a class should be implemented in the class itself

7. Class Fields

- **Class fields**: associated with a class (exactly one instance throughout lifetime of the program)
  - Declare using `static`
  - `final`: value of field will not change
  - Accessed through class name w/o instantiating class
  - Universal constants: `public static final`
- **Instance field**: associated with an object

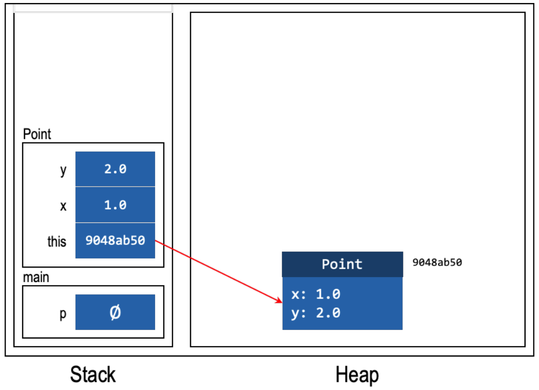
8. Class Methods

- **Class method**: associated with a class
  - Invoked without an instance, so no access to the instance's fields or methods → `this` has no meaning
  - Accessed through class name: `class.method()`
- **Instance method**: defined inside of a class
  - Varies with different instances of the class

9. Composition

- **Composition**: HAS-A relationship

10. Heap and Stack



- **Stack**: one method invoked ⇒ one **frame** created
  - Last-In First-Out
  - When method completes, frame is removed
- **Heap**: one `new` keyword ⇒ one new **object** created

11. Inheritance

- **Inheritance**: IS-A relationship (using `extends`)
- Subclass **inherits** all accessible fields/methods from superclass
  - All public fields/methods of superclass are accessible to subclass
  - Any private fields/methods of superclass are not accessible to subclass
- `super`: call superclass's constructor

12. Overriding

- `Object`: ancestor of all classes
  - `equals(Object obj)`, `toString()`
- **Method signature** = method name + number, type, order of parameters `C::foo(B1, B2)`
- **Method descriptor** = method signature + return type `A C::foo(B1, B2)`
- **Method overriding**: subclass defines instance method with same method descriptor as superclass

  Instance method in subclass **overrides** instance method in superclass
- `@Override` annotation: hint to compiler that a method is intended to override another method in superclass
  - Not needed by the compiler but good practice
  - If overridden method does not exist in the superclass, compiler generates error

13. Overloading

- **Method overloading**: when two or more methods in the same class have same name but different method signature

  If two methods have same name & method signature → not overloaded → cannot compile

14. Polymorphism

- **Polymorphism**: subclasses of a class can define their own unique behaviours, and yet share some of the same functionality of the parent class
- Use **method overriding**: the same target of invocation can invoke different methods

15. Method Invocation

- **Dynamic binding**: decide which *instance* method is invoked e.g. `curr.foo(arg);`

  Compile time step:
  1. Determine CTT(`curr`)
  2. In CTT(`curr`), find all accessible methods with name `foo`
  3. Check `arg` can bind to which method
  4. Choose the most specific one (use subtyping)

     Record its method descriptor M

  Run time step:

  5. Determine RTT(`curr`)
  6. In RTT(`curr`), find M *exactly*
  7. Found it? Execute M

     No find? Repeat search in the superclass
- Class methods, instance fields, class fields are resolved via **static binding**

16. LSP

- **Liskov Substitution Principle**: if $S <: T$, then an object of type T can be replaced by that of type S without changing the desirable property of the program
- `final`: used to prevent
  - field from being re-assigned
    (Note: a constructor is only invoked once, so there is only one assignment)
  - method from being overidden (**prevent overriding**)
  - classes from being inherited (**prevent inheritance**)

17. Abstract Class

- **Abstract method**: cannot be implemented, should not have method body
- **Abstract class**: cannot be instantiated, may provide no/incomplete implementation for its methods
  - One or more of its instance methods cannot be implemented without further details
  - A class with ≥ 1 abstract method must be declared abstract

    But an abstract class may have no abstract method
- **Concrete class**: not abstract (no abstract methods)
  - Concrete subclass of an abstract class must override abstract methods

18. Interface

- **Interface**: all methods public abstract, no fields
- **Inheritance rules**:
  - A class can extend at most one class,
    implement multiple interfaces
  - An interface cannot extend from another class,
    can extend multiple interfaces

19. Wrapper Class

- **Wrapper class**: class that encapsulates a primitive type → treat primitive types as reference types

  E.g. `Integer` for `int`, `Double` for `double`
- Primitive wrapper class objects are immutable
- **Auto-boxing**: primitive to wrapper

  Unboxing: wrapper to primitive

```
Integer i = 4;  // auto-boxing
int j = i;      // unboxing
```

- No subtyping relationship b/w wrapper classes

```
Double d=4; // Invalid: Integer /<: Double
Object o=4; // Valid: Integer <: Object
```

- Due to immutability, during every mutating operation, a new wrapper object is created

```
for (Integer i = 0; i < 10; i += 1) {
  // :
}
```

20. Runtime Class Mismatch

Narrowing type conversion e.g. `a = (C) b;`

- **Compile-time check**
  1. Check if it is *possible* for `RTT(b)` $<:$ `C`
  2. Check if `C` $<:$ `CTT(a)`
- **Runtime check**
  3. Check if `RTT(b)` $<:$ `C`

21. Variance

- Let $C(T)$ be complex type based on type T. C is
  - **covariant** if $S <: T \Rightarrow C(S) <: C(T)$
  - **contravariant** if $S <: T \Rightarrow C(T) <: C(S)$
  - **invariant** if neither covariant nor contravariant
- Java arrays of reference types are **covariant**
- Possible runtime error: we can stuff a string into an array of integers

```
Integer[] intArray = new Integer[2] {...};
Object[] objArray;
objArray = intArray;
objArray[0] = "Hello!";
```

22. Exceptions

- Handle exceptions: `try-catch-finally` block
  1. Check in the order they appear
  2. Select the *first* (and *nearest*) `catch` block that the thrown exception can bind to (use subtype relationship)

  If `ExceptionX` $<:$ `ExceptionY`, the second catch will never be executed → prevented with compilation error

```
  :
} catch(ExceptionY e) {
  // handle ExceptionY
} catch(ExceptionX e) {
  // handle ExceptionX
}
  :
```

- Throw exception:
  1. Declare method that throws exception with `throws`
  2. Create a new `IllegalArgumentException` object and throw it to the caller with `throw` keyword
- **Checked exception** $<:$ `Exception`: programmer has no control over, even if perfect code is written → actively anticipate the exception and handle them
  - Exceptions that are checked at compile time
  - Compiler forces you to either handle them with a `try-catch` block, or declare them in the method signature with `throws` (else program won't compile)
- **Unchecked exception** $<:$ `RuntimeException`: caused by programmer's errors, should not happen if perfect code is written
  - Exceptions that are not checked at compile time
  - Compiler does not force you to catch or declare them

23. Generics

- **Generic class**: takes other types as type parameters

  Parameterized type: generic type is instantiated
- Extend generics

```
class A<T> extends Pair<String, T> {
  // :
}
```

  `String` is fixed, `T` can be any type we want for `A`
- **Generic method**: parametrise a method with type parameters, without being in a generic class

```
class A {
  <T> boolean contains(T[] array, T obj)
}
```

  Calling a generic method:
  `A.<String>contains(strArray, "123")`
- **Bounded type parameters**: use `extends`

  If `T` $<:$ `GetAreable`, then `T` must have `getArea()`
- `Comparable<T>` interface: compare two things using `int compareTo(T o)` → need to override

24. Type Erasure

- Procedure:
  1. Remove angle brackets
  2. Replace types `S`, `T` with upper bound (if none given, then it is `Object`)
  3. If generic type is instantiated and used, add explicit type casting (check CTT)
  4. Add bridge method: subtype method does not override superclass method, since method signatures don't match

```
public int compareTo(Object var1) {
  return this.compareTo((Pair) var1);
      // delegate to compareTo(Pair)
}
```

- Generic array declaration is ok, but instantiation is not
  - **Heap pollution**: a variable of a parameterised type refers to an object not of that parameterised type

    Reason: Java arrays are covariant, so we can put anything into the array

    Retrieving leads to `ClassCastException`
  - **Reifiable type**: full type info available during runtime

    Java array is reifiable: Java runtime can check what is stored in array, whether it matches the type of array → if mismatch, `ArrayStoreException`

    Java generics are not reifiable: due to type erasure - type information missing during runtime

25. Unchecked Warnings

- Generics are **invariant**: no subtyping relationship
- **Unchecked warning**: message from compiler - runtime error it can't prevent due to type erasure
  - Compiler unsure if a type operation is safe
- `@SuppressWarning("unchecked")` annotation: suppress warning messages from compiler, assure compiler that type operation is safe
- **Raw types**: generic type used w/o type arguments

```
Seq s = new Seq(4); // compiles
```

26. Wildcards

- **Wildcard**: can be substituted for any type
- **Unbounded wildcard**: `C<?>`
  - `C<T>` $<:$ `C<?>`
- **Upper-bounded wildcard**: `C<? extends T>`
  - **Covariant**: if `S` $<:$ `T`, then
    `C<? extends S>` $<:$ `C<? extends T>`
  - `C<T>` $<:$ `C<? extends T>`
- **Lower-bounded wildcard**: `C<? super T>`
  - **Contravariant**: if `S` $<:$ `T`, then
    `C<? super T>` $<:$ `C<? super S>`
  - `C<T>` $<:$ `C<? super T>`
- **Producer Extends; Consumer Super**
- Raw types not allowed; use unbounded wildcards
  - `a instanceof A<?>`: works in `instanceof`
  - `new Comparable<?>[10]`: instantiate generic arrays

27. Type Inference

- **Diamond operator** `<>`: no need to declare type arguments twice when instantiating generic type

```
Pair<String, Integer> p = new Pair<>();
```

  Inside the diamond, the type is inferred to be the declared type (CTT)
- **Local type inference algorithm**:
  1. Write down all local type constraints
     - Target typing
     - Argument typing
     - Type parameter bound
  2. Solve type constraints
  3. Choose most specific one (mentioned or superclass)