CS2030/S Term Review (Unit 1-27)

*All summative assessments, i.e., practical exams and written exams, are **closed book** assessments.*

Program and Compiler

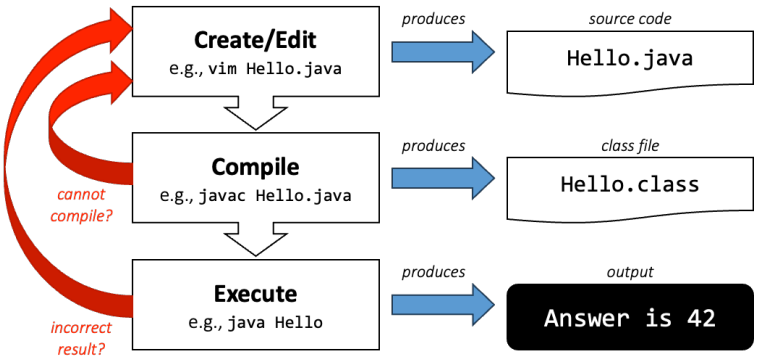
1. Java programs can be excute in 2 ways
   - Java $\rightarrow$ bytecode $\overset{JVM}{\rightarrow}$ machine code
   - Interpreted by the Java interpreter.
2. Compiling and Running Java Programs
   - `$ javac Hello.java`
     `javac` is the Java compiler, `*.java` $\rightarrow$ `*.class`
   - `$ java Hello`
     Invoke the JVM `java` and execute the bytecode contains in `Hello.class`,
     JVM language(bytecode) $\rightarrow$ x86-64 machine language
3. Interpreting a Java program(Using `jshell`)
   - `$ jshell`
     interpreter `jshell` interprets `Hello.java` interpreted from Java **directly** to the x86-64 machine language.
4. Compiler's

   (1) Job
   - Translate source code into machine code or bytecode
   - Parse the source code written and check grammar and return error if grammar is violated
   - Detect any syntax error **before** the program is run. (During compilation)

   * **More Specific**
   - Violations to *access modifiers* are checked by the compiler. Trying to access, update, or invoke fields or methods with `private` modifier will give a *compilation error*.

   (2) Cannot tell
   - if a particular statement in the source code will ever be executed
   - what values a variable will take

   (3) Behavior
   - **Conservative:** report an error as long as there is a possibility that a particular statement is incorrect
   - **Permissive:** If there is a possibility that a particular statement is correct, it does not throw an error, but rely on the programmer to do the right thing.
5. Workflow



Compile-time errors are better than run-time errors, but the compiler cannot always detect errors during compile time.

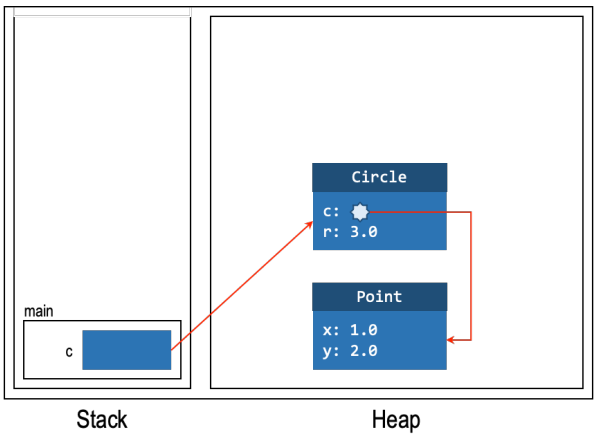
Heap and Stack

1. JVM manages the memory of Java programs while its bytecode instructions are interpreted and executed.
2. **Stack** for local **variables** and call frames.
   * **Note that** instance and class fields are not variables → fields are not in the stack.
   - Variables are contained within the call frames (created when we invoke a method and removed when the method finished).
   - Use ∅ to indicate that the variable is not yet initialized
   - When instance method is called, the JVM creates a stack frame for it, containing
     - the `this` reference
     - the method arguments
     - local variables within the method, among other things

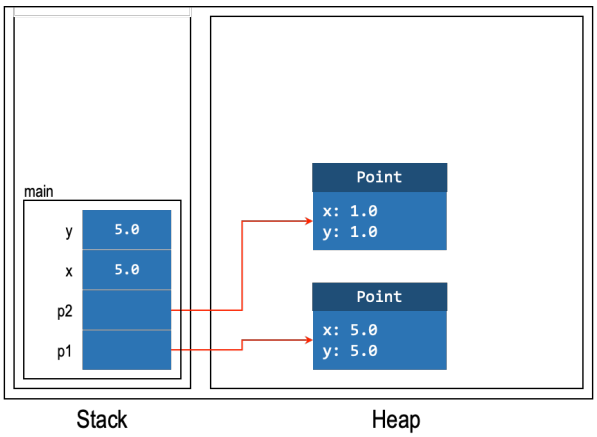
   When a **class method** is called, the stack frame **does not** contain the `this` reference.
   * After the method returns, the stack frame for that method is destroyed.
3. **Heap** for storing dynamically allocated objects
   Whenever you use the keyword `new`, a new object is created in the heap. An object in the heap contains the following information:
   - Class name.
   - Instance fields and the respective values.
   - Captured variables.

   * The memory allocated on the heap stays there as long as there is a reference to it (either from another object or from a variable in the stack). You do not have to free the memory allocated to objects. The JVM runs a garbage collector that checks for unreferenced objects on the heap and cleans up the memory automatically.
4. Heap and Stack Diagram
   - We can omit both memory addresses stored in the variable and of the object.
   - The intermediate call frames (e.g., Point constructor) can be omitted. Only the final effect matters.

Example 1



Example 2



Varible and Type

1. Dynamic and Static Type
   - **Dynamically typed** language: The same variable can hold values of different unrelated types.
   - **Statically typed** language: The variable can only hold values of that declared type.
2. Compile-time type and Run-time type
   Compile-time Type
   - The only type that the compiler is aware of
   - The compiler will check if the compile-time type matches → Error if there is a **type mismatch**

   Run-time Type The type actually created using the `new` keyword. Assume `T x = new S();`, Compile-time type of `x` is `T`. Run-time type of `x` is `S`.
3. Strong Typing vs. Weak Typing
   - **Strong Typing:** Enforces strict rules in its type system, to ensure type safety and allows typecasting only if it makes sense.(Java) It can detect Type Mismatch during Compile-time
   - **Weak Typing:** More permissive in terms of typing checking.(C) It can detect Type Mismatch only at Run-time

   Example

   ```
   i = 4;
   i = (int)"5";
   ```

   Java will throw an error while C will allow this typecasting
4. Primitive Type and Reference Type
   Primitive Type variables never share their value with each other!

   Reference Type stores only the reference to the value, and therefore two reference variables can share the same value. If they share the same value, changing one of them will change another one as well.

   Share Reference (Aliasing): To avoid aliasing, one solution is to avoid sharing references as much as possible.
   we are sacrificing the computational cost to save programmers from potential suffering!
5. **Subtypes and TypeCasting**
   - We use $T <: S$ to denote that T is **subtype** of S. It follows 3 properties: Reflexive, Transitive, Anti-Symmetry.
     Anti-Symmetry: If $S <: T$ and $T <: S$, then S must be the same type as T.
   - **Widening type conversion:** Java allows a variable of type T to hold a value from a variable of type S only if $S <: T$.
   - **Narrowing type conversion** requires *explicit typecasting* `(Type) expr` and *validation* during run-time.

*Type Checking

- Java allows a variable of type T to hold a value from a variable of type S only if $S <: T$
- Java will only use compile-time type information for its type checking.
- Tips 1

```
class T { foo() {...} }
class S1 extends T { bar() {...} }
class S2 extends T { baz() {...} }
T x = new S1();
x.bar(); // Error, because view of compiler is CTT. Since type T has no method called bar
```

- Tips 2 (*anti-symmetry property*) Java prevents a cyclic subtyping relationship.

```
class A extends B { }
class B extends A { } // Error!
```

- Tips 3 **Nominal subtyping:** subtyping relationship has to be *explicitly* declared

Consider `a = (C) b;`

Compile Time Check

1. Find CTT(`b`)
2. Determine whether there is a *possible* RTT(`b`) <: `C`, if impossible → compilation error
   Some *possible* cases
   - `CTT(b) <: C`: simply widening
   - `C <: CTT(b)`: narrowing and requires run-time checks.
     Consider `C <: B`:
     - If CTT(`b`) = `B` and RTT(`b`) = `C` (or subtype of `C`), then it is allowed at run-time.
     - If CTT(`b`) = `B` and RTT(`b`) = `C` (or other subtype of B that is not `C`), then it not allowed at run-time. Since there is a possibility, the compiler will add codes to check at run-time.
   - `C` is an interface
     - Let RTT(b) = B. Then it may have a subclass A such that A <: C (i.e., implements the interface C). `class A extends B implements C { .. }`
     - If RTT(b) = A, then it is allowed at run-time.
3. Find CTT(`a`)
4. Determine whether there is a possible `C` <: RTT(`a`), if impossible → compilation error
5. Runtime Check for RTT(`b`) <: `C`

Run Time Check

1. Find RTT(`b`)
2. Check if RTT(`b`) <: `C`

Common Subtype Relation

1. **Type Casting:** $A <: B \rightarrow$ `B b = a` will compile
2. **Inheritance:** `A extends B` → $A <: B$
3. **Interface:** If a class C implements an interface I, $C <: I$. This definition implies that a type can have multiple supertypes.
   - Specifically, `I i2 = (I) new A();` compiles even though `A` does not implement `I`
4. **Warp Class:** `Integer` <: `Number` <: `Object` → `Integer[]` <: `Number[]` <: `Object[]` (Covariant)
5. **Exception:** `Exception a` <: `Exception b` → `catch(Exception a)` **cannot** catch the `Exception b`

Type Inference

1. **Definition**
   - **Argument Typing:** Type of argument is passed to parameter.
   - **Target Typing:** Return type is passed to variable.
   - **Type Parameter:** The declared type, especially for bounded type parameter.
2. **Bound Constrain**
   - `Type1 <: T <: Type2`, then `T` is inferred as `Type1`
   - `Type1 <: T2`, then `T` is inferred as `Type1`
   - `T <: Type2`, then `T` is inferred as `Type2`
3. **Methodology:** Consider `public <T extends Circle> T bar(Array<? super T> array)`
   We have `GetAreable c = bar(new Array<Circle>());` Hence
   - **Argument Typing:** `Array<Circle>` is passed into `Array<? super T>` $\rightarrow$ `?` = `Circle` $\rightarrow$ `T <: Circle`.
   - **Target Typing:** `T` is assigned to `GetAreable` $\rightarrow$ `T <: GetAreable`.
   - **Type Parameter:** `<T extends Circle>` $\rightarrow$ T <: Circle.

   Therefore, pick the **most specific** one, `T <: Circle` .
4. **Unexpected Consequences:** For `String[] strArray = new String[] { "hello", "world" };`
   We have `A.contains(strArray, 123); // ok!`
   As we have `Object <: T <: Object`, hence code above is equal to `A.<Object>contains(strArray, 123);`

OOP Principle

1. **Information Hiding:** Expose as few fields/methods as possible
   - Isolate the internal representation of a class using an abstraction barrier.
   - Data hiding using `private`
   - Constructors / The `this` Keyword
2. **"Tell, Don't Ask" principle:** The client should tell an object what to do, instead of asking to get the value of a field, then perform the computation on the object's behalf.
3. **Liskov Substitution Principle (LSP)** If a class `B` is substitutable for a parent class `A` then it should be able to pass all test cases of the parent class `A`. If it does not, then it is not substitutable and the LSP is violated.
   - Ensure that any inheritance with method overriding does not introduce bugs to existing code
   - Let $f(x)$ be a property provable about objects x of type T. Then $f(y)$ should be true for objects y of type S where $S <: T$.
   - LSP cannot be enforced by the *compiler*
   - A subclass should not break the expectations set by the superclass.

Definition

1. **Class:** A data type with a group of functions associated with it.
2. **Fields:** Data in the class.
   class field: `static` fields that are associated with a class
   - Behaves like a global variable
   - Note that a `static` class field needs not be `final` and it needs not be `public`.
   - We can think `static` fields as having **exactly one** instance during the *entire execution of the program*

   instance field: fields that are associated with an object

3. **Methods:**
   class method
   - is always invoked without being attached to an instance
   - cannot access its instance fields or call other of its instance methods.
   - the reference `this` has no meaning within a class method.
   - should be accessed through the class
4. **Objects:** Instances of a class, each containing the same set of variables of the same types, but (possibly) storing different values.
5. **Constructor:** A method that initializes an object. In Java, a constructor method has the *same name* as the class and *has no return type* (i.e. `public Circle (para.)`), and the access modifiers can actually be omitted (i.e. `Circle(para.)`).
   When the constructor is called by the keyword `new`:
   - Allocate sufficient *memory location* to store all the fields of the class and assign this reference to the keyword this.
   - Invoke the constructor function and **passing the keyword `this` implicitly**.
   - Once the constructor is done, **return the reference** pointed to by this back.

   Hence, `Circle c = new Circle(..)` actually assign the refernce of newly created object to the variable `c`.
6. **Default Constructor:** If there is no constructor given, then a default constructor is added automatically. The default constructor *takes no parameter* and has *no code written for the body*.
7. **Fully Qualified Name:** A name that can always be used to **unambiguously** refer to a specific name, be it variable or field, i.e. `Circle.this.r`.
   - always starts with a sequence of class names separated by a dot.
   - If the name refers to a field, the FQN is then followed by the keyword `this`. Otherwise, there is no keyword `this`.
   - Finally, the FQN is followed by the actual name used.
8. **Composition** models that HAS-A relationship between two entities.
9. **Inheritance** models that IS-A relationship between two entities.
   - if a given class does not *explicitly* inherit another class, it will *implicitly* inherit from the `Object` class
10. **Method Summary**
    - **Method Signature:** Only the name of the method and the type of parameters. Optionally, we may include the class name. (i.e. `C::foo(B1, B2)`)
      - Method name.
      - Number of parameters.
      - Types of parameters.
      - Order of parameters.
    - **Method Descriptor:** Only the name of the method, the type of parameters, and the return type. Optionally, we may include the class name.(i.e. `A C::foo(B1, B2)`)
      - Method name.
      - Number of parameters.
      - Types of parameters.
      - Order of parameters.
      - Return type.

11. **Override** When a subclass defines an instance method with the same **method descriptor** as an instance method in the parent class, we say that the instance method in the subclass *overrides* the instance method in the parent class.
    Generally, consider `S<:T`, `A2<:A1`, then `A1 T::foo(B,C)` can be overriden in class `S` as `A2 S::foo(B,C)`

12. **Overloading:** We create an overloaded method by changing the type, order, and number of parameters of the method but keeping the method name identical. **Note that** `contains(double x, double y)` and `contains(double y, double x)` are not distinct methods and they are not overloaded
    - It is possible to overload the class *constructor* (see `this` key word)
    - It is possible to overload `static` class methods

    Attention Here As subclass inherits all the method from the superclass, in the following example, we still have overloading.

    ```
    class T { void foo(double x) { } }
    class S extends T { void foo(int x, int y) { } }
    ```

13. **Abstract Class:** A class that has been made into something so general that it **cannot and should not** be instantiated.
    - If a class has **at least one** abstract method (Not all the methods have to be abstract.), it must be declared as an abstract class with the keyword `abstract` in the class declaration.
    - On the other hand, an abstract class **may have no** abstract method. (Not Symmetric)
    - An `abstract` method cannot be implemented and therefore should not have any method body.
      i.e. `public abstract double getArea(); // No method body, ends with a semicolon`
    - The subtype of abstract class inherits the abstract methods unless the method is **overridden**. If the inherited abstract method is not overridden by the subclass, it means that the subclass has at least one abstract method. Therefore, the subclass must also be declared abstract.
    - On the other hand, an abstract class may inherit from concrete class. Even if the abstract class does not have abstract method and does not inherit one, it can still be declared as an abstract class simply to **prevent instantiation** of the class.

14. **Interface:** The abstraction models what an entity can do.
    - All methods declared in an interface are `public abstract` by default.
    - A class implementing an *interface* must be an abstract class. On the other hands, for a class to implement an interface and be concrete, it **has to override all abstract methods** from the interface and provide an implementation to each
    - Know about what is impure interface (It will not be talked in the future of CS2030/S)
    - **Multiple Inheritance**

15. **Wrapper Class:** A class that encapsulates a type, rather than fields and methods.
    - All primitive wrapper class objects are **immutable** —— once you create an object, it cannot be changed.
    - Using an object comes is **less efficient** than primitive types.
      i.e. To find sum of number, as primitive wrapper class objects are **immutable**, thus, every time the sum in the first example above is updated, a new object gets created.

16. **Autoboxing:** Like `Integer i = 4;` where the primitive value `int` of 4 is converted into an instance of `Integer`.
    - **Single-step process**. Hence, code `Double d = 2;` will not compile, as we need to convert `int` to `double` first, and then "put it in the box", which cost 2 steps

17. **(Auto-) Unboxing:** We need to open the box, follow the arrow, and only then we can retrieve the value.

- Auto-unboxing is not restrict in only one step, hence the following code will compile.

```
Integer x = 4; // auto-boxing
double d = x; // auto-unboxing and primitive subtyping
```

18. **Variance:** Variance of types refers to how the *subtype* relationship between *complex types* relates to the subtype relationship between components.
    - Java Array is Coverient. We only create the array type for *reference type*.
    - **covariant** if $S <: T$ implies $C(S) <: C(T)$
    - **contravariant** if $S <: T$ implies $C(T) <: C(S)$
    - **invariant** if it is neither covariant nor contravariant.
    - Due to covariance of Java array, there are some run-time errors that cannot be detected by our compiler. An example of a type system rule that is unsafe

```
Object[] obj <-- intArray // ok
Integer[1] = "Hello" // ok, as String <: Object. But will have runtime
                     // error, as assign String to a integer array!
```

Keywords

1. `null`: Any reference variable that is **not initialized** will have the special reference value `null`
    - In Java, uninitialized variables $\neq$ variables initialized to `null`
    - Uninitialized variables cannot be used
    - Uninitialized fields have default values but not uninitialized variables.
2. `private`: Fields and methods with `private` access modifier can only be accessed from within the class.
3. `this` is a reference variable that refers back to self, and is used to distinguish between two variables of the same name.
    - Expected behavior of `x = x` rather than `this.x = x`? Do nothing! The field has not been updated with the value of parameter `x`.
    - `this` can be *automatically added* if there is no ambiguity that we are referring to the field.
    - Use `this(...)` at the **first line** to chaining constructor. It will invoke our original constructor.
    - Cannot have both call to `super(..)` and call to `this(..)`
4. `final` keyword can be used in three places:
    - In a class declaration to prevent inheritance.
    - In a method declaration to prevent overriding.
    - An *optional* modifier for the `main` method.
    - In a field declaration to prevent re-assignment.
5. `static`: a field or method with modifier `static` belongs to the class rather than the specific instance. In other words, they can be accessed/updated or invoked without even instantiating the class.
   * **Note that**
    - Java actually prevents the use of this from any method with static modifier, or it will throw an error.
    - On the other hand, non-static methods including constructor has the keyword `this`.
6. `extends` use for any type of **subtyping** when inheritance or declaring a bounded type parameter, *even if the supertype is an interface.*
    - A class can only extend from one superclass, but it can implement multiple interfaces.
    - An interface can extend from one or more other interfaces, but an interface cannot extend from another class.
7. `super` call the constructor of the superclass
    - has to appear as the first line in the constructor
    - if there is no call to `super`, Java will automatically add the default `super()`

- No `super` – 1 error, while Non-first `super` – 2 error
- The methods that have been overridden can be called with the `super` keyword.
- Cannot have both call to `super(..)` and call to `this(..)`

8. `instanceof` Assume `obj instanceof Circle`, it returns true if `obj` has a **run-time type** that is a **subtype** of `Circle`.
9. `abstract`: Declare an abstract class/method.
10. `throws`: Used in method declaration, throw an exception
11. `throw`: Actual throwing of exceptions.
12. `try/catch/finally`: group statements that check/handle errors together making code easier to read.

*Method Invocation

If we want to invoke `obj.foo(arg)`

During Compile Time

1. Determine `CTT(obj)` and `CTT(arg)`
2. Determine all the methods with the name `foo` that are accessible in `CTT(obj)`, including the parent of `CTT(obj)`, grand-parent of `CTT(obj)`, and so on.
   - An abstract method is considered accessible although there is no method implementation.
   - RTT(`obj`) must be a concrete class → must have an implementation of the abstract method → the implementation of the abstract method must override the abstract method
3. Determine all the methods from Step 3 that can accept CTT(arg).
   - Correct number of parameters.
   - Correct parameter types (i.e., **supertype of `CTT(arg)`**).
4. Determine the **most specific** method. If there is no most specific method, fail with compilation error.

- **Specific:**
  - Assume `A<:B` → `foo(A)` is more specific than `foo(B)`
  - Given `S1 <: T` and `S2 <: T`, `foo(S1)` is not more specific than `foo(S2)` and `foo(S2)` is not more specific than `foo(S1)`.

During Run Time

1. Determine `RTT(obj)`
2. Starting from `RTT(obj)`, find the first method that match the method descriptor (continue up the class hierarchy)

Example

Consider

```
class U {
  void foo(T t) { }
  void foo(U u1, U u2) { }
}
class T extends U {
  void foo(S s) { }
}
class S extends T {
  void foo(U u) { }
}
```

```
U u = new T();
S s = new S();
```

1. `u.foo(s)`
2. `u.foo(u)`
3. `s.foo(s)`
4. `s.foo(s)`

Invocation of Class Methods

Class methods do not support dynamic binding.

- The method to invoke is resolved statically during compile time.
- The corresponding method implementation in class will always be executed during run-time, without considering the run-time type of the target.

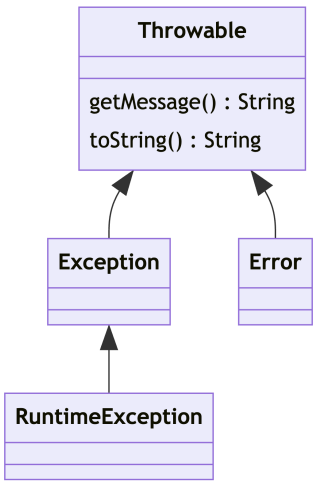
Exception

1. The `try`/`catch`/`finally` keywords group statements that check/handle errors together making code easier to read.
2. Exceptions are instances that are a subtype of the `Exception` class.
3. `finally` is ALWAYS Executed!
4. **Throwing Exceptions**
   - Declare that the construct is throwing an exception, with the `throws` keyword.
   - Create a new Exception object and throw it to the caller with the `throw` keywords. (`throws` $\neq$ `throw`)
   - `throw` statement causes the method to **immediately return** (the following part will not happen!!!)
5. **Unchecked Exceptions:** (Perfect Code should not have) Not explicitly caught or thrown $\rightarrow$ something is wrong with the program and cause **run-time errors** (`unchecked exceptions` $<:$ `RuntimeException`).
   - Unchecked Exceptions $\rightarrow \begin{cases} \text{caught} \\ \text{Not caught} \rightarrow \text{Error message} \end{cases}$
6. **Checked Exceptions:** (No Control Over) A checked exception must be either handled, or else the program **will not compile**.
   - A checked exception must be handled.
7. **Normal Execution** `try` $\rightarrow$ `finally`
8. **Error Execution** `try` $\rightarrow$ `catch` $\rightarrow$ `finally`
   - `error` find $\rightarrow$ all subsequent lines in the try block is **not** executed
   - look one-by-one from top to bottom for the first `catch` block
   - continues from the `catch` block to the `finally` block
   - Consider `ExceptionX <: ExceptionY`, then the following code

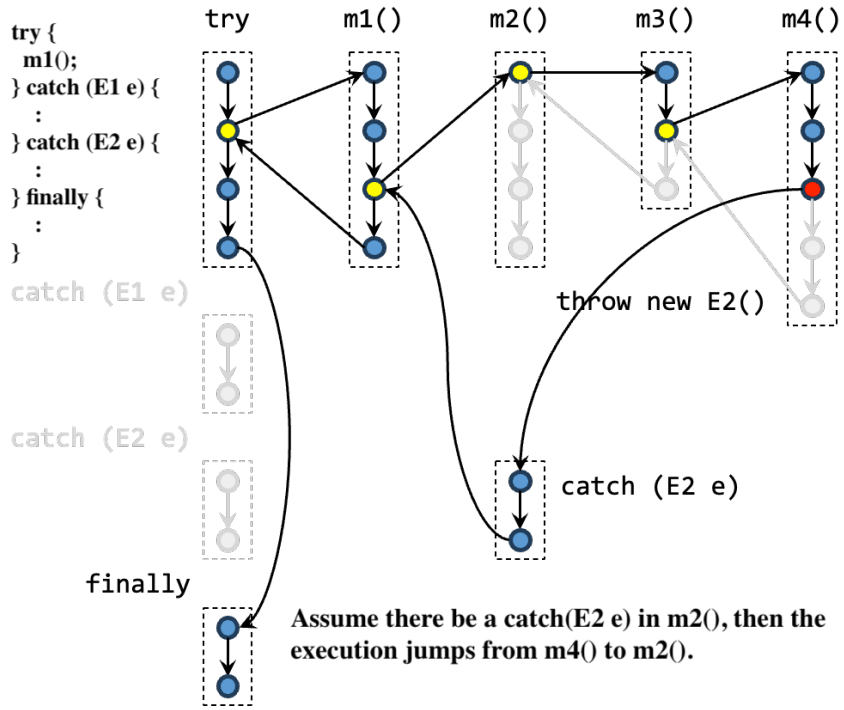
     ```
     catch(ExceptionY e) {// handle ExceptionY}
     catch(ExceptionX e) {// handle ExceptionX}
     ```

     will cause a compilation error since we will never catch `ExceptionX` because it will already be caught by `catch(ExceptionY e) { .. }`

     Common Mistake: use `catch(Exception e)` **above** other `catch` bolcks that hands the subclass of `Exception` $\rightarrow$ Compilation Error! DO NOT USE `catch(Exception e)` *Pokemon Exception Handling*
9. **Overriding**
   - Override a method that throws a checked exception $\rightarrow$ the overriding method must throw only the same, or a more specific checked exception, than the overridden method.(By LSP)
   - The caller of the overridden method cannot expect any **new** checked exception beyond what has already been "promised" in the method specification.
10. **`Error` class** `Exception` and `Error` $<:$ `Throwable`.
    `e` $\in$ `Throwable` or sub(`Throwable`) $\rightarrow \begin{cases} \text{thrown by the throw} \\ \text{caught by the catch block} \end{cases}$
11. **Control Flow**



Error Class Diagram



Generic

1. Java allows us to define a **generic type** that takes other types as **type parameters**, just like how we can write methods that take in variables as parameters. Consider `class Pair<S,T>`
   - `S,T` are type parameters, `Pair<S,T>` is generic type
   - To instantitated a generic type, we pass in **type arguments** and only *reference types* (i.e. `Integer`) can be used as type arguments.
   - Once a generic type is instantiated, it is called a **parameterized type**.
   - We can pass the type parameter of a generic type to another, consider
     `class DictEntry<T> extends Pair<String,T>`
     Type `T` is passed from the `DictEntry` class to the `Pair` class. $\iff$ We declare a generic type `T` in class `DictEntry<T>` and use the generic type `T` in `.. extends Pair<.., T>`.
2. Generic Methods
   - The type parameter `T` is declared within < and > and is added **before** the return type of the method.
   - `<T>` only scoped in its own method
   - To call a generic method, we need to pass in the type argument placed before the name of the method
3. **Type Parameter Confusion:** Declare type parameter with the same name but are actually different type
4. **Bounded Type Parameters**
   - Compiler won't compile if it is not sure that it can find the method `getArea()` in type `T`
   - Hence, we put a constraint on `<T>` using `<T extends GetAreable>` $\rightarrow$ `T` is a subtype of `GetAreable`
   - Use `Comparable<T>`, Consider
     `class Pair<S extends Comparable<S>,T> implements Comparable<Pair<S,T>>`
5. **Type Erasure:** Java *erase* the type parameters and type arguments during **compilation**, resulting compiled code **does not** to be recompiled when encountering new types.

- Each type parameter `S` and `T` are replaced with Object. If the type parameter is **bounded**, it is replaced by the **bounds** instead
- Generate Code is generated by the **compiler** after *type checking*

6. **Generics and Array**
   - **Heap pollution:** A variable of a parameterized type refers to an object that is not of that parameterized type.
   - Array is what is called **reifiable type**, a type where full type information is available during **run-time**.
   - Java generics is **not** reifiable due to type erasure.
   - Generic array **declaration** is fine but generic array **instantiation** is not

   How to get around this -- `ArrayList`
   - Array is **covariant**
   - Generics are **invariant** → no subtyping relationship → preventing the possibility of heap pollution

```
private T[] array;
Array(int size) {
  this.array = (T[]) new Object[size]; // we cannot new T[] directly
}
```

7. **Unchecked Warnings:** Use `@SuppressWarning`, a powerful annotation that suppresses warning messages from compilers.
   - We must always use `@SuppressWarning` to the most limited scope to avoid unintentionally suppressing warnings that are valid concerns from the compiler.
   - We must suppress a warning only if we are sure that it will not cause a type error later.
   - We must always add a note (as a comment) to fellow programmers explaining why a warning can be safely suppressed.
8. ***Raw type**: A generic type used **without type arguments**.
   - Without a type argument → the compiler can't do type checking → uncertainty (`ClassCastException`)
   - Mixing raw types with paramterized types can also lead to errors.
   - Raw types must not be used in your code, ever. The only exception to this rule is using it as an operand of the `instanceof` operator.
9. **Wildcards:** `A.<Object,Object>foo(circles, c)` will not compile
   - Generics is invarient → `Seq<Circle>` <: / `Seq<Object>`
   - We have to contrain array type to be same in Generics
   - Why inveriant? Avoided the possibility of run-time errors by avoiding covariance arrays
10. **Upper-Bounded Wildcards:** `Array<? extends S>`, an example of **covariance**
    - If `S <: T`, then `A<? extends S> <: A<? extends T>` (covariance)
    - For any type `S`, `A<S> <: A<? extends S>`
11. **Lower-Bounded Wildcards** `Array<? super T>`, an example of **contravariance**
    - If `S <: T`, then `A<? super T> <: A<? super S>` (contravariance)
    - For any type S, `A<S> <: A<? super S>`
12. **PECS:** "Producer Extends; Consumer Super"
13. **Unbounded Wildcards:** `Array<?>` is the **supertype** of every parameterized type of `Array<T>`
    - **Pretty restrictive:** For following code, `x` has to be `Object` and `y` has to be `null`

```
void foo(Array<?> array) {
  :
  x = array.get(0);
  array.set(0, y);
}
```

- `Array<?>`, `Array<Object>`, and `Array`
  - `Array<?>` is an array of objects of some specific, but unknown type;
  - `Array<Object>` is an array of `Object` instances, with type checking by the compiler;
  - `Array` is an array of `Object` instances, without type checking.
- **Revisiting Raw Types** We can use unbounded Wildcards like
  - `a instanceof A<?>`
  - `new Comparable<?>[10];`

14. **Reifiable type:** A type where no type information is *lost* during compilation. `Comparible<?>` is reifiable. Since we don't know what is the type of `?`, no type information is lost during *erasure*!

PE Guide

1. When to Stop? You may want to ask the following questions.
   - Is there multiple properties to be stored?
   - Is there an action associated with the entity?
   - Is there a real world counterpart?
   - Is there potential changes to the entity?
2. Class Diagram in CS2030S
3. **OOP Principle**
   - It is better to not define classes with any accessor and modifier to the private fields, and forces yourselves to think in the OO way
   - Typically we want to minimize the number of accessors and mutators that we have.
   - Use composition to model a has-a relationship; inheritance for a is-a relationship. Make sure inheritance preserves the meaning of subtyping
4. **Exceptions**
   - Do not exit a program just because of an exception. → prevent the calling function from cleaning up their resources
   - Do not exit a program silently.
   - Do not use `Exception e` (Pokemon Exception Handling)