Software program

- a collection of data variables and instructions on how to modify those variables.
- Translation from a high-level programming language to machine code.
  - Compiler. (Software tool)
    - machine code is saved into an executable file which can be executed later.
  - Interpreter
    - Software that reads in the program one statement at a time, interprets it, then executes it.
  - Java programs are executed in 2 ways.
    - compiled into bytecode, then during execution the bytecode is interpreted and compiled on-the-fly by JVM into machine code.
    - Java program can also be interpreted by the Java interpreter.
- Java is a statically typed language - once compile-time type of a variable is declared, it cannot be changed

- type system: a set of rules that govern how types can interact with each other.
- Java compiler can help check for type errors.

OOP

- int initialises to 0
- Reference Types initialises to null
- Encapsulation
  - exposes just the right method interface for others to use
  - maintains the abstraction barrier.
  - concept of keeping all data and functions operating on the data related to a composite data type together
- keyword 'new' creates an object of a given class
- '.' notation to access an object's fields and methods.
- Information Hiding
  - private, public modifier. — private ⇒ cannot access even if child class.
  - protects the abstraction barrier from being broken. — client vs implementer
  - protection is enforced by compiler at compile time.
- keyword 'this' - reference variable that refer back to itself.
  - distinguishes between 2 variables of the same name.
- Tell, Don't Ask.
  - accessor (getter), mutator (setters)
  - if we provide a setter and getter for every private field ⇒ breaking encapsulation.
  - Right approach: Implement a method within the class that does whatever we want the class to do.
    In such a way that when the implementer decides to change the representation of the object and remove the direct access to the field, the client does not need to change anything.
- To associate a method or field (global values) with a class in java, use 'static'
- 'final' to indicate the value of the field will not change.

- we should tell an object what task to perform as a client, rather than asking for the data and doing the computation ourselves.

super()
- calls no args constructor if not specified in child class
  - if no args constructor exist.

Class fields/methods.
- associates with a class, not an instance of a class.

Class fields are useful for storing pre-computed values / configuration parameters associated with a class rather than individual objects.

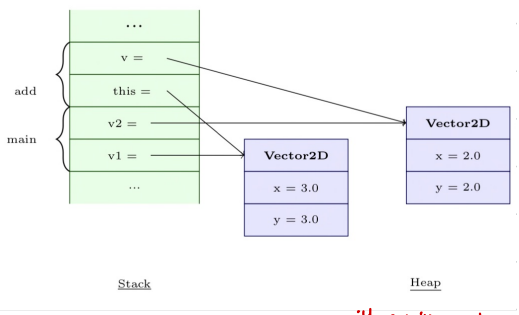
Hard Stuff.

Composition.

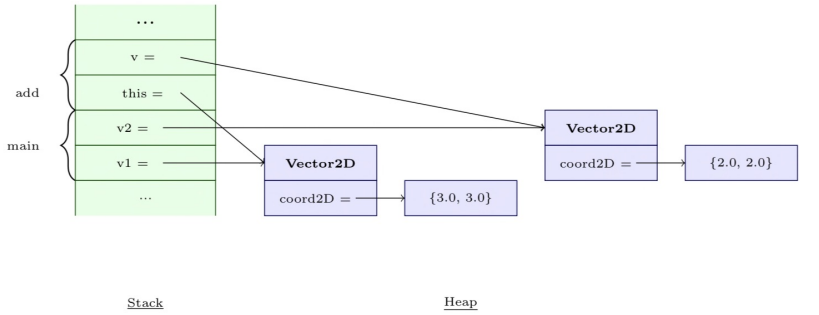
- models a HAS-A relationship between 2 entities
  - eg. circle has a point as the center.
- Sharing references (Aliasing)
  - common source of bugs
  - avoid → even though computational resources is not optimised as there are tons of objects.
  - use immutability.

Heap and Stack

- typically JVM partitions the memory into several regions:
  1. Method area → storing code for the methods.
  2. Metaspace → storing meta information about classes.
  3. Heap → storing dynamically allocated objects
  4. Stack → local variables and call frames
- Heap is the region in memory where all objects are allocated in and stored. → memory stays as long as there is a reference to it.
- Stack is the region where all variables (incl. primitives and object references) are allocated in and stored. → memory deallocated when a method returns
- Java call by value for primitive types, call by reference for objects



with primitive val.



with objects (eg. coords)

Inheritance

- subtyping → If S <: T, then any piece of code written for T must also works for S.
- using extends
- super calls the constructor for superclass.
- model for IS-A relationship.

Use composition (compose a new class) for HAS-A relationship

Use inheritance for a IS-A relationship. Make sure inheritance preserves the meaning of subtyping

Run-Time type

e.g. ColouredCircle <: Circle

Circle c = new ColouredCircle(p, 0, blue)

Circle is the compile-time type of c.

c is now referencing an object of subtype ColouredCircle

Since the assignment happen during run-time,

∴ runtime type of c is ColouredCircle.

Method Overriding

- able to alter the behaviour of an existing class.
- Method signature: method name + number, type and order of its parameters.
- Method descriptor: method signature + return type.
- when a subclass defines an instance method with the same method descriptor as an instance method in the parent class,
  instance method in the subclass overrides the instance method in the parent class

Method Overloading

- When we have 2 or more methods in the same class, with the same name
  but differing method signature.
- change a method's parameter number, type, order to overload it.
- says nothing about the name of the arguments
- possible to overload the constructor and static class methods.

Polymorphism

- related to method overriding.
- which method to invoke is decided during runtime, depending on the runtime type of the Object.
  - dynamic binding.

```
@Override
public boolean equals(Object obj) {
  if (obj instanceof Circle) {
    Circle circle = (Circle) obj;
    return (circle.c.equals(this.c) && circle.r == this.r);
  }
  return false;
}
```

type casting.

- equals takes in a parameter of CT-type Object.
  checks RT-type of obj if its a subtype of Circle (using instanceof)
  type cast assures the RT-type of obj is Circle
- Java is strongly typed and very strict about type conversions.
  - allows type casting from T → S if S<:T. (narrowing type conversion)
    - needs the type casting to be explicit and validation during run-time. (bad casting leads to runtime error).
  - (widening type conversion) is always allowed

Method Invocation

- Dynamic Binding → only applies to instance method invocations

Step 1: Compile Time

- Java determines the method descriptor of the method invoked, using the compile-time type of the target.
- multiple method that could be correctly invoked?
  - choose most specific one.

    method M is more specific than N if the arguments to M can be passed to N without compilation errors.
- stores method descriptor in the generated bytes

  (do not incl. info about class that implements this method.)
  - determined during RT.

Step 2: Run Time

- when a method is invoked, the method descriptor from CT is retrieved. ← CT type of the argument.

  Then RT - type of target is determined. → R.

  Java then looks for an accessible method with the matching descriptor in R

  If no such method is found → search will continue up the class hierarchy.
- first method implementation with a matching descriptor found will be the one executed.

Class methods (static) does not support dynamic binding.
- method to invoked is resolved during CT.
- step 1 will be taken, but step 2 will be 'just executing code found in step 1.

LSP

- Let $\phi(x)$ be a property provable about objects x of type T.

  Then $\phi(y)$ should be true for objects y of type S where $S <: T$.
- A subclass should not break the expectations set by the superclass.

  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.
- final class → bans a class from being inherited.

  final method → bans from being overriden. → critical for the correctness of the class.
- In another words, subclass must have the same implementation, but more detailed than the superclass.

Wrapper Class

- wraps around primitive types
- they are reference types, created with 'new'
- instances are stored on the heap.
- immutable - once created, cannot change.
- auto-boxing / unboxing.
- cost of allocating memory for the object and collecting garbage.
  - less efficient

Abstract Class

- a very general class that should not / could not be instantiated.
- one or more of its instance methods cannot be implemented without further details. → keyword 'abstract'
- reverse: Concrete class (no abstract method)

Interface

- is a type declared with interface
- interface can extends multiple interfaces,

  class can only extend one class, implements multiple interfaces
  - If class C implements an interface I, then C<:I → a type can have multiple supertypes

- fields in interface are static and final.

  but not for abstract class

Types

Casting

- Ask compiler to treat the object returned.
- Need to be sure about the run-time type.
- Leads to fragile code → depends on RT type, compiler won't catch the error.

Variance of types

- How the subtype relationship between complex types relates to the subtype relationship between components.
- Covariant if $S <: T \rightarrow C(S) <: C(T)$

  Contravariant if $S <: T \rightarrow C(T) <: C(S)$

  invariant if neither.
- Java Array is Covariant

  `S <: T`

  `S[] <: T[]`.
  - prone to run-time errors (even without typecasting)
  - eg.

```
1  Integer[] intArray = new Integer[2] {
2    new Integer(10), new Integer(20)
3  };
4  Object[] objArray;
5  objArray = intArray; → allowed since array is covariant.
6  objArray[0] = "Hello!"; // <- compiles!
```

Compiler thx Object[] refers to Integer[]

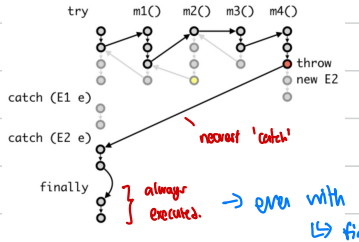
Exception

- Throwing Exceptions. (using try-catch-finally)
  - keyword 'throw' 'new'
  - Checked vs unchecked exception.

"Checked" for exceptions at Compile time.
- deal w at CT.
  - try-catch block
  - throws
- no control. (even with perfect code) eg. FileNotFound Exception.
  - must be handled else won't compile
- programmer's error eg. NullPointer Exception
  - not explicitly caught/thrown.
  - version of RuntimeException.
    - can do throw new RuntimeException();
- no compilation error
  - should use try-catch block.

  - Parsing the error:

    propagate automatically down the stack until either caught or throws an error.
  - Control Flow of Exceptions



→ even with return in the try block.
↳ finally's return (if have) will override.

  - Can create own exception by 'inheriting' from existing ones.
    - for additional useful information to the exception handler.
  - overriding method that Throw Exceptions.
    - must follow LSP → throw only the same, or a more specific checked exception than the overriden method.
    - The caller of the overriden method cannot expect any new checked exception beyond what has already been 'promised' in the method specification.

— Good Exceptions

- Catch exception to clean up (using finally)
  - if exception not handled, code that deallocates the resource is not called.
- Do not catch all vanilla Exception.
  - unable to figure out if something went wrong
- Do not exit program when caught exception.
  - prevent calling function from cleaning up the resource.
- Do not break abstraction barrier.
  - handle the implementation-specific exceptions within the abstraction barrier.
- Do not use it as a control flow mechanism
  - meant to handle unexpected errors not logic of program.
  - less efficient, might not be correct as exception may be triggered by something else in the program.

— Error class

- program should terminate as there is no way to recover from these errors.
- eg. StackOverflowError.
- no need to create/handle such error.

— Error and Exception are children of Throwable.

Generics Types (invariant) → Complex types

- takes other types as type parameters.
- use "< and >" to declare generic types.
- Cannot mix up the types
- To use: pass in type arguments — non-generic / generic / another type parameter that has been declared.
- once instantiated → parameterised type.
- only reference types can be used as type arguments.

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or `instanceof` With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Static need to have <T> instead or not?

Generic Methods

- type parameter T is declared within < >, added before the return type of the method.
- T is then scoped within the whole method.
- eg. A.<String> convert(strArray, 123);
- normally static methods.

- either GenericExample.<String>shout("John")
- or shout("John")

Bounded Type Parameters.

- use extends
- T must be a subtype of S.
- normally declared at the introduction of the generic type.

- can have multiple bounds, class must be specified first, only one class, can extends multiple interfaces.

↳ use &.

Type Erasure. (for generics)

- Java takes a code sharing approach, instead of creating a new type for every instantiation, it chooses to erase the type parameters and type arguments during compilation. (after type checking)
- There is only one representation of the generic type in the generated code, representing all the instantiated generic types, regardless of the type arguments
- For backwards compatibility.

```
class Pair<S,T> {
  private S first;
  private T second;

  public Pair(S first, T second) {
      this.first = first;
      this.second = second;
  }

  S getFirst() {
      return this.first;
  }

  T getSecond() {
      return this.second;
  }
}
```

→

```
class Pair {
  private Object first;
  private Object second;

  public Pair(Object first, Object second) {
      this.first = first;
      this.second = second;
  }

  Object getFirst() {
      return this.first;
  }

  Object getSecond() {
      return this.second;
  }
}
```

If type is bounded, then it is replace by the bound instead.

Where a generic type is instantiated and used, the code

```
Integer i = new Pair<String,Integer>("hello", 4).getSecond();
```

is transformed into

```
Integer i = (Integer) new Pair("hello", 4).getSecond();
```

↳ ensures casting will not lead to ClassCastException during run-time.

- Generics and Arrays Can't Mix.

```
// create a new array of pairs
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2];

// pass around the array of pairs as an array of object
Object[] objArray = pairArray;

// put a pair into the array -- no ArrayStoreException!
objArray[0] = new Pair<Double,Boolean>(3.14, true);
```

↳ type checking is done during RT.

Heap pollution.

→

```
// create a new array of pairs
Pair[] pairArray = new Pair[2];

// pass around the array of pairs as an array of object
Object[] objArray = pairArray;

// put a pair into the array -- no ArrayStoreException!
objArray[0] = new Pair(3.14, true);
```

due to type erasure, no information about the type argument.

- Causes heap pollution (variable of a parameterized type refers to an object that is not of parameterized type)
→ ClassCastException

```
// getting back a string?  -- now we get ClassCastException
String str = pairArray[0].getFirst();
```

- Java array is reifiable (a type where full type info is available during run-time)
  ↳ can check when we store into the array and maintain the type of the array ⇒ throws an ArrayStoreException if mismatch.
- Java generic is not reifiable due to type erasure.

Unchecked Warnings

- A message from the compiler that it has done what it can, because of type erasures → a runtime error that it cannot prevent.
- compiler can't guarantee the code is safe anymore.
- @SuppressWarning
  - use in the most limited scope to avoid unintentionally suppressing warnings that are valid concerns from the compiler.
  - only if we are sure it will not cause a type error later.
  - must always add a note to explain why a warning can be safely suppressed
  - cannot apply to assignment but only to declaration.

Raw types.

- generic type used without type arguments
- what code will look like after type erasure
- missing raw with parameterised type will lead to an error.
- Never used in code.
  - unless as an operand of the `instanceof` operator (since instanceof checks for RT type and type arguments have been erased).

Wildcards

- Upper-bounded (covariance)
  - If $S <: T$
  - then $A<? \text{ extends } S> <: A<? \text{ extends } T>$ ('anything below' T & T.)
  - note: if $K <: S$ then $A<K> <: A<? \text{ extends } S>$.
  - also $A<S> <: A<? \text{ extends } S>$
- Lower-bounded (contravariance)
  - If $S <: T$
  - then $A<? \text{ super } T> <: A<? \text{ super } S>$ ('anything above' S & S.)
  - also $A<S> <: A<? \text{ super } S>$.
- PECS

  Producer Extends; Consumer Super
  - Depends on the role of the variable
    - If the variable is a producer that returns T → declared as ? extends T.
    - If the variable is a consumer that accepts T → declared as ? super T.
- Unbounded
  - eg. $Array<T> <: Array<?>$ for every T.
  - Object is the supertype of all reference types
    - We want to write a method that takes in a reference type, but we want the method to be flexible enough → make method to accept a parameter of type Object.
  - Useful when want to write a method that takes in an array of some specific type, and the method to be flexible enough to take in an array of any type.

Generally do not use wildcards as return types.

- 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.

- Using ?, we can remove exceptions for raw types
  1. a instanceof A<?> — unknown/erased type parameter
  2. new Comparable<?>[10].

  ↓ reifiable since no type info is lost.

It's important to note that `List<Object>` and `List<?>` are not the same. You can insert an `Object`, or any subtype of `Object`, into a `List<Object>`. But you can only insert `null` into a `List<?>`

Type Inference

- Java will look among the matching type that would lead to successful type checks, and pick the most specific one.

- eg.

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

- find all possibilities of types, get the most specific one.

eg.

```
Shape o = A.findLargest(new Array<Circle>(0));
```

We have a few more constraints to check:

- Due to target typing, the returning type of `T` must be a subtype of `Shape` (including `Shape`)
- Due to the bound of the type parameter, `T` must be a subtype of `GetAreable` (including `GetAreable`)
- `Array<Circle>` must be a subtype of `Array<? extends T>`, so `T` must be a supertype of `Circle` (including `Circle`)

Intersecting all these possibilities, only two possibilities emerge: `Shape` and `Circle`. The most specific one is `Circle`, so the call above is equivalent to:

```
Shape o = A.<Circle>findLargest(new Array<Circle>(0));
```