CS2030S Notes

Assign	
Progress	In Progress
Series	
Tags	Computer Science NUS
Type	Notes

Unit 1: Program and Compiler

Software program: collection of data variables and instructions on how to modify these instructions

Programming language: formal language to help programmers specify *precisely* what are at a higher level of *abstraction* so that they only need to write a few lines of code to give complex instructions to the computer

Compiler: software tool that reads in the entire program written in a higher-level programming language and translates it into machine code; machine code is then saved into an executable file which is executed later

- *E.g.* `clang` — C/C++ compiler
- **Parsing source code:** checks that the syntax is accurate and produces *syntax errors* if any syntax is violated
- **Limitations:** cannot tell if particular statement will every be executed or values of variable during *compile time*
  - **Conservative:** report an error if there is a possibility that a statement might be incorrect (Rust)
  - **Permissive:** report an error only if there is no possibility that a statement might be correct

Interpreter: software tool that reads in the program one statement at a time, interprets what the statement means, and executes it directly

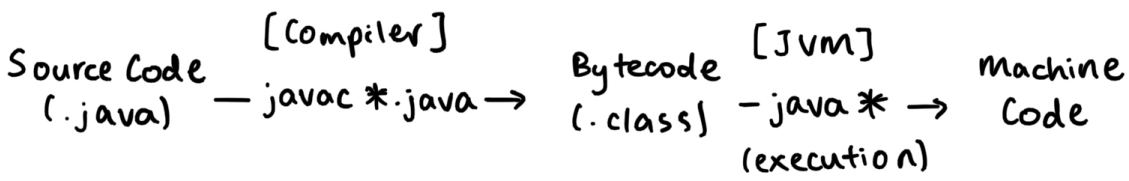
- *E.g.* Python and Javascript

Just-in-time compiler: reads bytecode and generates machine code dynamically at runtime with optimized performance

- *E.g.* V8 by Chrome

Java program execution:

1. Traditional



- Compile with `javac *.java` which generates a `.class` file
- Execute with `java *` where `*` is the file name without `.class` extension

2. `jshell`
    - Read-Eval-Print Loop (REPL) for Java
    - Performs compilation and run-time in the same step — creates ambiguity where and when errors occur

Compile time error: errors detected during compilation (before running the source code) where the program is still under development — favoured

Runtime error: errors that occur when running the source code where the system might already be live — less desirable

Unit 2: Variable and Type

Variable: abstraction that provides a *user-friendly name* to a piece of data in memory (since location in memory might change)

- **Pointer:** reference the address of the location that the variable is referring to

Type: data type of the variable; informing compiler/interpreter what operations are valid on this variable and how the operations behave

Dynamically typed: same variable can hold values of different types and type checking is performed during *runtime*

- *E.g.* Python and Javascript

```
x = 5   # x is of type Int
x = "6" # ok, x is now of type String
```

Statically typed: variable can only hold values of the same type (assigned at compile time) and type checking is performed during compile time

- *E.g.* Java

```
int x;   // x has a compile time type of int
x = 5;   // ok
x = "6"; // wrong, cannot assign a String to an int variable
```

- **Compile time type:** type the variable is assigned with when declaring the variable and used by the compiler when parsing syntax and checking for type mismatches

Type system: set of rules that govern how types interact with each other

- No set definition of *strength* of typing
- **Strong typing:** (generally) enforces strict rules in its type system and ensures *type safety* by catching type errors during compile time (instead of runtime); including things like *type casting*
- **Weak typing:** (generally) permissive with *type checking*
- Programming language can be static and weakly typed (C) or static and strongly typed (Java)

```
int i;
i = 4;
i = (int) "5"; // not permitted in Java
```

Subtyping: given that S and T are types, T is a *subtype* of S if a piece of code written for variables of type S can also be safely used on variables of type T

- Denoted by $T <: S$ (T is a subtype of S and S is the *supertype* of T)
- **Transitive:** $S <: T$ and $T <: U$ implies that $S <: U$
- **Reflexive:** for any type S, $S <: S$

Types in Java:

1. **Primitive type:** types that hold numeric and boolean values

Primitive Type	Stores
`byte`	8-bit signed integer
`short`	16-bit signed integer
`int`	32-bit signed integer
`long`	64-bit signed integer
`char`	16-bit unsigned integer representing UTF-16 Unicode character
`float`	16-bit floating-point number
`double`	32-bit floating-point number
`boolean`	`true` or `false`

   - **Subtyping:** $byte <: short <: int <: long <: float <: double$ and $char <: int$(used during type checking)
   - **Widening type conversion:** allowing a variable of type T to hold a value from a variable of type S only if $S <: T$

     ```
     double d = 5.0;
     int i = 5;
     d = i; // ok, double is the supertype of int so an int can be "fit" into a double
     i = d; // error, int cannot "fit" a double
     ```

2. **Reference type** (discussed later on)

Unit 3: Functions

Function: (or *procedure*) abstraction that groups a set of instructions and gives them a name

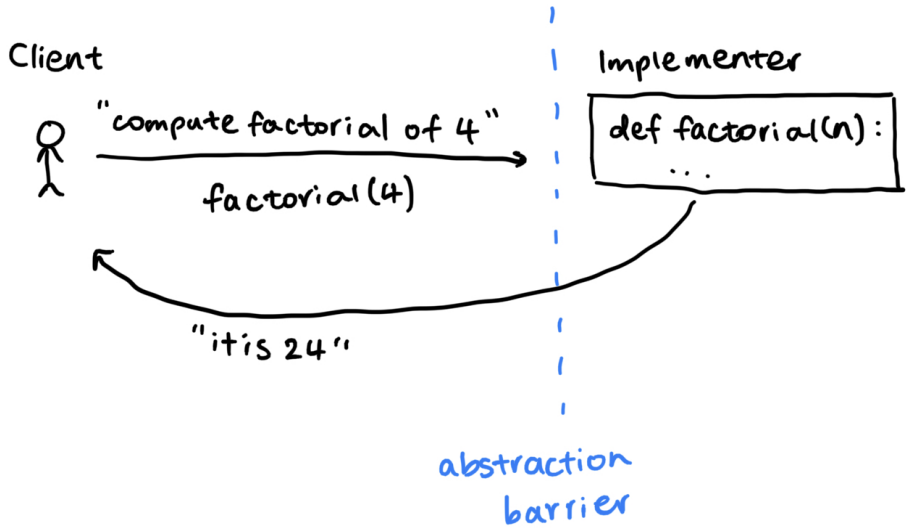
- Takes one or more variables as *input parameters*
- Return one or more values

Benefits of functions:

1. Compartmentalise computation and its effects
2. Hide the implementation of a task

    a. Minimises changes necessary as software evolves
3. Reduce repetition through *code reuse*
    a. Creates succinct code
    b. Minimises changes necessary as software evolves; minimises chance to introduce new bugs

Abstraction barrier: separates the role of the programmer into two: (a) implementer and (b) client



1. **Implementer:** provides implementation of abstraction
2. **Client:** uses abstraction to perform task

- Enforces *separation of concerns* between roles

Unit 4: Encapsulation

Composite data type: group *primitive types* together using a name

- *E.g.* `struct` in Elixir

Class: bundled composite data type and its associated functions; maintains abstraction barrier and exposes the right *method interface* for others to use

- **Methods:** functions in a class
- **Fields:** (or *members, states, attributes*) data in the class

Encapsulation: keeping all the data and functions operating on the data related to a composite data type together within an abstraction barrier

```
class Circle {
  double x; // field
  double y; // field
  double r; // field

  // method
  double getArea() {
    return 3.1415926 * this.r * this.r;
  }
}
```

Object: *instance* of a class, containing the same set of fields and methods (but with differing values)

- Created with `new` keyword in Java: `Circle c = new Circle()`
- Fields and methods access with `.`: `c.getArea()`

Object-oriented programming: instantiate objects of different classes and orchestrating their interactions with each other by calling each other's methods

- **Classes/objects:** modelled after *nouns*
- **Fields:** properties and relationships among classes
- **Methods:** *verbs/actions* of the corresponding objects

Reference type: everything else that is not a primitive type in Java

- Variables store only the reference to the value (in memory), so two reference variables can share the same value

```
Circle c1 = new Circle();
Circle c2 = c1; // any changes to c2 affects c1 and vice versa
```

`null`: any reference variable not initialised

- Leads to `NullPointerException` if attempting to access a `null` value

Unit 5: Information Hiding

Data hiding: (or *information hiding*) protecting the abstraction barrier from being broken by explicitly specifying if a field/method can be accessed from outside the abstraction barrier

- Enforced by the *compiler* at compile time
- *E.g.* `private` fields/methods cannot be accessed from outside the class, only within while `public` fields/methods can be accessed/modified outside of the class

Constructor: method that initialises an object

- Cannot be called directly but invoked automatically when an object is instantiated
- Same name as the class and no *return type*
- Takes in *arguments* like other methods

```
class Circle {
  private double x; // private field cannot be accessed outside of Circle
  private double y;
  private double r;

  // Constructor
  public Circle(double x, double y, double r) {
    this.x = x;
    this.y = y;
    this.r = r;
  }

  public double getArea() {
    return 3.1415926 * this.r * this.r;
  }
}
```

```
Circle c = new Circle(0.0, 0.5, 10.0);
```

`this`**:** reference variable that refers back to self; used to distinguish between two variables of the same name

- Idiomatic to use `this` to refer to members of a class — explicitly referring to a field in the class

Unit 6: Tell, Don't Ask

Accessor: (or *getter*) method to retrieve the property of an object

Mutator: (or *setter*) method to modify the properties of the object

"**Tell, Don't Ask" principle:** instead of retrieving each field using accessors and performing the operations on the client side, tell the object what task to perform and implement this task within the class as a method

- Maintains encapsulation and avoids unnecessary accessors and mutators

Unit 7: Class Fields

Class fields: associating global values and functions with a *class* instead of with an *object*

- `static`**:** declares a field/method to be associated with a class
- Useful for storing pre-computed values or configuration parameters associated with a class (rather than individual objects)
- Accessed through *class name* without instantiating class

```
class Math {
  public static final double PI = 3.141592653589793;
}

Math.PI; // not new Math().PI
```

`final`**:** indicates that the value of the field will not change

`import`**:** imports external library and shortens reference type when declaring variables

```
import java.lang.Math;
Math.PI; // equivalent to java.lang.Math.PI
```

Unit 8: Class Methods

Class methods: associating a method with a *class*, not an *object* of the class

- Invoked without an instance so no access to the instance's fields or methods
- `this` has no meaning
- Accessed through *class name*

`main` **method:** entry point to the program

```
public class Hello {
  // final is optional
  public final static void main(String[] args) {}
}

java Hello
```

- `void`: `main` does not return any value

Unit 9: Composition

Composition: build up layers of abstractions and construct sophisticated classes, modelling a HAS-A relationship between two entities

- *E.g.* A `Circle` HAS-A `Point` as the center and a `Cylinder` HAS-A `Circle` as the base

Aliasing: referencing types that share the same reference value, creating unintended side effects

- When passing a reference type between methods, the reference is passed, not a cloned copy of the type
- **Solutions:**
  - **Avoid sharing references:** implicates proliferation of objects and lack of optimisation for computational resource usage
  - **Immutability** (discussed later on)

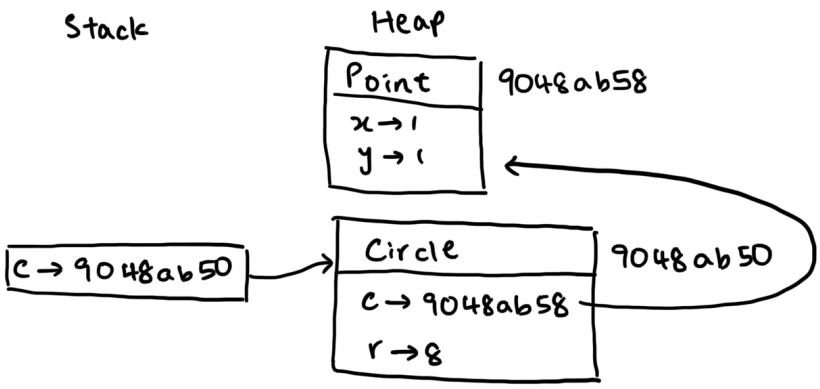
Unit 10: Heap and Stack

JVM memory partitions:

- **Method area:** storing the code for the methods
- **Metaspace:** storing meta information about classes
- **Heap:** storing dynamically allocated objects (common to all execution environments)
  - Memory kept as long as there is a reference to the object on the heap
- **Stack:** storing local variables (including primitive types and object references) and call frames (common to all execution environments)

Object initialisation:

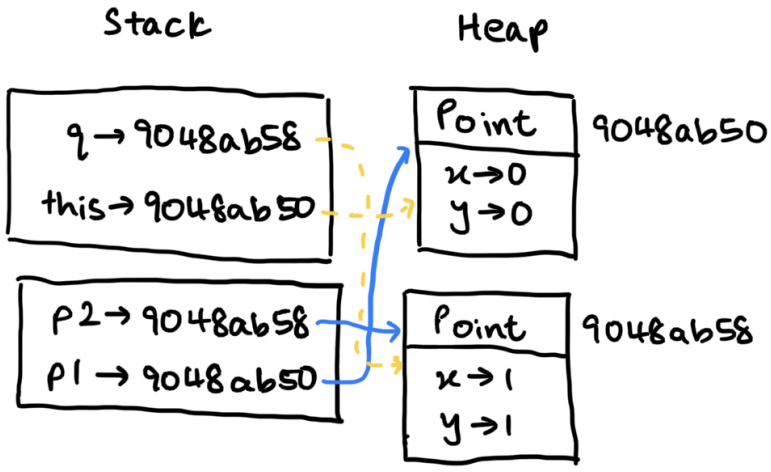
```
Circle c;
c = new Circle(new Point(1, 1), 8);
```



Method invocation:

- **Stack frame:** region of memory created when calling a method (class or instance) that tentatively contains (a) `this` reference, (b) method arguments, (c) local variables within the method, etc.
  - `this` reference not contained in *class method* call
  - Destroyed after method returns

```
Point p1 = new Point(0, 0);
Point p2 = new Point(1, 1);
p1.distanceTo(p2); // creates stack frame for internal method implementation
```



Call by value: primitive type values are copied onto the stack, not referenced (like reference types)

- **Wrapper class:** replicate call by reference by wrapping primitive type

Call by reference: reference types are referenced to the heap

Garbage collector: checks for unreferenced objects on the heap and cleans up the memory automatically

Unit 11: Inheritance

Inheritance: modelling the IS-A relationship between two entities

- Ensure that inheritance *preserves* the meaning of subtyping
- Avoid misusing inheritance when *composition* is better

`extends`**:** creates a subtype relationship between classes

- **Parent class:** (or *superclass*) is the base class that is being extended
- **Child class:** (or *subclass*) is the class that is extending the *parent class*
- $\text{subclass} <: \text{superclass}$
- All public fields/methods of superclass is accessible to the subclass (inherited)
- Any private fields/methods of superclass is not accessible to the subclass — creating abstraction barrier

```
import java.lang.Math;

class Circle {
  private Point center;  // Inaccessible to ColoredCircle
  private double radius; // Inaccessible to ColoredCircle

  public Circle(Point center, double radius) {
    this.center = center;
    this.radius = radius;
  }

  // Accessible to ColoredCircle
  public double getArea() {
    return Math.PI * this.radius * this.radius;
  }
}

class ColoredCircle extends Circle {
  private Color color;

  public ColoredCircle(Point center, double radius, Color color) {
    super(center, radius);
    this.color = color;
  }
}
```

`super`**:** reference the superclass instance

- Used to call superclass constructor or public fields/methods

Runtime type: type assignment that occurs during runtime when compile time type differs from the actual assigned type of the variable

- Only possible with subtypes

```
Circle c = new ColoredCircle(p, 0, blue); // Circle is the compile time type, ColoredCircle is the runtime type
```

Unit 12: Overriding

`Object` **class: "**ancestor" of all classes in Java and is at the root of the class hierarchy

- Every class that does not extend another inherits from `Object`
- Contains methods common to all objects

`Object::toString` **method:** invoked *implicitly* by Java to convert a reference object to a `String` object during string concatenation using `+`

- Don't need to explicitly call `"Circle c is " + c.toString()`
- **Default implementation:** (from `Object`) returns the reference to the object (similar to the ones in the *Heap*)

Method signature: method name and the number, type, and order of its parameters

Method descriptor: method signature and the return type

Method overriding: altering the behaviour of an existing class (in the subclass) by defining an instance method with the same *method descriptor* as an instance method in the superclass

- Must include the same parameter types (subtypes are not allowed)

`@Override` **annotation:** *hint* to the compiler that the target method is intended to override the method in the superclass

- Allows the compiler to check for any typos or if overriding is not possible
- `super` will still call the superclass's implementation of the overriden method, if necessary

```
class Circle {
  @Override
  public String toString() { ... }
}
```

Unit 13: Overloading

Method overloading: having to or more methods in the same class with the same name but a differing *method signature*, i.e. changing the number, type, and order of the parameters

- Possible to overload *class methods* as well

Unit 14: Polymorphism

Polymorphism: change how existing code behaves without changing a line of the existing code (or even having access to the code), allowing it to take many forms

- Allows for future proof and succinct code
- New classes can be added without needing to re-compile existing code

`Object::equals` **method:** compares if two object references refer to the same object by default

- When overriden, can be used to check if two objects are *semantically* the same, i.e. same field values
  1. Check that the runtime type of both objects are the same using `instanceof` operator
  2. Once assured, cast/assign the object from `Object` to the target type through *type casting*
  3. Compare the fields accordingly and return if equal or not

Type casting: type casting from type T to S if $S <: T$ (aka *narrowing type conversion*)

- Requires explicit typecasting and validation during runtime
- If runtime type of target is not the same as the cast type, then runtime error will occur

Unit 15: Method Invocation

Dynamic binding: (or *late binding,* or *dynamic dispatch*) determines which method implementation a method invocation is bound to

```
Object c1 = new Circle(new Point(1, 1), 5);
Object c2 = new Circle(new Point(2, 2), 5);
c1.equals(c2); // how does the compiler know to use Circle#equals, not Object#equals
```

1. **Compile time:** search for the most specific *method signature* and store the corresponding *method descriptor*
    a. Determine the *method descriptor* of the method invoked using the compile time type of the target, i.e. `c1` has a compile time type of `Object`
        - Compiler searches for all methods that can be correctly invoked using the given argument based on *method signature*
        - If multiple are found, choose the *most specific* one, determined by the method which can receive the given arguments without compilation errors
            - *E.g.* `equals(Circle)` is more specific than `equals(Object)`

            ```
            boolean equals(Circle c) { ... }

            @Override
            boolean equals(Object c) { ... }
            ```

    b. Store found method's descriptor in generated bytecode
        - Does not include information about the class implementing this method
2. **Runtime**
    a. Retrieve the method descriptor from the bytecode
    b. Determine runtime type of the target
    c. Look for an accessible method with the matching description of the runtime type
        - If no such method is found, search continues up the class hierarchy until `Object`
    d. First method implementation with a matching method descriptor found will be executed

Support for class methods: dynamic binding not supported as method to invoke is resolved statically during compile time (without consideration of the runtime type)

Unit 16: Liskov Substitution Principle

Liskov Substitution Principle (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 $S <: T$

- Formal definition of subtyping $S <: T$
- Substituting a superclass with a subclass should not break the program (in terms of predictable behaviour), i.e. a *subclass* should not break the expectations set by the *superclass*, otherwise, it violates the LSP

- **Enforcement:** cannot be done by the compiler; managed and agreed upon among programmers through documentation
- Applied in testing where test cases are designed based on the *specification* of the method, not implementation

Preventing inheritance and overriding: explicitly prevent a class from being inherited or a method from being overriden using `final`

Unit 17: Abstract Class

Abstract class: class that has been made into something so general that it cannot and should not be instantiated

- Usually means one or more of its instance methods cannot be implemented without further details
- Declared using the `abstract` keyword in the class declaration: `abstract class Shape`
- Models an IS-A relationship as concrete classes are subclasses of the abstract class

```
abstract class Shape {
  abstract public double getArea();
}

class Circle extends Shape {
  @Override
  public double getArea() { ... }
}
```

Abstract method: cannot be implemented and should not have a method body

- Declared using the `abstract` keyword in the method declaration: `abstract public double getArea();`
- Not all methods must be abstract but once one is abstract, the class must be abstract
- Any subclasses of an abstract class must override the abstract methods

Concrete class: class that is not abstract, i.e. no abstract methods

Unit 18: Interface

Interface: type that models what an entity can do and is declared with the `interface` keyword

- Has `-able` suffix, e.g. `GetAreable`
- All methods declared as `public abstract` by default

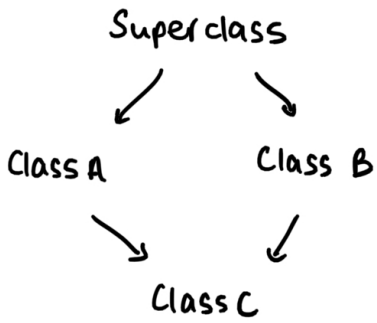
```
interface GetAreable {
  double getArea();
}

class Shape implements GetAreable {
  @Override
  public double getArea() { ... }
}
```

Implementing an interface and being concrete: must override all abstract methods from the interface and provide an implementation to each

- Otherwise, the class is abstract
- Implement using `implements` keyword

Limitations on `extends` and `implements`: a class can only extend from one superclass, but can implement multiple interfaces

- **Diamond problem:** diamond-shaped class diagram where there is ambiguity for the compiler to decide which superclass method to execute when calling from the superclass



  - Problem is avoided with multiple interfaces as the implementation of the same method across interfaces will be used across all interfaces so no ambiguity
- An interface can extend from one or more other interfaces, but cannot extend from another class

Interface as supertype: if a class C implements an interface I, $C <: I$

- Implies that a type can have multiple supertypes (but not superclasses)
- *E.g.* $\text{Flat} <: \text{GetAreable}$ and $\text{Flat} <: \text{RealEstate}$

`default` **implementations:** resolve breaking changes caused when adding new methods to an interface which may otherwise cause compilation errors (as the subclass won't override the abstract methods)

- Acts like an abstract class
- Referred to as *impure interfaces*

Unit 19: Wrapper Class

Wrapper class: class that encapsulates a *type*, rather than fields and methods

- *E.g.* `Integer` for `int`, `Double` for `double`, etc.
- Behaves like any other class
- Reference types and created with `new` keyword
  - Useful when trying to superclass with `Object` since primitive types do not subclass `Object`
- Instances stored on the *heap*
- All primitive wrapper class objects are *immutable*, i.e. cannot be changed after creation

Auto-boxing and unboxing: performs type conversion between primitive type and its wrapper class

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

Added performance costs: using an object involves allocating memory for the object and garbage collection which is less efficient than primitive types (auto-boxing and unboxing does not eliminate this)

- Due to immutability, new objects of each primitive wrapper class is created every mutating operation

Unit 20: Runtime Class Mismatch

Cast carefully: typecasting asks the compiler to trust that the object returned by a method has a target runtime type (this needs to be proven and assured)

- Happens when performing *narrowing type conversion*
- Can lead to fragile code as the correctness is dependent on the runtime type, and the compiler cannot help catch any errors with this
- If the runtime type is mismatched and not properly casted, an error can occur

```
GetAreable findLargest(GetAreable[] arr) { ... }
GetAreable ga = findLargest(arr);      // ok
Circle c1 = findLargest(arr);          // error, Circle <: GetAreable so need to be cast
Circle c2 = (Circle) findLargest(arr); // ok
```

Unit 21: Variance

Complex types: types that are composed of simpler types (primitive or reference) such as arrays

Variance of types: how the subtype relationship between complex types relates to the subtype relationship between components (of the complex type)

- Let $C(S)$ correspond to some complex type based on type S
  - **Covariant:** if $S <: T$ implies $C(S) <: C(T)$
  - **Contravariant:** if $S <: T$ implies $C(T) <: C(S)$
  - **Invariant:** neither covariant or contravariant

Covariant arrays: arrays are covariant in Java, so if $S <: T$, then $S[] <: T[]$

```
Integer[] intArr;
Object[] objArr;
objArr = intArr; // ok, widening type conversion
```

- Opens up the possibility of runtime errors

```
Integer[] intArray = new Integer[2] {
  new Integer(10), new Integer(20)
};
Object[] objArray;
objArray = intArray;
objArray[0] = "Hi!"; // ok, compile time type is Object which allows String (String <: Object)
```

- Above compiles as $\text{String} <: \text{Object}$ and $\text{Integer} <: \text{Object}$
- Compiler does not know that `objArray` is referring to an `Integer[]` until runtime

Unit 22: Exception

Pitfalls of error value handling: (like in Go or C)

1. Need to check different error values and react accordingly — verbose code
2. (Maybe) global error value prone to unintended mutations
3. Clean up after every error value — easy to forget or mess up and creates repeated code

Exceptions: programming construct that provides a cleaner way of checking and handling errors to improve code readability

`try` - `catch` - `finally`: error handling construct in Java

```
try {
    // do something
} catch (an exception parameter) {
    // handle exception
} finally {
    // clean up code
    // regardless of there is an exception or not
}
```

- `try` **body:** allows the code to execute all at once and handle the exceptions later on in `catch`
- `catch` **clauses:** handles a different type of exception with information about the exception encapsulated in the exception instance, `catch (FileNotFoundException e)`
  - Handle specific exceptions first before more general exceptions
- `finally` **clause:** handles house-keeping tasks like closing any open streams
  - Always executed even when return or throw is called in a `catch` block
- **Avoiding repeated error handling:** `catch (FileNotFoundException | InputMismatchException e)`

`Exception` **superclass:** superclass all exceptions, compile time or runtime

Throwing exceptions: declare that a construct is throwing an exception with `throws` and throw an exception to the caller with `throw`

- `throw` causes immediate return

```
public Circle(Point c, double r) throws IllegalArgumentException {
    if (r < 0) {
      throw new IllegalArgumentException("radius cannot be negative.");
    }
    this.c = c;
    this.r = r;
  }
}

try {
    c = new Circle(point, radius);
} catch (IllegalArgumentException e) {
    System.err.println("Illegal arguement:" + e.getMessage());
}
```

Unchecked exception: exception caused by a programmer's errors that should not happen if perfect code is written

- *E.g.* `IllegalArgumentException`, `NullPointerException`, and `ClassCastException`
- Typically not explicitly caught or thrown
- Indicate that something is wrong with the program and causing runtime errors
- $\text{Unchecked exception} <: \text{RuntimeException}$

Checked exception: exception that a programmer has no control over even if perfect code is written

- *E.g.* `FileNotFoundException`
- Actively anticipate and handle these exceptions using `try`-`catch`-`finally`

Exception propagation: if an exception is not caught in its immediate caller, it can be passed to its caller and so on, propagating down the call stack until its either caught or results in an error state (for unchecked exceptions)

- Checked exceptions must be caught otherwise the program will not compile, unless `throws` is used to pass this exception to a better place to handle
  - If left to propagate till the end, will result in an error state

Graceful checked exception handling: a good program always handle checked exception gracefully and hide the details of the exception from users

Custom exceptions: if none of the existing exceptions meet the requirements, then create a custom exception by inheriting from an existing one

- Only to be done with good reason such as providing more information to the exception handler
- Override `toString` method to provide custom error message

Overriding methods with thrown exceptions: when overriding a method that throws a checked exception, overriding method must throw only the same, or a more specific checked exception, than the overriden method

- Inline with the LSP — caller of overriden method cannot except any new checked exceptions beyond what was already "promised"

Exception handling good practices:

1. **Catch exceptions to clean up:** if a method allocates system resources (e.g. temporary files, network connections) and cleans up these resources at the end, an exception in between can cause the system to fail to deallocate the resources and leave these system resources allocated
   - Best to catch the exception and deallocate in the `finally` block
   - Re-`throw` the exception if the callers need to know of it
2. **Do not catch them all:** do not use an umbrella "catch-all" such as `catch (Exception e)` as this will silently ignore all exceptions — checked or unchecked
3. **Overreacting:** do not exit a program (silently) because of an exception as that can cause the problems discussed in (1)
4. **Do not break abstraction barrier:** handle implementation-specific exceptions within the abstraction barrier to avoid leaking details about implementation to the client

- *E.g.* throwing a `FileNotFoundException` would indicate to the client that the implementation is reading from a file at some point

5. **Do not use exception as a control flow mechanism:** do not use exceptions to replace what proper control statements like `if-else` is built to handle

`Error` **class:** indicates that the program should terminate as there is no way to recover from an error

- *E.g.* when the heap is full (`OutOfMemoryError`) or the stack is full (`StackOverflowError`)
- Typically do not create or handle these errors
- Be careful when phrasing to not use "Error" in place of "Exception"

Unit 23: Generics

Generics: mechanism to abstract the type of a parameter by defining a *generic type* that takes other types as *type parameters* (like how methods take in variables as parameters)

Declaring a generic type: declare a generic type by specifying its type parameters between `<` and `>` when declaring the type

- Conventionally use a single capital letter to name each type parameter
- Type parameters are scoped within the definition of the type

```
class Pair<S,T> {
  // S and T are type parameters
  private S first;
  private T second;

  // Constructor still declared as Pair without the type parameters
  public Pair(S first, T second) {
      this.first = first;
      this.second = second;
  }

  S getFirst() {
      return this.first;
  }

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

Using/instantiating a generic type: pass in *type arguments* (which can be non-generic or generic or another type parameter declared)

- **Parameterised type:** instantiated generic type
- Must use primitive wrapper classes as only reference types can be used as type arguments

$$\overbrace{\text{Pair}<\underbrace{S,T}_{\text{type parameters}}>}^{\text{generic type}} \qquad \overbrace{\text{Pair}<\underbrace{\text{String, Integer}}_{\text{type arguments}}> = \text{new Pair}<>(\text{“string”}, 15);}^{\text{parameterized type}}$$

Generic method: parameterise a method with type parameters without being in a generic class

```
public <T> boolean contains(T[] arr, T target) { ... }
```

- Type parameter must be declared before the return type (can use the type parameter as the return type)
- `T` is scoped within the whole method
- **Calling a generic method:** `A.<String>contains(strArry, "123")` ; ensure that the type argument is placed before the method name

Bounded type parameters: constrain the type parameter to only accept specific types based on their class hierarchy using `extends`

```
public static <T extends GetAreable> T findLargest(T[] arr) { ... }
```

- T must be a subtype of $\mathrm{GetAreable}$, $T <: \mathrm{GetAreable}$

Unit 24: Type Erasure

Code specialisation: generating new code for the instantiated generic type during runtime (C#) or compile time (C++ or Rust)

Code sharing: instead of creating a new type for every instantiation, *erase* the type parameters and arguments during compilation (after type checking)

- Results in only one representation of the generic type in the generated code, representing all the instantiated generic types, regardless of the type arguments
- Motivated by *backward compatibility*
- Unbound type parameters are replaced with `Object`
- Bound type parameters are replaced with the bounds

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

Heap pollution: situation where a variable of a parameterised type refers to an object that is not of that parameterised type, leading to `ClassCastException` when retrieved

```
// Hypothetical - compile time
Pair<String, Integer>[] pairArr = new Pair<String, Integer>[2];
Object[] objArr = pairArr;
objArr[0] = new Pair<Double, Boolean>(3.14, true);

// Runtime
Pair[] pairArr = new Pair[2];
Object[] objArr = pairArr; // allowed as arrays are covariant
objArr[0] = new Pair(3.14, true); // due to type erasure

String str = pairArr[0].getFirst(); // error, type checking done at runtime
```

Reifiable type: type where full type information is available during runtime

- *E.g.* Java arrays so the Java runtime can check what is stored in the array and whether it matches the type of the array

Generics vs reifiable types: generics are not reifiable due to type erasure and so this type information is missing during runtime

Unit 25: Unchecked Warnings

`ArrayList` **class:** provides similar functionality to an array with some performance overhead

- Imported using `java.util.ArrayList`
- Used to get around problem of mixing arrays and generics
- Implements an array internally

```
class Array<T> {
  private T[] array;

  Array(int size) {
    // this.array = (T[]) new Object[size]; // not possible
    // Instantiate temporary array first
    T[] temp = (T[]) new Object[size]; // raises unchecked warning
    // Assign to class variable
    this.array = temp;
  }
}
```

Invariance: generics are invariant so there is no subtyping relationship between `ArrayList<Object>` and `ArrayList<Pair<String, Integer>>` (e.g.)

- Cannot alias one with the other so heap pollution is avoided

Unchecked warning: message from the compiler that because of type erasures, there could be an unpreventable runtime error

- Arises if the compiler is unsure if a type operation is safe, e.g. *heap pollution* example in unit 24 or in above example

`@SuppressWarning("unchecked")` **annotation:** suppresses warning messages from compilers and indicates to the compiler that the type operation is deemed and assured to be safe (by the programmer and code design)

- **Limited scoping:** annotation can be declared at different scopes (local variable, method, type, etc.) so should be used at the *most limited scope* to avoid unintentionally suppressing warnings that are valid concerns from the compiler
- **Certainty:** only suppress warnings if it can be assured that the operation will not cause a type error later
- **Documentation:** adding documentation about why the warning can be suppressed is important
- Can only be applied to declaration, not assignment

Raw types: generic type used without type arguments which is allowed by Java (for backward compatibility) but no type checking is done

```
Array a = new Array(4);
```

- Never use raw types in code unless using the `instanceof` operator since `instanceof` only works with raw types (addressed in unit 26)

Unit 26: Wildcards

Wildcard: can be substituted for any type (not bound to the declared type parameters)

Upper-bounded wildcard: wildcard that can be substituted with any subtype of a given type (including the bound type), e.g. `Array<? extends Shape>`

- If $S <: T$, then $A <? \text{ extends } S > <: A <? \text{ extends } T >$(covariance)
- For any type S, $A < S > <: A <? \text{ extends } S >$
- Used when more flexibility is required

Lower-bounded wildcard: wildcard that can be substituted with any supertype of a given type (including the bound type), e.g. `Array<? super T>`

- If $S <: T$, then $A <? \text{ super } T > <: A <? \text{ super } S >$(contravariance)
- For any type S, $A < S > <: A <? \text{ super } S >$
- Used when more strictness is required

PECS: (or *Producer Extends; Consumer Super*) rule that governs when to use upper-bounded wildcard and lower-bounded wildcard depending on the role of the variable

- If the variable is a *producer* that returns a variable of type T, then it should be declared as an upper-bounded wildcard
- If the variable is a *consumer* that accepts a variable of type T, then it should be declared as a lower-bounded wildcard

Unbound wildcards: can be substituted with any type, e.g. `Array<?>`

- Supertype of all possible T, i.e. $A < T > <: A <? >$
- **Differences from `Array<Object>`:** `Array<Object>` is not a supertype of all `Array<?>` due to generic invariance so cannot do `Array<Object> arr = new Array<String>(5);`
- **Quick reference:**
  - **`Array<?>`:** array of objects of some specific, but unknown type
  - **`Array<Object>`:** array of `Object` instances, with type checking by the compiler
  - **`Array`:** array of `Object` instances without type checking

Revisiting raw types: can be used in `instanceof` and instantiating generic arrays

- `a instanceof A<?>` works
- `new Comparable<?>[10]` works
  - Array creation requires reifiable type and `Comparable<?>` is reifiable since the type of `?` is not known so no information is lost

Unit 27: Type Inference

Type inference: looks for all possible types that match a given type parameter and picks the most specific one without considering any additional subclasses that are not specified in the type constraints

- Start with the type of the expression (target typing), then the arguments (in order) of the method, and then the return type

Diamond operator: remove the need declare the type arguments twice when instantiating a generic type

```
Pair<String, Integer> p = new Pair<>(); // type of second Pair is inferred
```

Target typing: type inference involving the type of the expression

```
public static <T extends GetAreable> T findLargest(Array<? extends T> arr)
Shape o = A.findLargest(new Array<Circle>(0));
```

- T must be a subtype of `Shape` (including `Shape`) from target typing in the expression
- T must be a subtype of `GetAreable` (including `GetAreable`) from the bound of the type parameter
- T must be a supertype of `Circle` (including `Circle`) from the upper-bounded wildcard

Unit 28: Immutability

Immutable class: instance of an immutable class cannot have any visible changes outside its abstraction barrier

- Every method call behaves the same way throughout the lifetime of the instance
- Reduces bugs when code complexity increases as change is avoided

Making a class immutable:

1. `private` and `final` fields to avoid mutation after initialisation in the constructor
2. `final` class to avoid subclasses of `Point` from overriding the methods
3. No setter methods or methods the modify field or objects referred to by fields
4. `private` constructors and factory methods to create new instances of the object
5. Do not allow fields that reference mutable objects to be changed through methods
    a. Don’t provide methods that modify the object
    b. Don’t share references to the mutable objects

```
final class Point {
  final private double x;
  final private double y;

  public Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  public Point moveTo(double x, double y) {
    return new Point(x, y);
  }
}
```

- By enforcing immutability, object references are not shared and as such there will not be any unintended changes even if these object references are passed around

Types of immutability:

1. Creating a new copy of the instance
2. Using immutable classes to share all references until modification to the instances are needed
    a. *Copy-on-write* avoids aliasing bugs without creating excessive copies of objects

Advantages of immutability:

1. **Ease of understanding:** as long as a variable is not re-assigned later on, any references to it will always be to the originally instantiated version and no changes will affect this original reference (since a new instance is always created instead)
2. **Enabling safe sharing of objects:** instances of the same class can be shared without worrying that the internals are going to be changed
    - Gives rise to the use of singletons such as

```
final class Point {
  // ...
  private Point(double x, double y) {
    this.x = x;
    this.y = y;
  }

  private final static Point ORIGIN = new Point(0, 0);

  public static Point of(double x, double y) {
    if (x == 0 && y == 0) {
      return ORIGIN;
    }
    return new Point(x, y);
  }
}
```

    - Only works with immutable classes as the fields won't be changed
3. **Enabling safe sharing of internals:** if the underlying fields will not mutate, they can be shared with other instances

```
class ImmutableArray<T> {
  private final int start;
  private final int end;
  private final T[] array;


  // T... => (var)iable (arg)uments akin to passing in an array of items
  // @SafeVarags => inform Java that the generic vararg used in this array
  //                is safe because every element of the array is T
  @SafeVarargs
  public static <T> ImmutableArray<T> of(T... items) {
    return new ImmutableArray<>(items, 0, items.length-1);
  }

  private ImmutableArray(T[] a, int start, int end) {
    this.start = start;
    this.end = end;
    this.array = a;
  }
```

```
  public T get(int index) {
    if (index < 0 || this.start + index > this.end) {
      throw new IllegalArgumentException("Index out of bound");
    }
    return this.array[this.start + index];
  }

  // The same `array` instance can be passed around because it is immutable
  public ImmutableArray<T> subarray(int start, int end) {
     return new ImmutableArray<>(
          this.array,
          this.start + start,
          this.start + end);
  }
}
```

4. **Enabling safe concurrent execution:** correctness ensured (more discussed later on)

Unit 29: Nested Classes

Nested class: class defined within another containing class; used to group logically relevant classes together

- Typically tightly coupled to the containing class and has no use outside of the container class
- Used to encapsulate information within a container class (see “Builder Pattern”) if the implementation of the container class becomes too complex (aka “helper” classes)
- Is a field of the containing class
- Can be declared as `private` so inaccessible outside of the container class
- Able to access fields and methods of the container class (even `private` fields)
  - Can potentially cause the container class to leak implementation details to the nested class if the nested class and container class do not belong to the same encapsulation as the container class
- *E.g.* `HashMap<K, V>` contains nested classes like `HashIterator` and `Entry<K, V>`
- **Static nested class:** associated with the containing class and not the instance
  - Can only access static fields and methods of the containing class

```
class Outer {
  static class Inner { }
}

// Creating a new instance
new Outer.Inner();
```

- **Non-static nested class (aka “inner class”):** can access all fields and methods of the containing class

```
class Outer {
  class Inner { }
}

// Creating new instance
new Outer().Inner();
```

- **Field access:**

```
class A {
  private int x;
  static int y;

  class B {
    void foo() {
      x = 1; // accessing x from A is OK
      y = 1; // accessing y from A is OK
    }
  }

  static class C {
    void bar() {
      x = 1; // accessing x from A is not OK since C is static
      y = 1; // accessing y is OK
    }
  }
}
```

- **Qualified `this` reference:** `<container>.this.<variable>` to reference container class `this` variables
  - `this` inside the nested class references the nested class itself

Local class: class declared inside a method (more specifically within a block of code between `{}`

- Cannot be declared as `static`
- Methods that are made `static` can only access containing class `static` variables (cannot access the containing method's variable as they cannot be declared as static)
- Makes code easier to read as the definition of the class is kept close to the usage of the class
- Scoped within the method with access to
  - container class through qualified `this` reference (can access `private` variables of the containing class)
  - local variables of the enclosing method

```
class A {
  int x = 1;

  void f() {
    int y = 1;

    class B {
      void g() {
        x = y; // accessing x and y is OK.
      }
    }

    new B().g();
  }
}

// Calling the method
new A().f();
```

- Can be used to declare `Comparator` without creating a new `.java` file for it

```
void sortNames(List<String> names) {

  class NameComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
      return s1.length() - s2.length();
    }
  }

  names.sort(new NameComparator());
}
```

Variable capture: local classes make a copy of local variables (of the enclosing method) inside itself

- Used because local variables of methods are removed from the stack when the method returns but the instance of the local class might still exist

```
interface C {
  void g();
}

class A {
  int x = 1;

  C f() {
    int y = 1;

    class B implements C {
      public void g() {  // must be public to override the interface
        x = y + 1;
      }
    }

    B b = new B();
    return b;
  }
}

public class VariableCapture {
  public static void main(String[] args) {
    A a = new A();
    C b = a.f();
    b.g();  // uses the captured value of y = 1
    System.out.println(a.x);  // prints 2
    b.g();  // uses the captured value of y = 1
    System.out.println(a.x);  // prints 2
  }
}
```

- Captured variables must be *effectively final*, i.e. explicitly declared `final` or one whose value does not change after initialisation (implicitly final)

```
void sortNames(List<String> names) {
  boolean ascendingOrder = true;
  class NameComparator implements Comparator<String> {
    public int compare(String s1, String s2) {
      // ascendingOrder cannot be accessed since it is mutable
      if (ascendingOrder)
        return s1.length() - s2.length();
      else
        return s2.length() - s1.length();
    }
  }

  ascendingOrder = false;
```

```
  names.sort(new NameComparator());
}
```

Anonymous class: declared and instantiated in a single statement without a name to reference later on

- **Format:** `new X (arguments) { body }`
  - `X` — class/interface to extend/implement; only 1 class/interface at a time
  - `arguments` — arguments to pass to constructor of anonymous class (interface does not have interface but need `()`)
  - `body` — body of the anonymous class; without a constructor
    - `static` variables must be `final`

```
Comparator<String> cmp = new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return s1.length() - s2.length();
  }
};
names.sort(cmp);
```

- Follows the rules of variable capture and access of local classes

Unit 30: Functions

Functions: mapping from a set of inputs (domain) X to a set of outputs (codomain) Y, i.e. $f : X \to Y$ with every input mapping to exactly 1 output (but the converse is not necessary)

1. **No side effects:** does not modify the given input and does not change any other variables that are not given
2. **Referential transparency:** if $f(x) = a$, then any uses of $f(x)$ or a can be replaced with its counterpart with guarantee that the resulting formulas are still equivalent

Functions as first-class citizens: functions that can be assigned to variables, passed around as parameters, returned from functions, etc. i.e. being treated as a regular variable

- Can be achieved using instances of anonymous classes like the `Comparator` in Unit 29

```
interface Transformer<T, R> {
  R transform(T t);
}

// Use of PECS left as an exercise
<T, R, S> Transformer<T,R> chain(Transformer<T,S> t1, Transformer<S,R> t2) {
  return new Transformer<T,R>() {
    public R transform(T value) {
      return t2.transform(t1.transform(value));
    }
  }
}
```

Pure functions: similar to mathematical functions that have *no side effects* and has *referential transparency*

- **Lack of side effects:** printing to screen, writing to files, throwing exception, changing other variables, modifying the value of the arguments, etc.

```
// Pure functions
int square(int i) {
  return i * i;
}

int add(int i, int j) {
  return i + j;
}

// Impure functions
int div(int i, int j) {
  return i / j;  // may throw an exception
}

int incrCount(int i) {
  return this.count + i; // assume that count is not final.
                         // this may give diff results for the same i.
}

void incrCount(int i) {
  this.count += i; // does not return a value
                   // and has side effects on count
}

int addToQ(Queue<Integer> queue, int i) {
  queue.enq(i);  // has side effects on queue
}
```

- **Deterministic:** given the same input, the function always produces the same output every time, ensuring referential transparency

Functional-style programming: cannot write programs out of only pure functions, but can try to minimise side effects and ensure immutability as much as possible

Functional interfaces: interface in Java with only one abstract method

- Annotated with `@FunctionalInterface`

```
@FunctionalInterface
interface Transformer<T, R> {
  R transform(T t);
}
```

- No ambiguity about which method is being overriden by an implementing subclass, thus only need to declare the parameters and method body

Lambda expressions: declaring an anonymous functional interface

```
// Declaring parameter type (unnecessary)
Transformer<Integer, Integer> incr = (Integer x) -> x + 1
// Single parameters don't need ()
Transformer<Integer, Integer> square = x -> x * x
// More than one line can be put into method body
Transformer<Integer, Integer> foo = x -> {
  System.out.println(x);
  return x * 2;  // return if the functional interface is supposed to return
}
```

Method references: used to refer to (i) static method in a class, (ii) instance method of a class or interface, (iii) constructor a class

```
Box::of            // x -> Box.of(x) used if x is the only parameter
Box::new           // x -> new Box(x)
x::compareTo       // y -> x.compareTo(y)
A::foo             // (x, y) -> x.foo(y) or (x, y) -> A.foo(x,y)

// Usage
list.stream().map(String::valueOf)
// Same as
list.stream().map(s -> String.valueOf(s))
```

- During compilation, Java searches for the matching method, performing type inferences to find the method that matches the given method reference
  - Compilation error thrown if there are multiple matches or if there is ambiguity in which method matches

Currying: breaking down a function with multiple parameters into separate, continuous set of functions that each receive just one parameter and return another function; $f : (X, Y) \to Z$ converted to $f : X \to Y \to Z$

- **Partial application: a**llows arguments to be evaluated at different times which is useful if some arguments are not available until later or saving a function for invocation later on

Higher-order function: function that takes in a single argument and returns another function (aiding in the composition of curried functions)

```
int add(int x, int y) {
  return x + y;
}

Transformer<Integer, Transformer<Integer, Integer>> add = x -> y -> (x + y);
add.transform(1).transform(1);
Transformer<Integer,Integer> incr = add.transform(1);
```

Lambda as closure: lambdas store the data from the environment where it is defined i.e. a closure

- Reduces the number of parameters needed
- Logic to do different tasks can be separated

Unit 31: Box and Maybe

Lambda as a cross-barrier state manipulator: lambdas can be used to change the internal state of an object without exposing the internal state of the object through a getter/setter

```
class Box<T> {
  private T item;

  public <U> Box<U> map(Transformer<? super T, ? extends U> transformer) {
    if (!isPresent()) {
      return empty();
    }
    return Box.ofNullable(transformer.transform(this.item));
  }
```

```
  public Box<T> filter(BooleanCondition<? super T> condition) {
    if (!isPresent() || !(condition.test(this.item)) {
      return empty();
    }
    return this;
  }
}
```

- The client is able to manipulate the data behind the abstraction barrier without knowing the internals of the object

Option types: wrapper around a value that is either there or is `null`

- Methods called when `null` is present will do nothing
- Helps to ensure that functions remain pure as they only return the same type (which encapsulates the state of whether it is `null` or not)
- Internalises all `null` checks on behalf of the programmer so missing `null` checks do not cause `NullPointerExceptions` during runtime
- Alternative: Implement a sub-class to represent when a `null` exists and return that instead but this is domain-specific (i.e. `NullCounter`)

Unit 32: Lazy Evaluation

Lazy evaluation: lambda expressions are only evaluated when parameters are supplied so they can delay the execution of the lambda body until it is needed

- Allows the build up of a sequence of complex computations without actually executing them until needed

Memoisation: caching the value of a function so that future calls to the same function avoid any expensive computations

```
class Lazy<T> {
  T value;
  boolean evaluated;
  Producer<T> producer;

  public Lazy(Producer<T> producer) {
    evaluated = false;
    value = null;
    this.producer = producer;
  }

  public T get() {
    if (!evaluated) {
      value = producer.produce();
      evaluated = true;
    }
    return value;
  }
}
```

- Since the value is evaluated once, the method body is only going to run once

Unit 33: Infinite List

Applications of lazy evaluation: building computationally-efficient data structures that does not evaluate all of its contents at once, but rather only when necessary

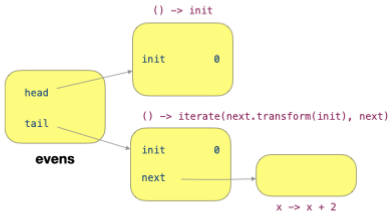
- *E.g.* building an infinite list (eagerly evaluated will result in infinite loop)

```
class InfiniteList<T> {
  private Producer<T> head;
  private Producer<InfiniteList<T>> tail;

  public InfiniteList(Producer<T> head, Producer<InfiniteList<T>> tail) {
    this.head = head;
    this.tail = tail;
  }

  public T head() {
    T h = this.head.produce();
    // Only return a valid head from the infinite list
    return h == null ? this.tail.produce().head() : h;
  }

  public InfiniteList<T> tail() {
    T h = this.head.produce();
    return h == null ? this.tail.produce().tail() : this.tail.produce();
  }

  public static <T> InfiniteList<T> generate(Producer<T> producer) {
    return new InfiniteList<T>(producer,
        () -> generate(producer));
  }

  public static <T> InfiniteList<T> iterate(T init, Transformer<T, T> next) {
    return new InfiniteList<T>(() -> init,
      () -> iterate(next.transform(init), next));
  }

  public <R> InfiniteList<R> map(Transformer<? super T, ? extends R> mapper) {
    return new InfiniteList<>(
      () -> mapper.transform(this.head()),
      () -> this.tail().map(mapper));
  }

  // Using null to represent when an element does not pass the filter
  public InfiniteList<T> filter(BooleanCondition<? super T> cond) {
    Producer<T> newHead = () -> cond.test(this.head()) ? this.head() : null;
    return new InfiniteList<>(newHead, () -> this.tail().filter(cond));
  }
}

InfiniteList<Integer> ones = InfiniteList.generate(() -> 1); // 1, 1, 1, 1, ....
InfiniteList<Integer> evens = InfiniteList.iterate(0, x -> x + 2); // 0, 2, 4, 6, ...
evens.head(); // -> 0
evens = evens.tail();
evens.head(); // -> 2
```

- Operations of the data structure can also be lazily evaluated so that the updated values of the new data structure is only evaluated when necessary (e.g. `map`)

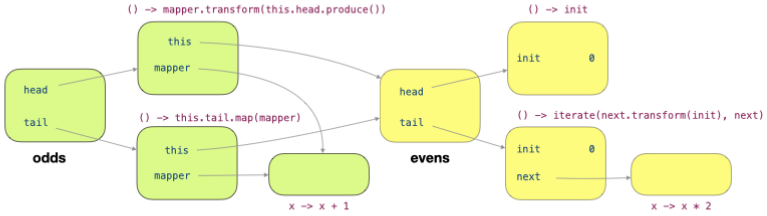
Heap diagram with lazy evaluation:

```
InfiniteList<Integer> evens = InfiniteList.iterate(0, x -> x + 2); // 0, 2, 4, 6, ...
```



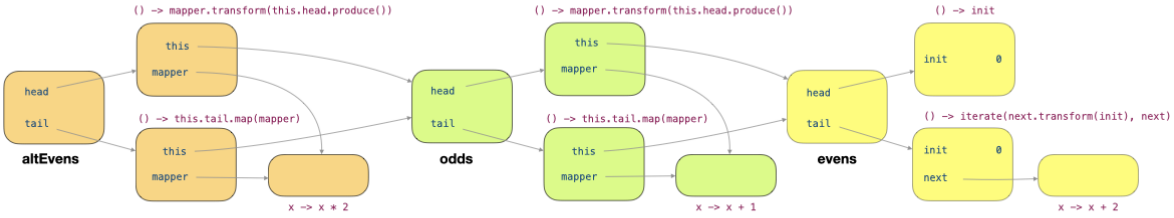
Infinite list after initialisation (note that an anonymous class is still a class that is going to be added to the heap)

```
InfiniteList<Integer> odds = evens.map(x -> x + 1); // 1, 3, 5, ...
```



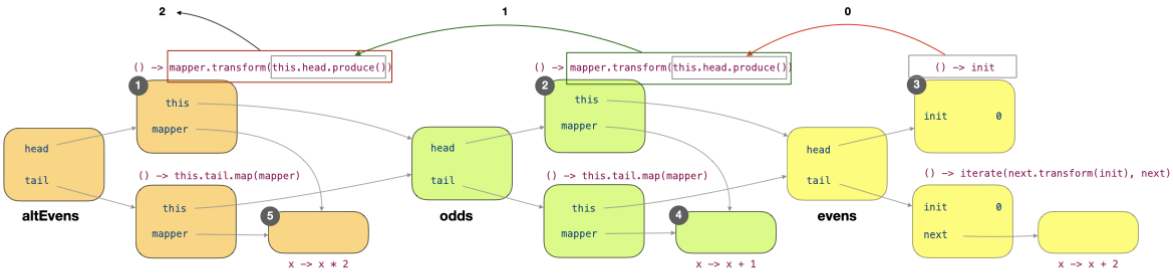
Infinite list after mapping

```
InfiniteList<Integer> altEvens = odds.map(x -> x * 2); // 2, 6, 10, ..
```



Infinite list after mapping twice

```
altEvens.head()
```



Call order

- Note that the terminating call is `head()` as all other method calls only continue building a list of lazily evaluated (and unevaluated) set of producers

Unit 34: Java Streams

CS2030S	java.util.function
`BooleanCondition<T>::test`	`Predicate<T>::test`
`Producer<T>::produce`	`Supplier<T>::get`
`Transformer<T, R>::transform`	`Function<T, R>::apply`
`Transformer<T, T>::transform`	`UnaryOp<T>::apply`
`Combiner<S, T, R>::combine`	`BiFunctions<S, T, R>::apply`

CS2030S	Java version
`Maybe<T>`	`java.util.Optional<T>`
`InfiniteList<T>`	`java.util.stream.Stream<T>`
`Lazy<T>`	N/A

Benefits of streams:

1. Don't need to use loops
2. Don't need to maintain states and counters
3. Declarative code where every step is "what we want" and not "how do we get that result" — resulting in more succinct and bug-free code

Building a stream:

1. `Stream.of(1,2,3)`
2. `Stream.generate(() -> 0)`
3. `Stream.iterate(1, x -> x + 1)`
4. `Arrays.stream(arr)`
5. `Arrays.asList(arr).stream()`
6. `List.stream()`

Terminal operations: operation on the stream that triggers the evaluation of the stream

- *E.g.* `forEach`, `reduce`, `collect`, etc.

Intermediate operations: operations that return another `Stream`

- *E.g.* `map`, `filter`, `flatMap`, etc.
- **Stateful operations:** keeping track of some state to operate
  - *E.g.* `sorted` returns a stream with the elements in the stream sorted
  - *E.g.* `distinct` returns a stream with only distinct elements in the stream
- **Bounded operations:** operations that should only be called on a finite stream as they need to gather all the available elements in the stream
  - *E.g.* `sorted` and `distinct`
- `peek` — performs an action on a fork of the stream without altering the state of the stream
  - Useful for doing things like viewing the contents of the stream while still performing operations after the fact

```
Stream.iterate(0, x -> x + 1)
  .peek(System.out::println)
  .takeWhile(x -> x < 5)
  .forEach(x -> {});
```

- `reduce(identity, (acc, cur) -> {})` (or `fold` or `accumulate`) — applies a lambda repeatedly on the elements of the stream, reducing it into a single value

```
Stream.of(1, 2, 3).reduce(0, (x, y) -> x + y);
```

Flat mapping: if the result of a map on a stream is another stream (i.e. a stream of a stream), `flatMap` can be used to flatten the second stream into a single stream

Truncating an infinite stream: converting from an infinite stream to a finite stream

- `stream.limit(n)` — returns a finite stream containing the first `n` elements of the stream
- `takeWhile(pred)` — returns a stream containing the elements of the stream that satisfy the predicate, terminating when the predicate becomes false; resulting stream might be infinite if the predicate never becomes false

Element matching: testing if the elements of a stream pass a given predicate

- `noneMatch` — true if none of the elements pass the given predicate
- `allMatch` — true if all of the elements pass the given predicate
- `anyMatch` — true if at least one of the elements pass the given predicate

Consumed once: streams can only be operated on once; attempting to operate on a stream more than once results in an `IllegalStateException` during compile time

- *I.e.* once any operation (intermediate or terminal) is called on a stream, the same stream cannot be operated on again. However, the resulting stream (from the operation) can be assigned to a variable and operated on again

```
Stream<Integer> s = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> o = s.filter(x -> x % 2 == 0);
s.map(x -> x + 1);  // this is illegal
o.map(x -> x + 1);  // this is allowed
```

Unit 35: Loggable

```
class Loggable<T> {
  private final T value;
  private final String log;

  private Loggable(T value, String log) {
    this.value = value;
    this.log = log;
  }

  public static <T> Loggable<T> of(T value) {
    return new Loggable<>(value, "");
  }

  public <R> Loggable<R> flatMap(
    Transformer<? super T, ? extends Loggable<? extends R>> transformer) {
    Loggable<? extends R> l = transformer.transform(this.value);
    return new Loggable<>(l.value, l.log + this.log);
  }

  public String toString() {
    return "value: " + this.value + ", log: " + this.log;
  }
}
```

Unit 36: Monad

Monad: class that holds a value and some side information about the given value

- Contains an `of` method to initialise the value and side information and a `flatMap` operation to update the value and side information
- Following the laws ensure that any and all operations performed on the monad will result in the same value and side information

1. **Left identity law:** `Monad.of(x).flatMap(x -> f(x)) == f(x)`
2. **Right identity law:** `monad.flatMap(x -> Monad.of(x)) == monad`
3. **Associative law:** `monad.flatMap(x -> f(x)).flatMap(x -> g(x)) == monad.flatMap(x -> f(x).flatMap(y -> g(y)))` (regardless of how we group the calls for `flatMap`, their behaviour must be the same)

Functors: class that holds a value (without the need to maintain side information)

- Contains an `of` method and `map` operation that updates the value but changes nothing to the side information

1. **Identity law:** `functor.map(x -> x) == functor`
2. **Composition law:** `functor.map(x -> f(x)).map(x -> g(x)) == functor.map(x -> g(f(x)))`

Unit 37: Parallel Streams

Single-core processor: executes only one instruction at a time, i.e. only one process can run at any one time

- OS will switch between processes (aka time slicing) to allow multiple processes to run “at once”

Concurrency: dividing computation into subtasks (**threads**)

- Each thread can represent a unit of work unrelated to the others
- If a thread is busy, the processor can switch to a different thread so other computations can be performed
- *E.g.* splitting I/O and UI rendering into separate threads
- Threads run within a process as they are units of work that the processor can take up

Parallelism: actually having multiple processes running simultaneously

- Enabled if a processor is capable of running multiple instructions at the same time or there are multiple cores/processors and the instructions are distributed across these processors
- All parallel programs are concurrent but not all concurrent programs are parallel

`forEach` **and parallel streams:** `forEach` will perform its actions on the units of worked that have already been completed first so the results might be reordered

- Remedy: `forEachOrdered` but some benefits of parallelisation is lost (follows the encounter order)

Embarrassingly parallel: computation where each element is processed individually without depending on other elements and the only communication needed for each parallel subtask is to combine the result of each subtask into a cumulative total

`parallel()` **:** marks a stream to be processed in parallel (can be inserted anywhere in the chain as this is lazily evaluated)

- If `sequential()` was called on the same chain as well, the last call will override the others

`parallelStream()` : used in place of `stream()`

Optimal conditions for parallelisation: stream operations must not interfere with the stream data and must be stateless with minimal side effects

Interference: one of the stream operations modifies the source of the stream during the execution of the terminal operation (applicable regardless of parallelisation)

```
List<String> list = new ArrayList<>(List.of("Luke", "Leia", "Han"));
list.stream()
    .peek(name -> {
         if (name.equals("Han")) {
           list.add("Chewie"); // they belong together
         }
      })
    .forEach(i -> {});
```

- Results in `ConcurrentModificationException`

Stateless: result does not depend on any state that might change during the execution of the stream

```
Stream.generate(scanner::nextInt)
    .map(i -> i + scanner.nextInt())
    .forEach(System.out::println)
```

- Depends on the state of the standard input so parallelising this causes incorrect output
- To ensure that the output is correct, additional work is necessary to ensure that the state updates are visible to all parallel subtasks

Side effects: can lead to incorrect results in parallel execution

Non-thread-safe data structures: data structures that are prone to inconsistencies when updated by multiple threads

- *E.g.* `ArrayList` might be updated with values that are not in the right order

```
List<Integer> list = new ArrayList<>(
    Arrays.asList(1,3,5,7,9,11,13,15,17,19));
List<Integer> result = new ArrayList<>();
list.parallelStream()
    .filter(x -> isPrime(x))
    .forEach(x -> result.add(x));
```

- Alternatives: (a) using `collect()` instead (`.collect(Collectors.toList())`) or (b) using thread-safe data structures like `CopyOnWriteArrayList`

Parallel reduce: a sub-stream can be reduced concurrently and combined

- `identity` — starting value of each sub-stream
- `accumulator` — accumulator for each sub-stream
- `combiner` — combines the result of each sub-stream's accumulation in the encounter order of elements
- Rules:
  1. `combiner.apply(identity, i)` must equals `i`
  2. `combiner` and `accumulator` must be associative meaning that the order of application does not matter
  3. `combiner` and `accumulator` must be compatible: `combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)`

Performance of parallelisation: not always going to improve the performance as creating a thread incurs some overhead and the overhead of creating too many threads might outweigh the benefits of parallelisation

Encounter order: order that elements are added (usually associated with ordered streams from ordered collections or `of`)

- **Stable operations:** operations respect the encounter order of a stream such as `distinct` and `sorted`
- Operations that attempt to preserve encounter order can get expensive as they need to coordinate between the streams to maintain the order such as `findFirst`, `limit`, and `skip`

Unordered streams: `unordered()` can be used to make parallel operations more efficient if the original order of elements is not important

Unit 38: Threads

Synchronous programming: if a method takes too long to process, it can stall the entire program, i.e. blocking it until it returns

- Inefficient if there are frequent method calls that block for a long period of time
- The processing can be done in the background instead and allow other tasks to be performed in the mean time

Threads: single flow of execution in a program

- `main` **thread:** default thread that all single-threaded programs run; automatically creating when `main()` is invoked

`java.lang.Thread` **:** encapsulate a function to run in a separate thread (wrapped in a `Runnable` instance as a lambda)

- `start()`**:** causes the `Runnable` to execute asynchronously
  - Returns immediately, not when `Runnable` is finished
  - Allows the program to not be blocked while waiting for `Runnable` is running
- The program only exits after all the threads created run to their completion

```
new Thread(() -> {
  for (int i = 1; i < 100; i += 1) {
    System.out.print("_");
  }
}).start();

new Thread(() -> {
  for (int i = 2; i < 100; i += 1) {
    System.out.print("*");
  }
}).start();
```

Scheduler: OS tool to decide which threads to run when, and on which core/processor

- Results in different interleaving executions every time a program is executed

Thread name: `Thread.currentThread().getName()` prints the name of the thread that is currently executing

Sleeping: causes the current execution thread to pause execution immediately for a given period (in milliseconds)

- Once the timer is up, the thread is ready to be chosen by the scheduler again to run (it is not immediately resumed as the scheduler might already be running a different task so this task needs to wait)

```
Thread findPrime = new Thread(() -> {
  System.out.println(
      Stream.iterate(2, i -> i + 1)
          .filter(i -> isPrime(i))
          .limit(1_000_000L)
          .reduce((x, y) -> y)
          .orElse(null));
});

findPrime.start();

// isAlive used to periodically check if another thread is still running
while (findPrime.isAlive()) {
  try {
```

```
    Thread.sleep(1000);
    System.out.print(".");
  } catch (InterruptedException e) {
    System.out.print("interrupted");
  }
}
```

Unit 39: Async

Limitations of `Thread`:

1. **Use of shared variables:** threads don't return anything; prone unsafe access and operations on shared variables
2. **Coordinating execution:** no mechanism to specify execution order and dependencies among them
3. **Exceptions:** one thread might encounter an exception — what happens to its dependencies
4. **Overhead:** reusing threads after it is done with computation is more efficient than creating new threads all the time

`CompletableFuture` **monad:** abstraction to counteract aforementioned limitations of `Thread`; encapsulates the value it takes on and whether the task that produces it is ready or not

- **Creation:**
  - **Independent:** `completedFuture(<value>)`, `CompletableFuture<Void> runAsync(Runnable)`, `CompletableFuture<T> supplyAsync(Supplier<T>)` (asynchronous)
  - **Dependent:** `allOf(CompleteableFuture<T>...)`, `anyOf(CompletableFuture<T>...)`
- **Chaining:** specify sequence of computations to perform
  - Same thread as caller:
    - `thenApply` — `map`
    - `thenCompose` — `flatMap`
    - `thenCombine` — `combine`
  - Different thread from caller: `thenApplyAsync`, `thenComposeAsync`, `thenCombineAsync`
- **Getting results:** `get()` (throws `InterruptedException` and `ExecutionException`) or `join()` (no exceptions)
  - Should be called last as blocking call

```
int findIthPrime(int i) {
    return Stream
        .iterate(2, x -> x + 1)
        .filter(x -> isPrime(x))
        .limit(i)
        .reduce((x, y) -> y)
        .orElse(0);
}

// Runs on two different threads asynchronously
CompletableFuture<Integer> ith = CompletableFuture.supplyAsync(
  () -> findIthPrime(i));
CompletableFuture<Integer> jth = CompletableFuture.supplyAsync(
  () -> findIthPrime(j));
```

```
// Combines the results of the two threads
CompletableFuture<Integer> diff = ith.thenCombine(jth, (x, y) -> x - y);

// Waits for all threads to complete computation
diff.join();
```

- **Exception handling:** `handle((value, exception) -> exception == null ? value : default value)`
  - Exceptions thrown passed down chain of calls until `join()` to handle (throws `CompletionException` with information on the original exception)

```
// This throws CompletionException
CompletableFuture.<Integer>supplyAsync(() -> null)
  .thenApply(x -> x + 1)
  .join();

// This handles the exception and uses 0 as the default value
cf.thenApply(x -> x + 1)
    .handle((t, e) -> (e == null) ? t : 0)
    .join();
```

Unit 40: Fork and Join

Thread pool: collection of threads waiting for a task to execute and a collection of tasks (queue); allows reuse of threads

```
Queue<Runnable> queue;
new Thread(() -> {
  while (true) {
    if (!queue.isEmpty()) {
      Runnable r = queue.dequeue();
      r.run();
    }
  }
}).start();

for (int i = 0; i < 100; i++) {
  int count = i;
  queue.add(() -> System.out.println(count));
}
```

Fork-join model: parallel divide-and-conquer; breaks up a problem into identical sub-problems (fork), solving a smaller version recursively, and combining the results (join) — repeated until the problem size is small enough (base case) to be solved sequentially

`RecursiveTask<T>`: problem to parallelise

- `fork()`: submit sub-problem to solve
- `join()`: waits for smaller tasks to complete and return
  - Order of calls should be reverse to the order of `fork()` calls
- `compute()`: behaviour of (sub-)problem

```
class Summer extends RecursiveTask<Integer> {
  private static final int FORK_THRESHOLD = 2;
  private int low;
  private int high;
```

```
  private int[] array;

  public Summer(int low, int high, int[] array) {
    this.low = low;
    this.high = high;
    this.array = array;
  }

  @Override
  protected Integer compute() {
    // stop splitting into subtask if array is already small.
    if (high - low < FORK_THRESHOLD) {
      int sum = 0;
      for (int i = low; i < high; i++) {
        sum += array[i];
      }
      return sum;
    }

    int middle = (low + high) / 2;
    Summer left = new Summer(low, middle, array);
    Summer right = new Summer(middle, high, array);
    left.fork();  // adds task to a thread pool so a thread calls its compute()
    return right.compute() + left.join();
  }
}

Summer task = new Summer(0, array.length, array);
int sum = task.compute();
```

Work-stealing algorithm: every thread contains a queue of tasks to perform

- **Idle thread:** check its queue
  - If not empty → pick up a task a the head of the queue and execute (using `compute()`)
  - If empty → pick up a task from the tail of the queue of another thread to run (i.e. work stealing)

`fork()`**:** caller (sub-problem) adds itself to the head of the queue of the executing thread (parent thread); ensures most recently forked task is executed next

`join()`**:** split into 3 cases

1. **Subtask has not been executed:** `compute()` called and subtask executed
2. **Subtask has been completed:** work has been stolen so result is read and `join()` returns
3. **Subtask has been stolen:** current thread finds another task to execute

Appendix

> ❗ **Note** commands starting with `$` indicate a command run in the terminal while `#` indicate a command run in an emulated environment like `jshell`

`jshell`

- `$ jshell` — launch `jshell`
- `# /open *.java` — open a given `*` Java file in `jshell`

Additional Notes

1. Check carefully whether a method is being overridden.
2. You cannot instantiate an interface.
3. After exception is thrown, everything else in try is ignores, and the `catch` and `finally` block are run.
4. LSP: You need to write what property of the superclass that is no longer holds for the subclass.
5. Generics allow classes / methods that use any reference type to be defined without resorting to using the `Object` type. It enforces type safety by binding the generic type to a specific given type argument at compile time. Attempt to pass in an incompatible type would lead to compilation error.
6. Object is upper bounded by `? extends Object`
7. The method grade in Assessments can be overridden by individual subclasses — polymorphism can be used here.
8. Existing code that has been written to invoke A's copy would still work if the code invoked B's copy instead after B inherits from A.
9. Do not forget the `this` when drawing the call stack.
10. Do not forget to create that arguments are allocated on stack too.