Ger Teck

# CS2030S Summary and Notes

AY22/23 Sem 2
github.com/gerteck

## 1. Variables & Types

### Type Systems

- Dynamically typed: Variables can hold values of different types, checking of right type is done during execution of the program.
- **Statically typed**: [Java] Variables can only hold values of the same type. Type must be declared at compile time (CTT), type cannot be changed once assigned.
- **Strong vs. Weak Typing**: Strongly typed language enforces strict rules in type system to ensure type safety, to catch type errors during compile time rather than leaving to runtime. Weakly (Loosely) typed is more permissive in terms of type checking.
- A variable is an abstraction! (of a name to some location in computer memory)
- **Primitive Types in Java:**

```
byte <: short <: int <: long <: float <: double
 char <: int
 boolean
```

- **Subtypes:**
  If S is a subtype of T, `S <: T`, we say that a piece of code written for variable of type `S` can be safely used on variables of type `T`. (LSP)
  - Widening Type Conversion → e.g. type `S` being put into type `T` (Auto).
  - **Narrowing Type Conversion** → requires typecasting (Error: Incompatible types / ClassCastException)
- **Run Time Type (RTT) vs. Compile Time Type (CTT)**

### Range-based For Loops

- `for(x : collection)` or `for(T x : collection)`
- Works with any collection that implements `Iterable<U>` where $U$ is implicit-convertible to $T$, even if $T$ is a primitive type.

### Liskov Substitution Principle

If `S <: T`, then

- Any property of `T` should be a property of `S`, including fields and methods. An object of type `T` can be replaced with `S` without changing some desirable property of the program.
- "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 is a subtype of T."

## 2 - 4. OOP Principles

**Abstraction, Encapsulation, Polymorphism and Inheritance.**

### Abstraction

- Intention revealing, easy/simple to use and understand. Allows compartmentalized computation and effects. Hide how tasks are performed, and reduce repetition through code reuse.
- **Abstraction Barrier**: Implementer vs. Client

### Encapsulation

- Keeping all data and functions related to a composite data type together within an abstraction barrier.
- Composite Data Types - Class is a data type with a group of functions (methods), data (fields).
- `private` attributes, `public` methods. (Data hiding) Use constructors to initalize object and access private variables. **Ex. Constructor:**

```
public Circle(Point p, double r){
  this.p = p;
  this.r = r;
\\this keyword used to refer to self / Object
}
```

⋆ Any reference variable that is not initialized will have the special reference value `null`.

### Tell, Don't Ask

- Avoid using unnecessary accessors (getters) and mutators (setters) as they break the abstraction barrier, [encapsulation]. Tell the object/class to do something, don't ask for details and do it yourself.
- Encourage behaviour to be moved into an object to go with the data.
  **Static vs. Instance Methods and Fields**
- To associate method or field with class, we declare them with `static` keyword. We may also add `final` to indicate that value of field will not change, and `public` to indicate that field is accessible from outside the class.
- `main` method is entry point to Java program. `main` method takes in an array of strings as parameters. Defined as follows:

```
public static void main(String[] args){
}
```

- use `import` to access classes from Java standard libraries.

### Inheritance

- "has-a" relationship: → use composition (e.g. Circle has-a point as center)
- "is-a" relationship → `extends` (subtyping)
  ⋆ In Java, every class not extending another class inherits from the class Object implicitly. ("ancestor" of all classes), Object is at the root of the class hierarchy.
- we use `super` to call the constructor of the superclass, to initialize (in example) its center and radius (since the child has no direct access to these fields that it inherited).

```
class ColoredCircle extends Circle {
  private Color color;

  public ColoredCircle(Point center, double
    radius, Color color) {
    super(center, radius);  // call the parent'
    s constructor
    this.color = color;
  }
}
```

- Inheritance tends to be overused, make sure inheritance preserves the meaning of subtyping.

Ger Teck

## Polymorphism

Changes how existing code behaves, without changing a single line existing of code.

- **Dynamic Binding** → method invoked is determined at run time RT (while possibilities [method signatures] determined at compile time CT).
- **Method Overloading (static Polymorphism**: same method name, but **different parameter types/number of parameters**
- **Method Overriding (dynamic Polymorphism**: same **method signature**: (method name, type/number/order of parameters). Note method descriptor = method signature + return type.
(ex. of polymorphism → overriding parent class method)
- Return type of overriding method can be subtype but not super type, or compiler will throw an error.
- Exceptions and return types are not part of function signature.
- `Overridding Object::toString method`

```
//Override annotation to help compiler help us
  @Override
  public String toString() {
      return "{ center: " + this.c + ", radius:
     " + this.r + " }";
  }
//center: (0.0, 0.0), radius: 4.0
```

- `private`, `static`, `final` methods cannot be overridden

### Method Invocation

Determine which non-static method to use through dynamic binding. During **compile time**, determine most specific method descriptor method to invoke (using `Type Information`. During **run time**, use `Objects` to determine binding.

- Class methods ( `static` do not support dynamic binding (resolved statically)

## Abstract Classes

- Cannot be instantiated. As long as one method is abstract, whole class is abstract.
- A concrete class cannot have abstract methods.

## Interface "can-do"

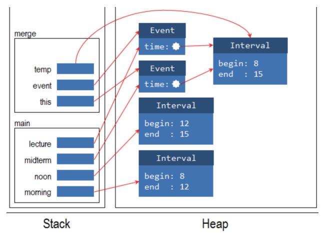
- Interface is also a type and is declared as follows. As it models what an entity can do, name usually ends with -able suffix.

```
interface GetAreable {
    public abstract double getArea();
//  public abstract fields r default, equiv to
    double getArea();
}
abstract class Shape implements GetAreable {
}
```

- For a concrete class to implement an interface, override all abstract methods **(*using same method signature)** from interface and provide implementation. Otherwise, the class becomes abstract.
- Note: A class can only extend from one superclass, but it can implement multiple interfaces.
- Note: An interface can `extend` from one or more other interfaces, but an interface cannot extend from another class. (Neither can an interface `implements` other interfaces as it is abstract.

## Stack and Heap (Memory Model)

**Stack** is the region where all variables (including primitives and object references) are allocated in and stored.
**Heap** is the region of memory where all objects are allocated in and stored,



**Stack frame of Primitives** Note radius value is primitive type instead of reference, we copy the value onto the stack. Java uses call by value for primitive types, and call by reference for objects.

- `this` reference is always placed on the stack when calling a non-static method
- The memory allocated on the stack is deallocated when a method returns. The memory allocated on the heap, however, stays there as long as there is a reference to it. The JVM runs a garbage collector that checks for unreferenced objects on the heap and cleans up the memory automatically.

## Wrapper Class for Primitives

"Making primitive types less primitive". A wrapper class is a class that encapsulates a type.

```
Integer i = new Integer(2); // = new Integer.
    valueOf(int a)
int j = i.intValue();
```

- All wrapper class objects are immutable. Autoboxing → primitive value converted to instance of Wrapper class ( `int` → `Integer` ). Unboxing is the opposite type conversion.
- Wrapper classes incur cost of allocating memory for object and collecting garbage afterwards. Because they are immutable, new object must be created for update of value. (Inefficient)

## Modifiers

- In Order of Java modifiers:

```
public protected private abstract default
    static sealed non-sealed final transient
    volatile synchronized native strictfp
```

- `private` → only within class, `public` → everywhere
- `default` → only within package, `protected` → within package or outside package through child class
- `final` variable → only assigned once (immutable)
- `final` class → cannot be inherited from
- `final` method → cannot be overridden

Ger Teck

## Casting

```
// Circle <: Shape <: GetAreable
GetAreable findLargest(GetAreable[] array){...}
GetAreable ga = findLargest(circles);  // ok

Circle c1 = findLargest(circles); // error
Circle c2 = (Circle) findLargest(circles); // ok
```

- In the snippet above, we can be sure (even prove) that the returned object from findLargest must have a run-time type of Circle since the input variable circles contains only Circle objects.
- Only cast when you can prove it is safe.

## Variance

Variance of types refers to how the subtype relationship between complex types relates to the subtype relationship between components.

- Let $C(S)$ corresponds to some complex type based on type S. An array of type $S$ is a complex type.
  We say a complex type is:
- **covariant** if $S <: T$ implies $C(S) <: C(T)$
- **contravariant** if $S <: T$ implies $C(T) <: C(S)$
- **invariant** if it is neither covariant nor contravariant.

  Note:
- **Array is covariant in Java**. This means that, if $S <: T$ implies $S[] <: C[]$
  By making array covariant, Java opens up the possibility of run-time erros without typecasting.

```
Integer[] intArray = new Integer[2] {
  new Integer(10), new Integer(20)
};
Object[] objArray;
objArray = intArray;
objArray[0] = "Hello!"; // <— compiles!

// But will lead to a runtime error, as we are
// stuffing a string into an array of integers.
     (Heap Pollution)
```

## Exceptions

### try catch finally blocks

```
try {
  new Circle(new Point(1,1), 0);
  // everything afterwards is skipped (r//= 0)
  System.out.println("This will not be reached")
    ;
} catch (IllegalCircleException e) {
  //runs if there is an exception
} finally {
  always runs
}
```

- exception is passed up the call stack until it is caught
- after exception is caught everything after proceeds normally.

### Creating Own Exceptions

```
class IllegalCircleException extends
    IllegalArgumentException {
  Point center;
  IllegalCircleException(String message) {
    super(message);
  }
  IllegalCircleException(Point c, String
    message) {
    super(message);
    this.center = c;
  }
@Override
    public String toString() {
      return "The circle centered at " + this.
    center + " cannot be created:" + getMessage
    ();
    }
```

- When you override a method that throws a checked exception, the overriding method must throw only the same, or a more specific checked exception, than the overridden method.
- Follows the Liskov Substitution Principle. The caller of the overridden method cannot expect any new checked exception beyond what has already been "promised" in the method specification. (must throw $E_1$ such that $E_1 <: E_0$)

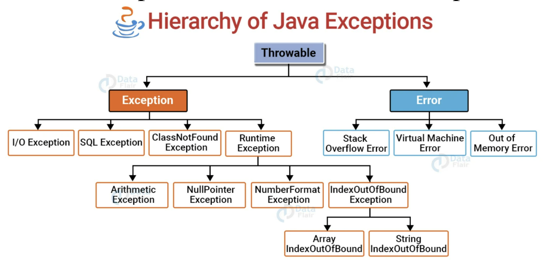
### throw Exceptions

```
public Circle(Point c, double r) throws
    IllegalCircleException {
    if (r < 0) {
      :throw new IllegalCircleException("radius
    cannot be negative.");
    }:
    this.c = c;
    this.r = r;
  }
}
// Throwing to caller
```

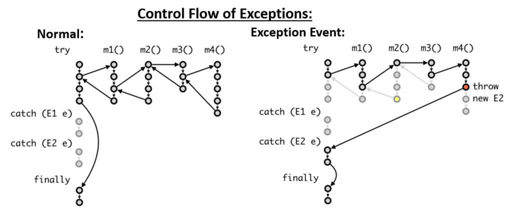
throw method causes method to immediately return. **unless there is** finally **block** which will run before exception gets thrown out.

### Checked vs Unchecked Exceptions

- An unchecked exception is an exception caused by a programmer's errors. e.g. ClassCastException. Not explicitly caught or thrown.
- A checked exception is an exception that a programmer has no control over. Need to actively anticipate the exception and handle them. e.g. FileNotFoundException. A checked exception must be handled to compile.



- A **checked exception** (Caught at Compile Time) must be handled either by **re-throwing** or with a try catch block, whereas an **unchecked exception** (Caught at Runtime) isn't required to be handled.



Ger Teck

## 5. Generics

- Allows classes/methods (that use reference types) to be defined without resorting to use of Object type.
- Ensures **type safety** → binds a generic type to a specfic type at compile time. Attempt to pass an incompatible type would lead to a compilation error.
- Errors will be at compile time instead of runtime.
- Generics are **invariant** in Java.

### Generic Class:

```
class Pair<S extends Comparable<S>, T>
    implements Comparable<Pair<S, T>> {...}
class DictEntry<T> extends Pair<String, T>
    {...}
```

### Generic Method:

```
// note generic goes before return type!
public static <T> boolean contains(T[] arr, T
    obj){...}
//to call generic method:
A.<String>contains(strArray, "hello");
```

- ⋆ type parameter `<?>` is declared before return type.
- note bounded type parameters. **Notes:**

```
B implements Comparable<B>{...}
A extends B {...}
 // A <: B <: Comparable<B>
// Comparable<A> INVARIANT Comparable<B>
// Comparable<A> <: Comparable<? extends B>
```

## Type Erasure

- at compile time, type parameters are replaced by `Object` or the bounds (e.g. `T extends Comparable<T>`, `T` is replaced by / erasured to `Comparable`)

```
Integer i = new Pair<String, Integer>("x", 4).
    foo() //before
Integer i = (Integer) new Pair("x", 4).foo() //
    after
```

- Java Generics are not **reifiable** due to type erasure. (Reifiable type where full type information is available during run time.)
- Hence, to prevent heap pollution, where Java arrays are reifiable, arrays are not generic.

### Suppress Warnings

- `@SupressWarnings` can only apply to declaration.

```
@SuppressWarnings("unchecked") \\, "rawtype"
T[] a = (T[]) new Object[size];
this.array = a;
```

### Raw Types

- A generic type used without type arguments.
- Only acceptable as an operand of `instanceof`
- `@SuppressWarnings("rawtypes")` : This is when compiler is not sure if line is a type safe operation, as we are using a Raw Type (generic type w/o type arguments).
- The compiler cannot check e.g. if it is safe to pass an `Integer` to the `keep` method. (in case it is populated with some other type, which could e.g. cause a `ClassCastException` trying to cast `Integer` to a `String`. Hence, allow, but warn the programmer (unsafe). (Raw types must not be used)

## 6. Wildcards

- **upper-bounded**: `? extends` **covariant**
  - if `S <: T`, then `A<? extends S>` <: `A<? extends T>`
- **lower-bounded**: `? super` : **contravariant**
  - if `S <: T`, then `A<? super T>` <: `A<? super S>`
- **unbounded**: `?`
  - `Array<?>` is the supertype of all generic `Array<T>`

### PECS Principle

- `PE` → If variable produces T values, use `List<? extends T>`
- `CS` → If variable consumes T values, use `List<? super T>`
- If both producer & consumer → use wildcard `<?>`

## Type Inference

- Ensures **Type Safety** → compiler can ensure that `List<myObj>` holds objects of type `myObj` at compile type instead of runtime.
- Type inference always chooses narrowest bound
- `<? super Integer>` ⟹ inferred as `Object` (supertype of Integer)
- `<? extends Integer>` ⟹ inferred as `Integer`

### Diamond Operator `<>`

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

- only for instantiating a generic type - not as a type e.g. `new Pair<>() //ok` `Pair<> p = ... // not ok`
- generic methods: type inference is automatic
- `A.contains()` and not `A.<>contains()` (no need)

### Constraints for Type Inference

- 1. target typing → the type of expression (e.g. `Shape`)
- 2. type parameter bounds → `<T extends GetAreable>`
- 3. parameter bounds → `Array<Circle>` <: `Array<? extends T>`. So, `Circle <: T`

```
public static <T extends GetAreable> T
    findLargest(Array<? extends T> array)

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

## 7. Immutability

An immutable class → an instance cannot have any visible changes outside its abstraction barrier.

### Making Immutable Classes

- `private final` : We make the class itself `final` to disallow inheritance and overriding. Note the the `final` keyword prevents assigning new value to the field, but does not prevent the field from being mutated. **final not sufficient for immutabilitiy.**
- We return a new instance every time to prevent mutating the current instance. (Copy on write semantic allows us to avoid aliasing bugs without creating excessive copies of objects.)

### Advantages of Immutability:

- **Ease of Understanding & Safe Sharing:** We are sure that an object's properties are unchanged unless explicitly re-assigned. Allows us to safely share instances of the class and reduces need to create multiple copies of the object.
- **Saves Space**: Share references until instances need to be modified (which creates a new copy)
- **Safe Concurrent Execution:** Allows multiple threads of code to run in interleaved fashion, without objects being changed.

### Varargs (variable arguments)

- Java 5 Introduced the concept of varargs, **a method parameter of variable length**.
- will be passed to the method as an array of items.
- `public void of(T ... items){}` → items will be `T[]`
- `@SafeVarargs` annotation if `T` is a generic type.

## Nested Classes: Static Nested & Inner Classes

**Static nested, inner class**: Java allows us to define a class within another class/method. **Nested Classes** are used to group logically relevant classes together. Nested class is a field of the containing class, and and can access fields and methods of the container class.

- `static nested class` → associated with the containg class. Can only access **static** field / methods of containing class.
- `inner class (non static )` → associated with an instance. Can access all fields/methods of containing class.

### qualified `this`

e.g. `A.this.x` : Used to reference container classes' variables. Otherwise we can't.

```
class A {
 private int x;

 class B {
   void foo() {
     this.x = 1; // error
    A.this.x = 1; // ok
   }
 }
}
```

## Local Class (Class in Methods)

- class defined within a method, like a local variable.
- can access class and instance variables from the enclosing class (use qualified `this`) + local variables of enclosing method.
- `effectively final` → variable does not change after initialisation.
- **Will not compile** if variables accessed are NOT effectively final.

### Variable Capture

- When method returns, the local variables are removed from stack. But an instance of that local class may still exist.
- The local class, as it can access the local variables in the enclosing method, makes a copy of the local variables inside itself. We say that the local class captures the local variables.
- In order to prevent bugs, Java only allows a local class to access variables that are explicitly declared final or implicitly final . (effectively final).

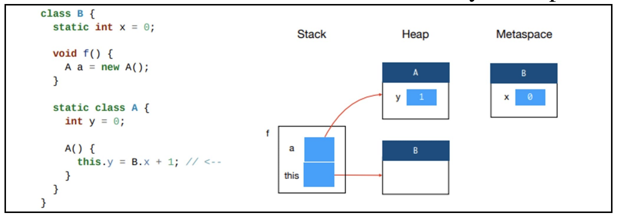
## Anonymous Class

Declare a class and instantiate it in a single statement. It is anonymous as no name, just like a local class, captures the variables of the enclosing scope.

- **Format:** `new Constructor(arguments) { body }` , or `new ( className implements interface ) (arguments) { body }`
- Cannot implement more than one interface, Cannot extend a class and implement an interface at same time.
- Same rule as local class for variable access.

### Stack & Heap:

- **Static Class:** Java RT: Static fields in memory metaspace.



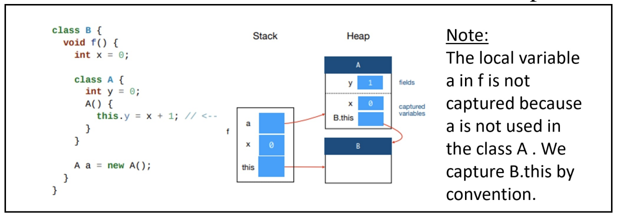
```
class B {
  static int x = 0;

  void f() {
    A a = new A();
  }

  static class A {
    int y = 0;

    A() {
      this.y = B.x + 1; // <--
    }
  }
}
```

- **Local Class:** Local Class: Variables in method will be captured.

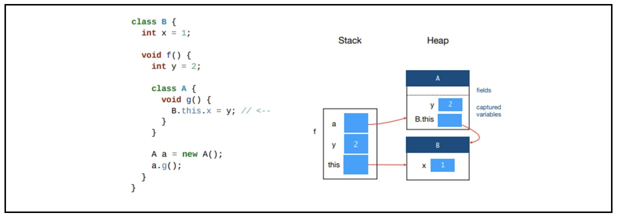


```
class B {
  void f() {
    int x = 0;

    class A {
      int y = 0;
      A() {
        this.y = x + 1; // <--
      }
    }

    A a = new A();
  }
}
```

Note:
The local variable a in f is not captured because a is not used in the class A . We capture B.this by convention.



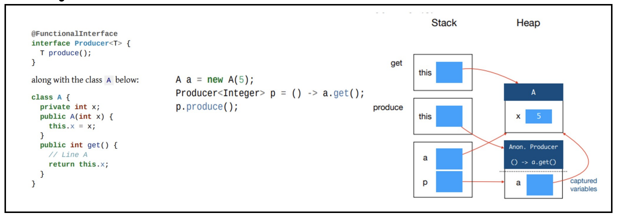
```
class B {
  int x = 1;

  void f() {
    int y = 2;

    class A {
      void g() {
        B.this.x = y; // <--
      }
    }

    A a = new A();
    a.g();
  }
}
```

- **Anonymous Class**



```
@FunctionalInterface
interface Producer<T> {
  T produce();
}
```

along with the class `A` below:

```
class A {
  private int x;
  public A(int x) {
    this.x = x;
  }
  public int get() {
    // Line A
    return this.x;
  }
}
```

```
A a = new A(5);
Producer<Integer> p = () -> a.get();
p.produce();
```

# 8. Functions (F. Programming)

## Functions

- Idea of Function: Some mapping from domain to codomain. It has to be deterministic, immutable, so that we can pass these methods around as objects.
- **Cross state barrier:** with functions, you can now pass in functions into a class to use instead of making a new class method for every time of data manipulation.
- **Functions as first class citizens:** functions are treated like any other variables, passed around.
- **Referential Transparency:** if $f(x) = y$, then any $y$ can be substituted with $f(x)$.
- **Pure Function**: No side effects (e.g. printing, changing value, throwing exceptions), every input mapped to an output (null is not within codomain), deterministic, and must return a value.

## Functional Interface:

Conceptually, a functional interface has exactly one abstract method. We annotate function interfaces with `@FunctionalInterface` annotation.

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

## Lambda Expressions

Functional interfaces have only one abstract method with `@FunctionalInterface` annotation. There is no ambiguity about which method is being overridden by an implementing subclass. The type can be inferred, simplify the instantiation of the class into a lambda expression!

```
Transformer<Integer, Integer> square = x -> {
    return x * x; };
Transformer<Integer, Integer> incr = x -> {
    return x + 1; };

Transformer<Integer, Integer> square = x -> x *
    x;
Transformer<Integer, Integer> incr = x -> x + 1;
```

## Method Reference

- **Double Colon Notation** for method reference.
- For **static method in class** ( `ClassName::staticMethodName` )
- For **instance method in class** ( `instanceName::MethodName` )
- For **constructor of a class** ( `ClassName::new` )

```
Box::of              // x -> Box.of(x)
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)
```

- **at compile time**: Java searches for the matching method, performing type inferences to find the method that matches the given method reference. A compilation error will be thrown if there are multiple matches or if there is ambiguity in which method matches.

## Currying

Converting functions with multiple arguments into a sequence of function that each take in a single argument

- Basically the act of returning a function that stores / computes the data of the previous input.

```
Transformer<Integer, Transformer<Integer,
    Integer>> add = x -> y -> (x + y);

add.transform(1)     //gives a lambda
add.transform(1).transform(2) //returns 3
increment.transform(3)  //returns 4
```

- translate a general n-ary functions a n unary functions.
- stores the data from the environment it is defined. `closure` : construct that stores a function together with the enclosing environment.
- **closure** $\rightarrow$ same concept as variable capture $\rightarrow$ variable must be either explicitly declared as final or effectively final.

# Lazy Evaluation

Lambda expressions $\rightarrow$ Nothing is executed by simply declaring them, we are saving them to be executed later. Allows us to delay the execution of code, saving them until we need it later, enabling lazy evaluation (a powerful mechanism). We can build up a sequence of complex computations, evaluated on demand.

## Delayed computation with lambda functions

```
@FunctionalInterface
interface Producer<T> { T produce(); }

@FunctionalInterface
interface Task {void run(); }
```

## Memoization

```
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;
  }
}
```

## 9. Monad

- A monad is a structure with at lest two methods of and flatMap that obeys three laws. Contains a value + side information
- `of` method to initalize the value & side information
- `flatMap` method to update value & side information (takes in some function)

### Monad Laws

#### Left Identity Law:

- `Monad.of(x).flatMap(y —> f(y))` is equivalent to `f(x)`.
- Basically that flatMap does not modify `f(x)`.

#### Right Identity Law:

- `monad.flatMap(y —> Monad.of(y))` is equivalent to `monad`.
- Basically that `.of()` should behave like an identity.

#### Associative Law:

- `monad.flatMap(x —> f(x)).flatMap(x —> g(x))` is equivalent to `monad.flatMap( x —> f(x).flatMap(x —> g(x)) )`.
- Aka same result regardless of how it is composed.

## Functor

- Has two methods `of` and `map`, does not carry side information.

### Functor Laws

#### Preserving Identity

- `functor .map(x —> x)` is same as `functor`

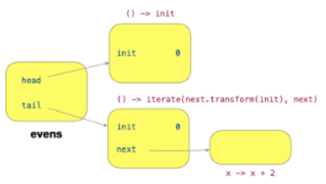
#### Preserving Composition

- `functor .map(x —> f(x)).map(x —> g(x))` is same as `functor .map(x —> g(f(x))`

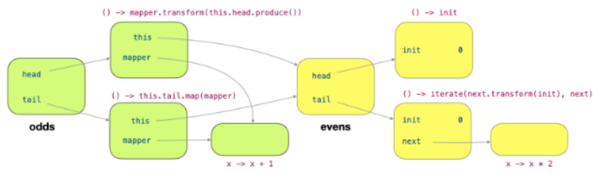
## InfiniteList

- Lazy evaluation allows us to delay the computation that produces data until the data is needed. This powerful concept enables us to build computationally-efficient data structures. List with a possibly infinite number of elements.
- we can delay a computation by using the Producer functional interface (or anything equivalent).

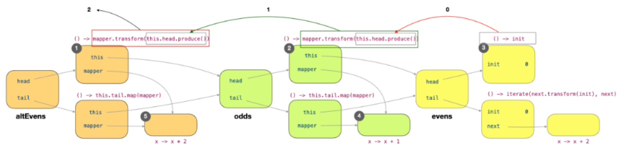
```
InfiniteList<Integer> evens =
  InfiniteList.iterate(0, x —> x + 2);
```



```
InfiniteList<Integer> odds = evens.map(x —> x +
    1);
```


() -> mapper.transform(this.head.produce())
this
mapper
head
tail
odds
() -> this.tail.map(mapper)
this
mapper
x -> x + 1
() -> init
init 0
head
tail
evens
() -> iterate(next.transform(init), next)
init 0
next
x -> x * 2

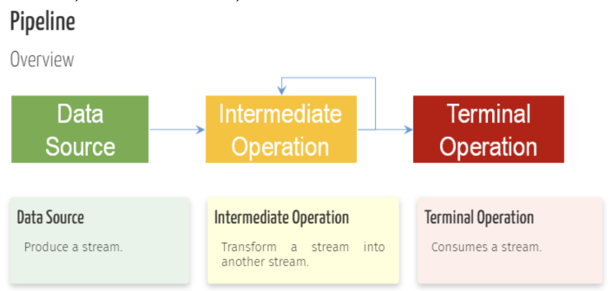
```
InfiniteList<Integer> altEvens = odds.map(x —> x
    * 2);
```


2
1
0
() -> mapper.transform(this.head.produce())
1
this
mapper
head
tail
altEvens
() -> this.tail.map(mapper)
this
mapper
5
x -> x * 2
() -> mapper.transform(this.head.produce())
2
this
mapper
head
tail
odds
() -> this.tail.map(mapper)
this
mapper
4
x -> x + 1
() -> init
3
init 0
head
tail
evens
() -> iterate(next.transform(init), next)
init 0
next
x -> x + 2

## 10. Streams

### Streams Pipelines

- **Source, Intermediate, Terminal**



- **consumed only once**: Limitation, `IllegalStateException` if consumed again.
- Makes code more declarative compared to how to do it, more succinct, less bug-prone.

## Parallel Streams

-