TYPES: S <: T iff a piece of code written for variables of type T can also be safely used on variables of type S.
- Widening conversion => a type S can be put into a variable of type T if S <: T
- Narrowing conversion require typecasting
- Reflexive and transitive
- S instanceof T returns true if S <: T

Primitive types:
```
byte <: short <: int <: long <: float <: double
char <: int
```
LSP: if S <: T then
- any property of T (includes fields, methods) should also be a property of S
- an object of type T can be replaced by an object of type S without changing some desirable property of the program
- Violation – subclass changes behavior of superclass (e.g. after subclass override, change from return A, B, C grade to return S, U grade)
```
Circle c = new ColouredCircle(p, 0, red);
// ColouredCircle <: Circle
```
CTT Circle, **RTT** ColouredCircle

OOP PRINCIPLES
- **encapsulation:** *composite data type* allows us to group *primitive types* together, give it a name to become a new type, and refer to it later.
- **abstraction:** *Abstraction barrier* lets us hide info & implementation. private attributes, public methods
- **inheritance:**
"is-a" relationship: use extends (subtyping)
"has-a" relationship: use composition
- **polymorphism:** (many forms)

To harness the Power Requires: *subclass* that inherit from superclass, *method overriding* for alternate forms, common *superclass* that has the given method.
Dynamic binding – method invoked is determined at runtime.
Method overriding – same method signature (method name, type + number + order of arguments) BUT different method descriptor (method signature + return type)
Method overloading (static polymorphism) – same method name, different parameter types or number of parameters

ABSTRACT CLASSES
- an abstract class cannot be instantiated. Instead, we instantiate classes (including anonymous ones) that extend from it.
- an abstract class can't be final or static (won't compile, because prevents inherit)
- note that private/final/static abstract fields are allowed
- an abstract class can have concrete and/or abstract methods
- an abstract method can't be private, static, or final (will not compile, because prevents overriding by subclass) BUT it's not automatically declared public.
- a concrete class cannot have abstract methods. Only abstract classes can.

INTERFACE ("can-do")
- methods are public and abstract by default
- concrete classes implementing the interface have to implement the body of ALL the methods
- if class C implements interface I, then C <: I
- a class can implement multiple interfaces
- an interface can extend multiple interfaces
- an interface cannot implement another interface (implementing requires defining methods, which interfaces don't do), and a class cannot extend an interface.
- default methods allowed in interfaces - allow us to add new methods to an interface that are auto-available in implementations, preserving backward compatibility

TELL, DON'T ASK:
- e.g. Client should *tell* a Circle object what to do (compute circumference), instead of *asking* "What is your radius?" to get the value of a field, then perform the computation on the object's behalf.

RAW TYPES
Generic type used without type arguments, only acceptable as an operand of instanceof

STACK AND HEAP
- Stack contains call frames and all the local variables, including parameters. Last-In-First-Out (like a stack of books)
- Heap contains dynamically created instances. New object is created whenever keyword new is used. Object in heap contains class name, instance fields and the respective values, and captured variables.
- Arrows represent pointers from a variable to instances
- Metaspace contains class fields (i.e. static fields)
- Method area stores code for methods

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

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

  public static <T> Lazy<T> of(Producer<T>
      producer) {
    return new Lazy<>(producer);
  }

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

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


Lazy<Integer> lazyInteger = Lazy.of(() -> 3);
Lazy<Integer> newValue = lazyInteger.map(x ->
    2 * x);
int result = newValue.get();
```

WRAPPER CLASS
- immutable (once u create an object, it can't be changed), less efficient memory-wise (cost of creating objects)
```
Integer i = new Integer(2);
int j = i.intValue();
```
- auto-boxing – primitive to wrapper (e.g. int to Integer)
- opposite is called unboxing
```
int i = 1;       // i is an int
Integer j = i;   // j is an Integer
int k = j;       // k is an int
```

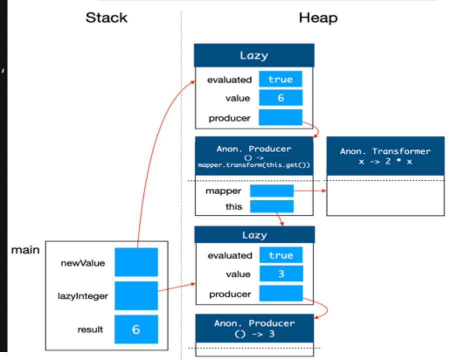
MODIFIERS
private – only within class
default – only within class or package
protected – within class or package, outside package by subclass only
public – everywhere

final variable – can only be assign once
final class – cannot be inherited from
final method – cannot be overridden

GENERICS
- allow classes/methods (that use reference types) to be defined without resorting to using Object type
- ensures type safety - binds a generic type to a specific type at compile time
- errors will be at compile time instead of run time
- 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
```
public static <T> boolean contains(T[] arr, T obj) {...}
// to call a generic method:
A.<String>contains(strArray, "hello");
```
- type parameter <T> is declared *before* the return type
- bounded type parameter: public <T extends Comparable<T>> T foo(T t) { ... }

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



CASTING (only cast when can prove it's safe)
```
// Circle <: Shape <: GetAreable
GetAreable findLargest(GetAreable[] array) { ... }
GetAreable ga = findLargest(circles);          // ok
Circle c1 = findLargest(circles);              // error
Circle c2 = (Circle) findLargest(circles);     // ok
```

VARIANCE
Let C(T) be a complex type based on type T. C is:
- *covariant* if S <: T implies C(S) <: C(T)
- *contravariant* if S <: T implies C(T) <: C(S)
- *invariant* if C is neither co- nor contravariant

Java array is covariant - S <: T => S[] <: T[]

TYPE ERASURE
- at compile time, type parameters are replaced by Object or the bounds (e.g. T extends Shape is replaced by Shape) – below shows before & after
```
Integer i = new Pair<String,Integer>("x", 4).foo();
```
```
Integer i = (Integer) new Pair("x", 4).foo();
```
- *heap pollution*: variable of a parameterized type refers to an object not of that parameterized type
- heap pollution can cause ClassCastException
- arrays are *reifiable* (full type info is available during run-time), but generics are not, due to type erasure. That's why they don't mix well.
- generic array *declaration* is fine but generic array *instantiation* is not.

SUPPRESS WARNINGS
- most limited scope, only if sure it won't cause type error later, must add note as comment, can only apply to declaration (assignment cannot)
- suppressed: "required: T[] found: Object[]"
```
@SuppressWarnings("unchecked")
T[] a = (T[]) new Object[size];
this.array = a;
```

WILDCARDS
Upper-bounded: <? extends T> covariant
- if S <: T , then A<? extends S> <: A<? extends T>
- for any type T, A<T> <: A<? extends T> <: A<?>

Lower-bounded: <? super T> contravariant
- if S <: T , then A<? super T> <: A<? super S>
- for any type S, A<S> <: A<? super S> <: A<?>

Unbounded: <?>
- Array<?> is the supertype of all generic Array<T>

PECS
Producer extends: if you need to produce T values, use List<? extends T>
Consumer super: if you need to consume T values, use List<? super T>
Both producer and consumer: unbounded <?>

IMMUTABILITY (alt: only class final, clone() method)
- immutable class means an instance can't have *visible changes* outside its abstraction barrier
- ease of understanding (no need trace methods)
- saves space (all references shared until instance needs to be modified, which will create a new copy)
- enables safe sharing of objects and internals (no need worry about aliasing)
- enables safe concurrent execution
- explicitly reassign to update something final
```
final class Circle {
  final private Point c;
  final private double r;
   :
  public Circle moveTo(double x, double y) {
    return new Circle(c.moveTo(x, y), r);
  }
}
```

EXCEPTIONS (passed up call stack until caught)
```
try {
  new Circle(new Point(1, 1), 0);
  // everything afterwards is skipped
  System.out.println("This will never reach");
} catch (IllegalException e) {
  // runs if there is an exception
} finally {
  // always runs if haven't return yet
}
// after exception caught, here runs if haven't return yet
```
throw causes method to immediately return
```
import java.lang.IllegalArgumentException
public Circle(Point c, double r) throws IllegalArgumentException {
    if (r < 0) {
      throw new IllegalArgumentException("radius cannot be negative."
    }
    // anything from here will not run if r<0
}
```
- programmer has no control over checked exceptions e.g. FileNotFoundException (must handle or program won't compile)
- unchecked exceptions e.g. NullPointerException are caused by programmer's errors. They are subclasses of RuntimeException.
- passing the buck: throw exception and let caller catch. But must ensure someone handles it gracefully, if not the details may be revealed to the user (bad)
- overriding a method that throws a checked exception must throw only the same, or a more specific, checked exception, NOT some other thing. (LSP)
- Don't catch all (Pokémon). Don't overreact and exit program (prevents resource cleaning). Don't use as control flow mechanism.
- try to handle implementation-specific behind abstraction barrier so details aren't leaked
- Error class ends program (e.g. OutOfMemoryError)

TYPE INFERENCE
- ensures type safety – compiler can ensure List<myObj> holds objects of type myObj at compile time instead of runtime
- Find subtyping relation then solve for T.
- Type 1 <: T <: Type 2 → T is inferred as Type 1
- Type 1 <: T → T is inferred as Type 1
- T <: Type 2 → T is inferred as Type 2

Diamond operator <> type inference is auto.
E.g. the following are equivalent:
```
Pair<String,Integer> p = new Pair<>();
Pair<String,Integer> p = new Pair<String,Integer>();
```
- Generic methods type inference is auto.
E.g. A.contains() NOT A<>.contains()
- Constraints for type inference:
Target typing → type of expression i.e. T <: Shape
Type parameter bounds → <T extends GetAreable>
Parameter bounds →
Array<Circle> <: Array<? extends T> , so T :> Circle
```
public static <T extends GetAreable> T findLargest(Array<? extends T> arr
Shape o = A.findLargest(new Array<Circle>(0));
```
- If no T that satisfy all, then compilation error. E.g. no soln:
```
public <T extends Circle> T foo(Seq<? extends T> seq)
ColoredCircle c = foo(new Seq<GetAreable>());
```
1. return type of foo must be a subtype of ColouredCircle and thus T <: ColouredCircle.
2. T is also a bounded type parameter. T extends Circle tells us that T <: Circle.
3. Our method argument is of type Seq<GetAreable> and must be a subtype of Seq<? extends T>, meaning T must be a supertype of GetAreable (i.e. GetAreable <: T <: Object)

VARARGS

- pass a variable number of arguments of the same type as an array of items
- public void of (T... items) {} → items will be T[]
- @SafeVarargs if T is a generic type
- final or static methods/constructors

NESTED CLASSES

- can access all fields and methods within its container class
- if static nested, associated with containing class, NOT an instance. Can only access static fields/methods of containing class.
- Inner class (non-static nested) can access all fields/methods of containing class.
- *Qualified this* reference is prefixed with enclosing class name, to differentiate between *this* of inner class and *this* of enclosing class.

```
class A {
  private int x;

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

LOCAL CLASSES

- Classes declared within a block {} or method, are known as *local classes*
- Like a local variable, local class is scoped within the method.
- Like a nested class, local class has access to variables of enclosing class through qualified this reference. Additionally, it can access the local variables of the enclosing method. The local class makes a *copy* of the local variables in itself (we call this variable capture).
- variables accessed must be declared final or *effectively final* (cannot be re-assigned after initialization. Note it is still possible to modify such a variable through mutation). If not, a compilation error will occur.
- when a method returns, all local variables of the method are removed from the stack

ANONYMOUS CLASSES

Format: new Constructor(arguments) { body } or new (className implements someInterface) (arguments) { body }

- can't implement more than one interface
- can't extend a class and implement an interface at the same time
- can extend from one class / one interface / a generic type
- () required with no arguments inside if implementing an interface
- can't have a constructor for an anonymous class within the body
- Captures variables of enclosing scope; same rules as for local classes.

FUNCTIONS

- pure functions:

1. no side effects (no print / write to file / change value of arguments / throw exceptions / change other variables)
2. every input maps to an output in the codomain (note: null is not within codomain)
3. deterministic; doesn't depend on external variables (given the same input, must produce the same output every single time)
4. deterministic → *referential transparency* - if f(x) = y, then any y can be substituted with f(x)
5. must return a value (cannot be void)

- if class is immutable, its methods are pure functions (they won't have side effects like updating fields of an instance or computing values using fields of an instance)

FUNCTIONS AS FIRST CLASS CITIZENS

- use local anonymous class

```
Transformer<Integer, Integer> square = new Transformer<>() {
  @Override
  public Integer transform(Integer x) {
    return x * x;
  }
};
```

- interface with only one abstract method is a @FunctionalInterface (e.g. Comparator, Transformer)
- benefit: no ambiguity about which method is being overridden by implementing subclass

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

- removing boilerplate code example
- the following are equivalent (and allowed) :

```
Transformer<Integer, Integer> incr = new Transformer<>() {
  @Override
  public Integer transform(Integer x) {
    return x + 1;
  }
};
Transformer<Integer, Integer> incr = (Integer x) -> { return x + 1; };
Transformer<Integer, Integer> incr = (x) -> { return x + 1; };
Transformer<Integer, Integer> incr = x -> { return x + 1; };
Transformer<Integer, Integer> incr = x -> x + 1;
```

METHOD REFERENCE

- Existing method as first class citizen
- Specified using the double colon ::
- We can use method references to refer to:

1. a static method in a class
className::staticMethodName
2. an instance method of a class or interface
instanceName::instanceMethodName
type::methodName (e.g. String::concat)
3. a constructor of a class
className::new

```
Box::of          // x -> Box.of(x)
Box::new         // x -> new Box(x)
x::compareTo     // y -> x.compareTo(y)
```

```
Transformer<T, U> foo = A::foo;
// (x, y) -> x.foo(y)      A is a type, foo is an instance of that type
// (x, y) -> A.foo(x, y)   A is an instance, foo is an instance method
```

- When compiling the last example, Java performs type inferences to find the method that matches the given method reference.
- Compilation error if multiple matches or there is ambiguity in which method matches

CURRIED FUNCTIONS

- instead of having a function that takes in two arguments, we can have a function that takes in one argument, and that returns another function to accept the second argument
- this is known as a *higher-order function*
- currying translates a general *n*-ary function to a sequence of *n* unary, "curried" functions
- allows us to *partially apply* a function first.
- useful if one of the arguments does not change often / is expensive to compute.
- We can save partial results of a function and continue applying it later. We can dynamically create functions as needed, save them, and invoke them later.

LAMBDA

```
Point origin = new Point(0, 0);
Transformer<Point, Double> dist = origin::distanceTo;
```

- here, variable origin is captured by lambda expression dist
- lambda expression stores the data from the environment where it is defined
- *closure* → a construct that stores a function together with the enclosing environment
- makes our code cleaner (fewer parameters to pass around, less duplicated code)
- can separate logic to do different tasks in a different part of our program more easily
- lambda as cross-barrier state manipulator (e.g. use of map and filter in Box)
- lambda as delayed data (lazy evaluation)

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

```
i = 4;
Task print = () -> System.out.println(i);
Producer<String> toStr = () -> Integer.toString(i);
```

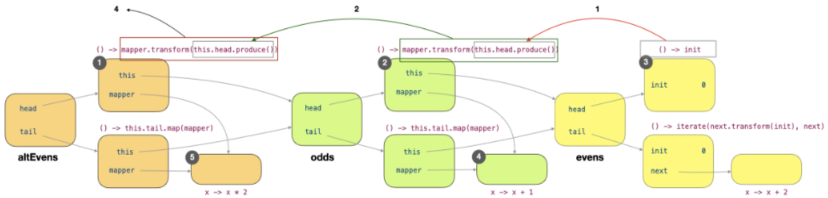
STREAMS

- can only be consumed once
- IllegalStateException if consumed again

PARALLEL STREAMS

- add .parallel() (lazy operation) anywhere between data source and terminator, or use parallelStream()
- when each element is processed individually without depending on other elements, the computation is *embarrassingly parallel*
- can't be stateful (result cannot depend on any state that may change during stream execution)
- can't interfere with stream data (stream operations may not modify source of stream during execution of terminal execution. Throws ConcurrentModificationException)
- can't have side effects

```
InfiniteList<Integer> evens = InfiniteList.iterate(0, x -> x + 2);
InfiniteList<Integer> odds = evens.map(x -> x + 1); // 1, 3, 5, ...
InfiniteList<Integer> altEvens = odds.map(x -> x * 2); // 2, 6, 10, ...
```



MONAD

- of method to initializes value and side info
- flatMap method to update value and side info
- creating with of and chaining with flatMap makes monad classes "well-behaved"

Container	Side info
Maybe<T>	*Option type* - Value might be there (i.e. Some<T> or not (i.e. None)
Lazy<T>	Value has been evaluated or not
Loggable<T>	Log describing the operations done on the value

LAWS OF MONAD

1. Left identity law
Monad.of(x).flatMap(x -> f(x)) must be same as f(x)
2. Right identity law
monad.flatMap(x -> Monad.of(x)) must be same as monad
3. Associative law
Monad.flatMap(x -> f(x)).flatMap(x -> g(x)) must be same as Monad.flatMap(x -> f(x).flatMap(x -> g(x)))

FUNCTOR

- ensures lambdas can be added sequentially to the value, without worrying about side info

1. Preserving identity law
functor.map(x -> x) must be same as functor
2. Preserving composition law
functor.map(x -> f(x)).map(x -> g(x)) must be same as functor.map(x -> g(f(x)))

- Note that our classes from cs2030s.fp, Lazy<T>, Maybe<T>, and InfiniteList<T> are functors.

PARALLEL STREAMS (cont'd)

- Can avoid side effects by using any of these:

1. thread-safe data structure (e.g. CopyOnWriteArrayList)
2. .collect(Collectors.toList()) method at the end
3. .toList() at the end (list in same order as stream)

- To run reduce in parallel (reduce each sub-stream then combine results with combiner), RULES:

1. combiner.apply(identity, i) must be equal to i
2. combiner and accumulator must be associative → order of applying must not matter
3. combiner and accumulator must be compatible → combiner.apply(u, accumulator.apply(identity, t)) must equal to accumulator.apply(u, t)

- Parallelizing doesn't always improve performance because creating thread to run incurs overhead
- Ordered collections (e.g. List, arrays), iterate, of → create ordered streams
- Unordered collections (e.g. Set), generate → create unordered streams
- Parallel version of findFirst, limit, and skip can be expensive on ordered stream as need to coordinate
- distinct and sorted preserve order (*stable*) for *finite* streams only

THREADS (java.lang.Thread)

- Normal java program – method blocks until it returns (synchronous programming model)
- Thread is a single flow of execution
- new Thread constructor takes in a Runnable
- Runnable is a functional interface with a method run() that takes in no parameter and returns void
- .start() makes thread begin execution. Given lambda expression runs. Returns immediately
- Thread.currentThread() returns reference of current running thread. Use Thread.currentThread().getName() to find name of current running thread
- Thread.sleep(ms) pauses execution of current thread
- .isAlive() returns boolean (whether thread is running)
- Program exits only after all created threads finish run.
- run in parallel() and print names → 'main' followed by 'ForkJoinPool.commonPool-worker-#' (number of unique # is number of concurrent threads running)
- if not call parallel() then only 'main' is printed # times

The CompletableFuture Monad

java.util.concurrent.CompletableFuture

- to instantiate, there are several ways:

1. completedFuture method .completedFuture(thing)
2. .runAsync(Runnable lambda expression) → returns a CompletableFuture<Void> that completes when the given lambda (Runnable) finishes
3. .supplyAsync(Supplier<T> lambda expression) → returns a CompletableFuture<T> that completes when the given lambda (Supplier) finishes
4. rely on other CompletableFuture instances using .allOf(...) or .anyOf(...) → take in a variable number of CompletableFuture instances, returns a CompletableFuture<Void> that completes when all/any of the given CompletableFuture complete.

- chaining in same thread (use Async if want different)
.thenApply(res -> f(res)) → analogous to map
.thenCompose(res -> CF) → analogous to flatMap
.thenCombine(CF, (x, y) -> ...) → analogous to combine
- .get() returns result. Synchronous – blocks until CF done. Throws InterruptedException and ExecutionException
- .join() is like get() but won't throw checked exception May throw CompletionException which is unchecked.
.handle((result, exception) -> (exception == null) ? result : somethingElse) either exception or result is null here

THREAD POOL consists of a *collection of threads*, each waiting for a task to execute, and a *collection of tasks* to be executed (usu *shared queue*)

FORK AND JOIN (parallel divide-and-conquer model)

- task is an instance of abstract class RecursiveTask<T>
- RecursiveTask<T> supports fork(), which submits a smaller version of the task for execution, and join(), which waits for the smaller tasks to end & return.
- RecursiveTask<T> has an abstract method compute() which the client has to define.

ForkJoinPool - Each thread has a deque (double ended queue that behaves like both a stack and queue) of tasks. Idle thread will compute() task at head of its deque. If deque empty, thread will compute() task at tail of another thread's deque (*work stealing*). When fork() called, caller adds itself to *head* of executing thread deque. The *most recently* forked task will be executed next. When join() called: if subtask *has not been executed*, compute() the subtask; if subtask *completed* (stolen), read result and return; if subtask stolen and *currently being executed by another thread*, find another task to do.