OOP

Principles of OOP

- **Abstraction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**

OOP Principle #1: Abstraction

Abstraction reduces complexity by filtering out unnecessary details.

- **Data Abstraction**
  - *Circle* only needs to know about *Point*, not about *x, y (i.e don't need to know about implementation)*
- **Functional Abstraction**
  - *Circle* can use *Point::distanceTo()*, does not need to know how *distanceTo()* is calculated

Abstraction Barrier

Interaction between two object is viewed as communiation across an abstraction barrier.

- Provides a seperation between the implementation of an object and how it's used by a client.
- Client does not need to know the implementation of the class / methods
- Implementer is free to change the representation without affecting the client

OOP Principle #2: Encapsulation

Encapsulation refers to the bundling of data with methods that operate on that data, and restricted of direct access to some of an object's components.

Aspects of Encapsulation

- **Packaging**
  - OOP requires that both data and functionality be packaged into an appropriate class. The data is not accessed directly, it is accessed through functions inside the class.
- **Information Hiding**
  - Prevent client access to lower level details of the implementer. (Using access modifiers – **private**)

Tell-Don't-Ask

- Tell an object what to do, don't ask an object for data – avoid using getters.

***Abstraction & Encapsulation enables objects to model a "HAS-A" relationship.**

OOP Principle #3: Inheritance

In OOP, inheritance is the mechanism of basing on object upon another object, retaining similar implementation.

Ways to extend a class

Design #1: As a stand-alone class

Design #2: Using Composition (**has-a** relationship)

Design #3: Using **Inheritance** (**is-a** relationship)

- The child/sub class (FilledCircle) **extends** from the parent/super class (Circle).

We should only inherit another class if it is substitutable for the super-class. (LSP)
If class S is a sub-class of T, then S is substitutable for T.

- Let Circle be a sub-class of Point, Circle is not substitutable for Point! (Makes no sense to compute Euclidean distance between two circles)

OOP Principle #4: Polymorphism

Polymorphism is the ability of an object to take on many forms. It allows us to change how existing code behaves, without changing a single line of the existing code (or even having access to the code).

Overloading Methods

- Overloading is a form of polymorphism in OOP, allowing objects or methods to act in different ways.
- Overloading happens when you have two methods with the same name, but **different method signatures**.

Overriding Methods

- Overriding is another form of polymorphism.
- Enables a child class to provide different implementation for a method already defined or implemented.
- Overriding happens when you override a method with the **same method name and signature (in sub-class)**. It must also have the **same or higher accessibility**.
- Return type cannot be more general than that of the overridden method

***Inheritance & Polymorphism enables objects to model a "IS-A" relationship.**

***Inheritance & Polymorphism supports substitutability**

Typing, Compile Time vs Runtime

```
Circle c = new FilledCircle(1.0, Color.BLUE);
```

- Variable c has a **compile-time type of Circle**
  - Restricts the methods it can call during compilation. (Can only access Circle methods, but not FilledCircle methods).
- Variable c has a **run-time type of FilledCircle**
  - The type of the object that the variable is pointing to
  - Determines the actual method called during runtime
    - FilledCircle::toString(), not Circle::toString()

Run-Time vs Compile-Time

Runtime only

- Type Checking [Checking if value matches the type of the variable it is assigned to]
- Late (Dynamic) Binding [Determine which instance method to call depending on the type of a reference object]

Compilation time only

- Type Inference [Inferring the type of a variable whose type is not specified]
- Type Erasure [Replacing a type parameter of generics with either Object or its bound]
- Early (Static) Binding [Determine which overloaded method to call during compilation]

Both runtime & compilation time

- Type Casting [Converting the type of one variable to another]
  - *Happens during compile time, checked during run time*
- Accessibility checks [Checking if a class has an access to a field in another class]

Type Erasure

To implement generics, the Java compiler applies type erasure to:

- **Replace all type parameters in generic types with their bounds or Object if the type parameters are unbounded**.
- Insert type casts if necessary to preserve type safety.
- Generate bridge methods to preserve polymorphism in extended generic types.

Type erasure ensures that no new classes are created for parameterized types

```
abstract class A {
    //NOT Allowed!! Both are erased to List<Object> / List
    abstract void test(List<Integer> list);
    abstract void test(List<String> list);
}
```

Early vs Late Binding (Related to Polymorphism)

Dynamic (Late) Binding

The exact method to invoke is not known until runtime. (Object::equals or Circle::equals?)

Static (Early) Binding

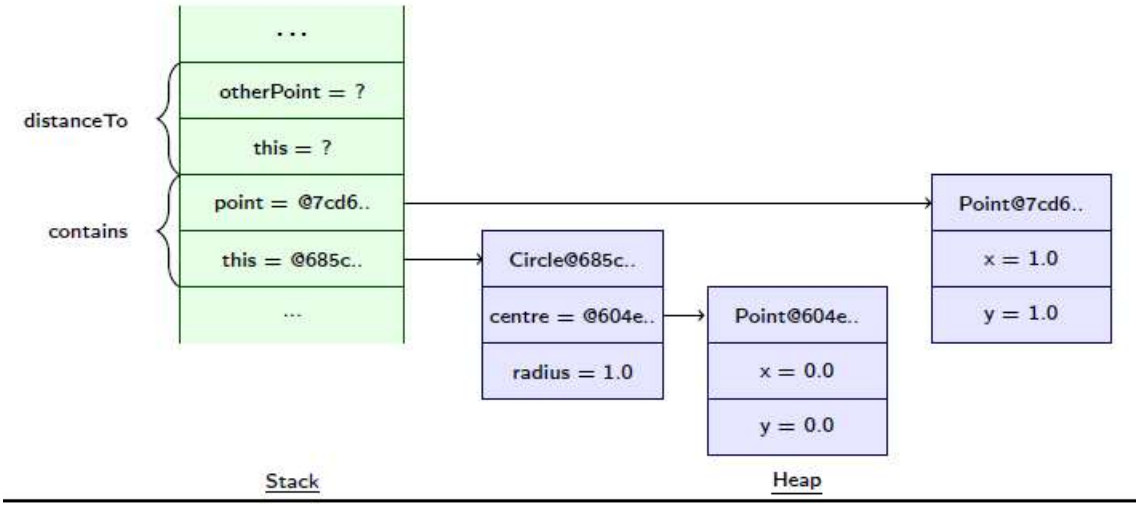
The compile-time type decides which **overloaded** method to call during compilation.

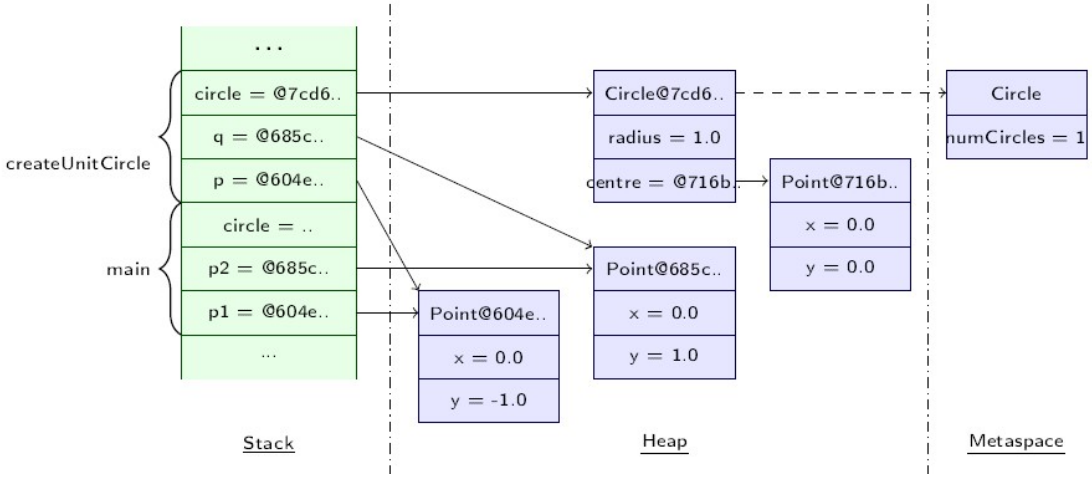
```
interface I1 {
    void bar(X x);
    void bar(I1 i);
}

interface I2 {
    void bar(Y y);
}

class X implements I1 {
    public void bar(X x) { System.out.print(1); }
    public void bar(I1 i) { System.out.print(2); }
}

class Y extends X implements I1, I2{
    public void bar(X x) { System.out.print(3); }
    public void bar(I1 i) { System.out.print(4); }
    public void bar(Y y) { System.out.print(5); }
}
```

What is the output of the following code excerpt?

```
X x   = new X();
X xy  = new Y();
Y y   = new Y();
I1 i  = new Y();

i.bar(y);        //3   I::bar(Y) -> I::bar(X) –run time> Y::bar(X)
x.bar(y);        //1
y.bar(xy);       //3
x.bar(x);        //1
```

Java Memory Model





Stack

- **Stores local variables and call frames**
- LIFO stack for storing activation records of method calls
- Primitive types and object references are allocated and stored in stack
- Method local variables are stored here

- **Heap**
  - **Stores dynamically allocated objects**
  - Stores object upon invoking **new**
  - **Enums** are treated as Objects in Java, therefore it is stored in the heap
- **Metaspace (Non-heap)**
  - For storing loaded classes and other metadata
  - Static fields are stored in metaspace

Abstract Class and Interface

Concrete Class is the actual implementation.

Interface is a contract specifying the abstraction between

- what the client can use, and
- what the implementer should provide

Abstract class is a trade off between the two, i.e. partial implementation of the contract

- typically **used as a base class**

"Impure" interfaces

- Since Java 8, default methods with implementations can be included into interfaces

abstract classes

E.g. Does not make sense to instantiate a FilledShape object.

```
jshell> new FilledShape(Color.BLUE).getArea()
$.. ==> -1.0
```

Abstract class with abstract methods – these will be implemented in the child classes

- Abstract classes **cannot be instantiated**
- Abstract classes **can be subclassed**
- **Child class can only inherit from one parent class**
    - `class Circle extends FilledShape, Scalable{} //Not allowed`
    - If parents have the same method signature but different implementation, it becomes ambiguous which method should be invoked

interface as a contract

Each interface is a contract to that **specifies methods to be defined in the implementation class**.

- **Classes can implement one or more interfaces**
    - `class Circle implements Shape, Scalable{} //Allowed`
    - If C implements A and B that both specify f(), then an overridden method in C would satisfy both contracts. There is also no ambiguity which method to invoke as the implementation is in C.
- **Methods in interfaces are implicitly public**
    - **Overriding methods must be public (Pre Java8)**

"Sub-classing" Arrays

```
Shape[] shapes = new Circle[]{new Circle(), new Circle()};

shapes[0] = new Rectangle();
```

Program will compile fine, but will encounter **ArrayStoreException** during runtime due to **Heap Pollution**.

(Attempting to put a Rectangle into a Circle[] during run time)

Design Principles (SOLID)

- **S**ingle Responsibility Principle
- **O**pen-Closed Principle
- **L**iskov Substituition Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle
-

Single Responsibility Principle

A class should do one thing and therefore it should have only a single reason to change Only one potential change (database logic, logging logic, and so on.) in the software's specification should be able to affect the specification of the class.

Open-Closed Principle

Classes should be open for extension but closed to modification. Modification means changing the code of an existing class, and extension means adding new functionality. We should be able to add new functionality without modifying the client's implementation. Whenever we modify existing code, we are taking the risk of creating potential bugs.

Liskov Substitution Principle

The substitutability principle says that **"if S a subclass of T, then an object of type T can be replaced by that of type S without changing the desirable property of the program**".

Interface Segregation Principle

The principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do no need.

Dependency Inversion Principle

DIP states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions. High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Java Collections

Java API provides collections to store related objects.

- **Abstraction:** methods that organize, store, and retrieve data
- **Encapsulation:** how data is being stored is hidden (encapsulated)

Common collections*: Collection (root interface), Set, List, Map, Queue*

Type parameter T is replaced with type argument to indicate the type of elements stored.

Eg. *ArrayList<String>* is a **parameterized type**

- Collection of type <T> contains references to objects of type T, or sub-type of T.
- Lists are ***mutable*** data structures

Only reference types allowed; primitives need to be auto-boxed/unboxed

```
ArrayList<Integer> list = new ArrayList<Integer>()
list.add(1)                    //auto-boxing      int -> Integer
list.add(new Integer(2))       //explicit boxing
int x = list.get(0);           //auto-unboxing    Integer -> int
```

ArrayList<E> is a sub-type of List<E>, so below is valid.

- `List<Integer> list = new ArrayList();`

Unlike arrays, given S is a subtype of T, **ArrayList<S> is not a sub-type of ArrayList<T>.**

- `ArrayList<Object> list = new ArrayList<Integer>();  //Invalid`
- Addresses the problem of **heap pollution**

Natural Ordering

Natural ordering of elements of type T is defined by implementing the compareTo method (of the Comparable<T> interface). There can only be one compareTo method, hence one natural order. **`public int compareTo(T other)`**

Packages

Java adopts a **package** abstract mechanism that allows the grouping of relevant classes/interfaces together under a namespace (like java.lang).

- private (visible to the class only)
- default (visible to the package)
- protected (visible to the package and all sub-classes)
- public (visible to the world)

static/enum/final, Assertions

A **static** field is class-level member declared to be shared by all objects of the class

- Used for defining aggregated data, e.g. number of Circles
- Used for defining constants, e.g. EPSILON
- Used for factory methods, e.g. Logger.of(…)

Static methods belong to the class instead of an object.

- **No overriding** as static methods resolved at compile time

An **enum** is a special type of class used for defining constants.

- Each constant of an enum type is an instance of the enum class and is a field declared with **public static final**
- Constructors, methods, and fields can be defined in enums

The **final** keyword can also be applied to methods or classes.

- Use the final keyword to explicitly prevent Inheritance
  - `final class Circle { }`
- To allow inheritance but prevent overriding
  - `final double getArea() { }`

Assertions are used to identify bugs during program development (as compared to exceptions which are used to handle user mishaps)

- Preconditions are assertions about a program's state when a program is invoked
- Postconditions are assertions about a program's state after a method finishes
- `assert Boolean_expression;`
- `assert Boolean_expression : string_expression;`

Exception Handling

Exceptions are used to track reasons for program failure

- Eg. Filename missing or misspelt, file contains non-numerical value…
- Only create your own exception if there is a good reason to do so

Handling Exceptions

Method #1: **throws** the exception out

- `public static void main(String[] args) throws FileNotFoundException`

Method #2: **handle** the exception

- **try/catch** -> try block encompasses the business logic, catch block encompasses exception handling
- Catch the most specific exceptions first

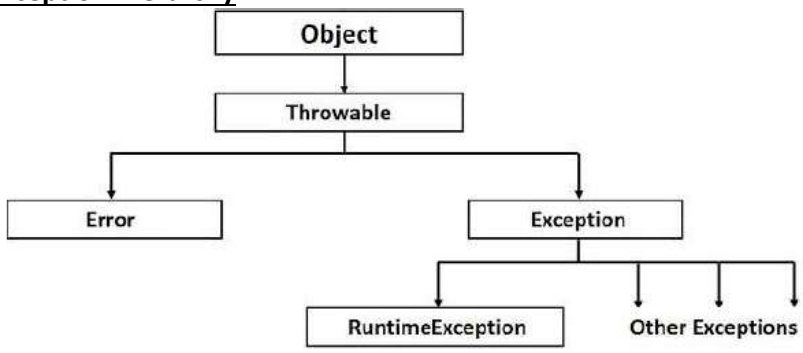
Checked Exceptions

A checked exception is one that the programmer should actively anticipate and handle (All checked exceptions should be caught (catch) of propagated (throw)
E.g. *FileNotFoundException might be thrown when opening a file*

Unchecked Exceptions

An unchecked exception is one that is unanticipated, usually the result of a bug
E.g. ArithmeticException when trying to divide by zero

Exception Hierarchy



- Unchecked exceptions are sub-classes of ***RuntimeException***
- All Errors are also unchecked
- When overriding a method that throws a checked exception, **the overriding method cannot throw a more general exception** (**but the overriding method can choose to not throw an exception!**)
- Avoid catching **Exception**, aka Pokemon Exception Handling
- **Handle exceptions are the appropriate abstraction level**, do not just throw and break the abstraction barrier

Generics

Generic typing is also known as parametric polymorphism.

Type scoping

Type parameter can be scoped within a method

```
<T> List<T> arrayToList(T[] arr)      //scope of T is within method
```

static methods are also defined as generic methods

```
static <U> ImList<U> of(U[] array)    //scope of U is within the method
```

Java actually can infer the type using type inference mechanism. But for clarity, we specify the type.

```
A.contains(...);                 //Type inference => A.<String>contains(...)
A.<Integer>contains(...);        //Type witness -> Encouraged!
```

- Only reference types can be used as type parameter. (Integer is allowed, but int is not).

Cross-Barrier Manipulation

Where the client defines a functionality that is passed to the implementer for execution.

For example, we are required to supply the Comparator for the List to sort.

```
List::sort(Comparator<? super E> c)
```

Defining a Comparator as a concrete class

```
class IntComp implements Comparator<Integer> {
    @Override
    public int compare(Integer i, Integer j) {
       return j - i;
    }
}
```

Defining a Comparator using anonymous inner class

```
list.sort(new Comparator<Integer>() {
    @Override
    public int compare(Integer i, Integer j) {
        return j - i;
    }
});
```

Which part of the anonymous inner class is *really* useful?

- Interface name (Comparator) does not add any value
- Method name (compare) does not add any value
- Comparator is a SAM (Single Abstract Method) interface

Defining a Comparator using lambda

- `list123.sort((Integer i, Integer j) -> { return j - i; })`
- `list123.sort((i, j) -> j - i)` `//Inferred parameter type!`

Functional Interface – Interface with only a single abstract method (SAM).
Eg. Predicate<T>, Function<T,R>, BiFunction<T,U,R>

Lambdas can be used with functional interfaces.
Lambda Syntax: `(parameterList) -> {statements}`

- Methods can now be treated as values
  - Assign lambdas to variables
  - Pass lambdas as arguments to other methods
  - Return lambdas from ethods

Bounded Wildcards

Upper bounded wildcards: ? extends T

- **All possible type T and its sub-type**

Lower bounded wildcards: ? super T

- **All possible type T and its super-type**

Use bounded wildcards to restrict the types allowed.

```
class FastFood
class Burger extends Fastfood
class CheeseBurger extends Burger
```

```
void cook(List<Burger> burgers) { }

// ArrayList<Fastfood> cannot be converted to List<Burger>
cook(new ArrayList<Fastfood>());

//ArrayList<CheeseBurger> cannot be converted to List<Burger>
cook(new ArrayList<CheeseBurger>());

cook(new ArrayList<Burger>());
```

```
void cook(List<? extends Burger> burgers) { }

// ArrayList<Fastfood> cannot be converted to List<? extends Burger>
cook(new ArrayList<Fastfood>());

cook(new ArrayList<CheeseBurger>());
cook(new ArrayList<Burger>());
```

Type Variance

Let <: denote a sub-type (substitutability) relationship

Java **arrays** are **covariant,** **C <: S => C[] <: S[]**
E.g. Object[] o = new String[0];

Java **generics** are **invariant,** **C <: S , G<C> <: G<S> NOR G<S> <: G<C>**
E.g. List<Object> </: List<String>

Parameterized types are **covariant,** **ArrayList <: List => ArrayList<C> <: List<C>**

? extends is covariant **C <: B => G<C> <: G<? extends B>**
E.g. ArrayList<? extends Object> list
= new ArrayList<String>()

? super is contravariant **B <: F => G<F> <: G<? super B>**

E.g. ArrayList<? super String> list
= new ArrayList<Object>()

Given X <: Y <: Z

`List<? super Y>`

- **get() -> Returns an Object instance**
  - Since we cannot know which supertype it is, the compiler can only guarantee it will be a reference to an Object since Object is the supertype of any Java type.
- **add() -> Can add sub-types of Y -> add(X), add(Y)**
  - The list must contain Y and its sub-type. Therefore it is safe to add objects that is Y, or sub-type of it.

`List<? extends Y>`

- **get() -> Returns a Y instance**
  - The element in the list must be a sub-type of Y. The upper bounded type is Y (the top type it can be).
- **add() -> Cannot add anything!!**
  - The list might be referring to any type that is sub-type of Y. Since there is no way to tell what is the type, and we need to guarantee type safety (invariance), we cannot add anything to a List<? extends Y>.
    - List<Y> -> not safe to add(Z)
    - List<X> -> not safe to add(Y)
    - List<sub-type of X> -> not safe to add(X)

```
HashMap<? super X, ? extends Y> hashmap = new HashMap<X, Y>()     //OK
HashMap<? super X, ? extends Y> hashmap = new HashMap<Z, X>()     //OK
```

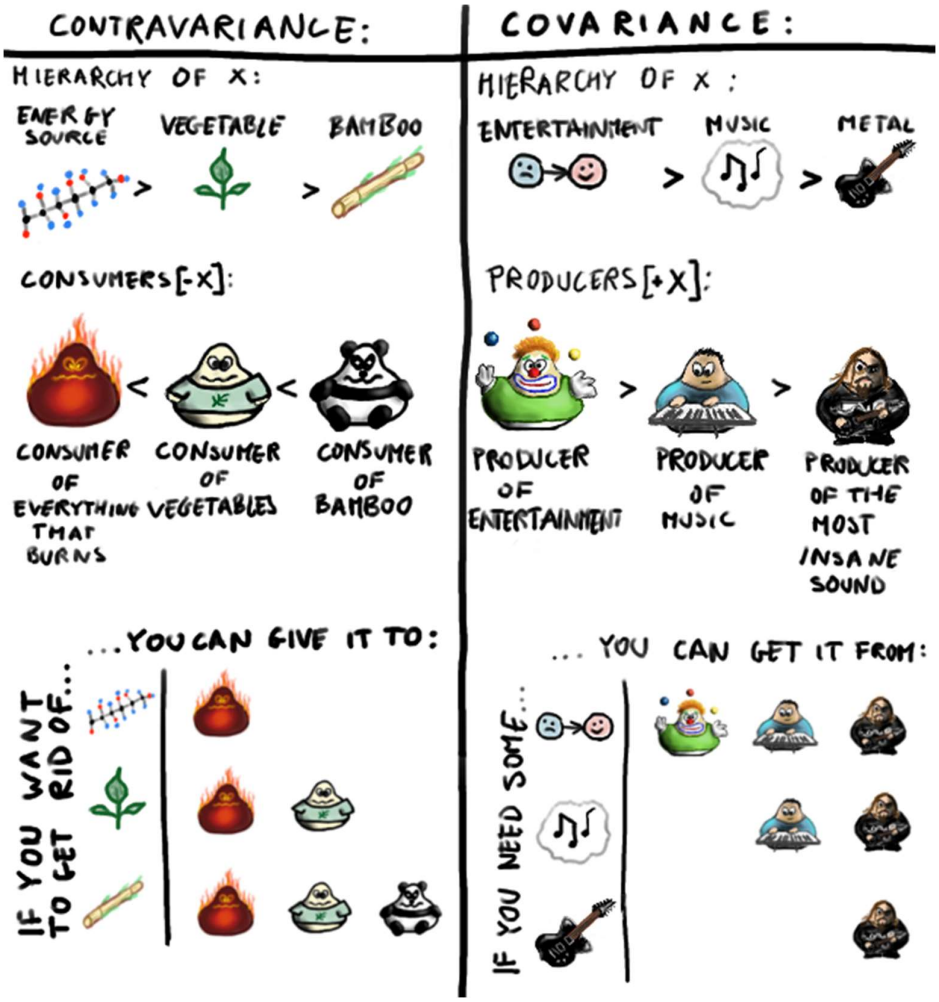
PECS (Producer Extends Consumer Super)

Covariant: Use extends to get items from a producer
Contravariant: Use super to put items into a consumer
Invariant: Use neither to get and put
***Do not use extends or super if it is both a producer and consumer**

"PECS" is from the collection's point of view. If you are only pulling items from a generic collection, it is a producer and you should use extends; if you are only stuffing items in, it is a consumer and you should use super.

`Stream<T> filter(Predicate<? super T> predicate)`
Filter is a consumer (consumes the stream to return a result). Therefore we should use super.

`<R> Stream<R> map(Function<? super T,? extends R> mapper)`
T consumes the stream (Consumer) -> use super.
R produces another element (Producer) -> use extends.



FP Contexts (Optional, Streams)

Optional

Context for Missing/Invalid values

- Rather than let the client handle the if/else branching associated with managing missing/invalid values, abstract the details of handling such values into **Optional**.
- **Tell-Don't Ask**: Avoid using `get(), isPresent(), isEmpty()`.

Stream

Context for Iteration/parallelism/infinite list

- No need to specify how to iterate through elements or use any mutable variables
- A **stream** is a sequence of elements on which tasks are performed; the stream pipeline moves the stream's elements through a sequence of tasks
- Stream elements within a stream *can only be consumed once*

Stream pipeline starts with a **data source**

- IntStream.range creates a stream of int elements

*IntStream is a stream of primitive *int* types, Stream<Integer> is a stream of *Integer* objects

Terminal Operations reduces the stream of values into a single value

- Eg. s*um(), reduce(), count()*

Intermediate Operations specify tasks to perform on a stream's elements

- Eg. *map(), flatmap()*
- Intermediate operations return a new stream made up of processing steps specified up to the point in the pipeline.

Lazy vs Eager Evaluation

Lazy Evaluation

Source/Intermediate operations use lazy evaluation. Lazy Evaluation does not perform any operations on stream's elements until a terminal operation is called.

Eager Evaluation

Terminal Operations use eager evaluation. The requested operation is performed as soon as it is called.

- **Lazy evaluation** allows us to work with **infinite streams** that represent an infinite number of elements. (**iterate(), generate()**)
- Intermediate operations, e.g. limit, can be used to restrict the total number of elements in the stream

Streams Parallelism

parallel() operation switches the stream pipeline to parallel

- Invoke anywhere between the data source and terminal
- **sequential()** switches off parallel operation

Avoid parallelizing trivial tasks.

- Creates more work in terms of parallelizing overhead
- Worthwhile only if the task is complex enough

Correctness of Parallel Streams

1. Must not interfere with data source

Lambda expressions in stream operations should not *interfere*. Interference occurs when the data source of a stream is modified while a pipeline processes the stream.

```
jshell> List<String> list = new ArrayList<>(List.of("abc","def","xyz"))
list ==> [abc, def, xyz]
jshell> list.stream().peek(str -> { if (str.equals("xyz")) list.add("pqr"); }).
...> forEach(x -> {})
| Exception java.util.ConcurrentModificationException
```

2. Preferably Stateless

Avoid using stateful lambda expressions as parameters in stream operations. A stateful lambda expression is one whose result depends on any state that might change during the execution of a pipeline. Eg. sorted() and limit() are examples of of stateful.

limit() is a stateful operation since it has to keep the state of the items that are being picked up.

No Side Effect

Side effects can lead to incorrect results in parallel execution.

```
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));
```

The forEach lambda generates a side effect - it modifies *result*. **ArrayList** is a non-thread-safe data structure. We can replace **ArrayList** with **CopyOnWriteArrayList** instead.

Associative Accumulation Function

Stream three-argument reduce method

```
<U> U reduce(U identity, BiFunction<U,? super T,U> accumulator,
BinaryOperator<U> combiner)
```

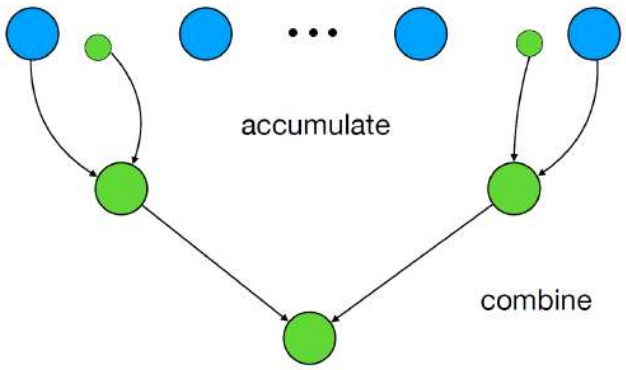
- `BinaryOperator<T> extends BiFunction<T, T, T>`

Rules to follow when parallelizing

- **combiner.apply(identity, i) must be equal to i**
  - reduce(**1**, (x,y) -> x+y, (x,y) -> x + y) //wrong!
  - combiner.apply(1, i) => i + 1 =/= i

- **combiner and accumulator must be associative**
  - Order of application must not matter
  - .reduce(1.0, **(x,y) -> x / y, (x,y) -> x / y**) // wrong! (1/2)/3 =/= 1(2/3)

- **combiner and accumulator must be compatible**
  - combiner.apply(U, accumulator.apply(identity, T)) must be equal to accumulator.apply(U, T)



- accumulator ((T,U) -> U) accumulates a stream element T with identity or outcome of another combine U
- combiner ((U,U) -> U) combines any identity U or outcome of another combine U

Lazy Evaluation

Eager Evaluation

```
int baz(int n, int x, int y) {
    return n %2 == 0 ? x : y;
}
baz(1, foo(), bar())
```

Both foo() and bar() have to be evaluated before passing the values to baz() even though baz() only needs to return one of the evaluations.

Lazy Evaluation

```
int baz(int n, Supplier<Integer> x, Supplier<Integer> y) {
    return n % 2 == 0 ? x.get() : y.get();
}
baz(1, () -> foo(), () -> bar())
```

Pass the functionalities of foo() and bar() to the baz() method and let it decide which of the two to invoke.

Caching

Supplier<T> to handle delayed data

Optional<T> to store the cache value

```
T get() {
    T v = this.cache.orElseGet(this.supplier);
    this.cache = Optional.<T>of(v); // mutating the property?
    return v;
}
```

Caching relies on the cache being mutable, but to the client, Lazy is still *observably immutable*.

Local Class and Variable Capture

Lambdas and anonymous classes declared inside a method are called local classes.

- Cannot mutate client-side states inside local classes

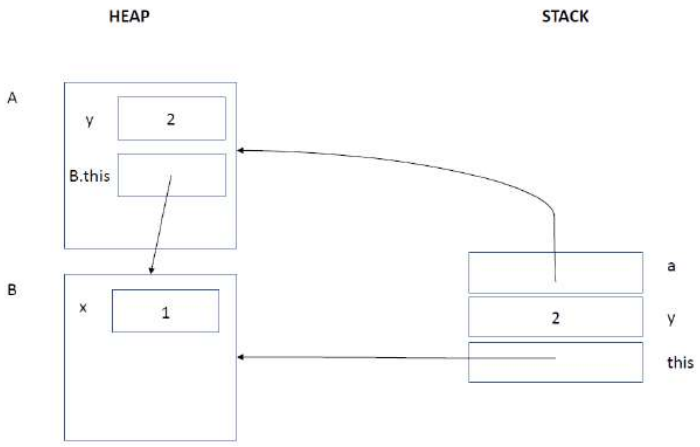
```
boolean isPrime(int n) {
    boolean prime = true;
    IntStream.range(2, n).forEach(x -> { if (n % x == 0) prime = false; });
    . . .    //local variable 'prime' is not effectively final
}
```

- Java only allows a local class to access variables that are explicitly declared final or effectively (or implicitly) final.
    - An implicitly final variable is one that does not change after initialization

Closure: Local class closes over it's enclosing method and class

- **Variable capture**: local class makes a copy of variables of the enclosing method and reference to the enclosing class

```
1  abstract class A {
2      abstract void g();
3  }
4
5  class B {
6      int x = 1;
7
8      void f() {
9          int y = 2;
10
11         A a = new A() {
12             void g() {
13                 x = y;
14             }
15         };
16
17         a.g();
18     }
19 }
```




```
B b = new B();
b.f();
```

```
class A {
  int x = 10;
  void foo(int y) {
    int z = 10;
    Supplier<Integer> sup = () -> x + y + z;
  }
}
```

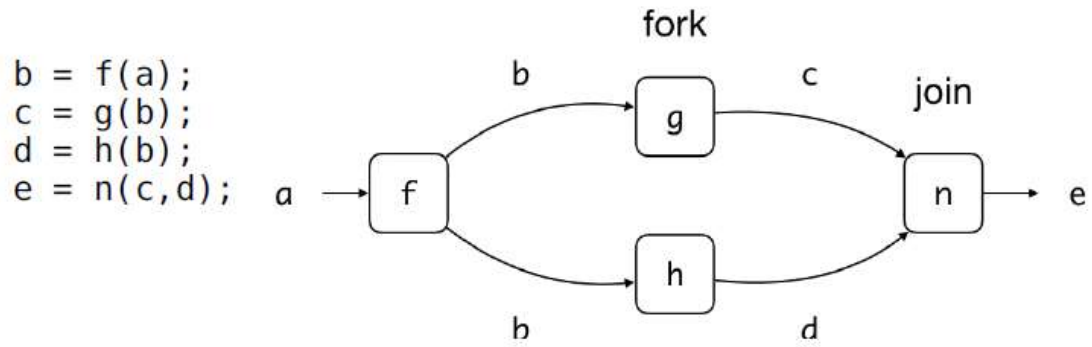
y, z, A.this is captured.

FP Contexts (CompletableFuture) / Asynchronous Programming

CompletableFuture

Context for asynchronous computation/promises

Fork-Join



- **Fork** task **g** to execute at the same time as **h,** then
- **Join** back task **g** later

CompletableFuture<T>

A key property of CompletableFuture is whether the value it promises is ready -- i.e., the tasks that it encapsulates has completed or not.

static methods **runAsync** and **supplyAsync** creates instances of **CompletableFuture** out of **Runnable** and **Supplier**.

- **The join() method is blocking and returns the result when execution completes**.
  - Returns Void for runnable tasks
- **Reminder: Always make sure that join() is called!**

thenApply is analogous to map

```
thenApply(Function<? super T, ? extends U> fn)

CompletableFuture.completedFuture(5)
    .thenApply(x -> x + 10).join();
   ⇨ 15
```

thenCompose is analogous to flatMap

```
thenCompose(Function<? super T, ? extends CompletableStage<U>> fn)

CompletableFuture.completedFuture(5)
    .thenCompose(x -> CompletableFuture.completedFuture(10)).join();
   ⇨ 10
```

Combining CompletableFutures

```
thenCombine(CompletionStage<? extends U> other,
    BiFunction<? super T, ? super U, ? extends V> fn)
```

```
jshell> CompletableFuture<Integer> cf2 = CompletableFuture.completedFuture(10)

jshell> CompletableFuture<String> cf1 = CompletableFuture.completedFuture("A")

jshell> cf1.thenCombine(cf2, (x,y) -> x + y).join()
==> "A10"
```

Async Variants of Callback Methods

- **thenRun** runs on the same thread
- **thenRunAsync** may run on a different thread

```
jshell> void foo() {
   ...>     CompletableFuture<Void> cf1 = CompletableFuture.runAsync(() -> {
   ...>         sleep(5);
   ...>         System.out.println("cf1: " + Thread.currentThread().getName()); });
   ...>
   ...>     CompletableFuture<Void> cf2 = cf1.thenRun(() -> {
   ...>         sleep(5);
   ...>         System.out.println("cf2: " + Thread.currentThread().getName()); });
   ...>
   ...>     CompletableFuture<Void> cf3 = cf1.thenRunAsync(() -> {
   ...>         sleep(5);
   ...>         System.out.println("cf3: " + Thread.currentThread().getName()); });
   ...>
   ...>     cf2.join();
   ...>     cf3.join();
   ...> }
|  created method foo()

jshell> foo()
cf1: ForkJoinPool.commonPool-worker-3
cf2: ForkJoinPool.commonPool-worker-3
cf3: ForkJoinPool.commonPool-worker-5
```

Pure Functions

A pure function is a function that

- takes in an argument and returns a deterministic value
- has no other *side effects*

Examples of side-effects

- Modifying external states
- Program input/output
- Throwing exceptions

The absence of side-effects is a necessary condition for referential transparency.

- Any expression can be replaced by its resulting value, without changing the property of the program.

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

int q(int x, int y) {   //Impure
    return x / y;       //Might throw Exception if divided by zero
}

void r(List<Integer> queue, int i) {      //Impure
    queue.add(i);                         //Modifies external state
}

int s(int i) {
    return this.x + i;
}
```