Unit 1: Program and Compiler

Compiler-time Errors. Errors detected from reading and analysing the source code without running it.

Runtime Errors. `

Unit 2: Variable and Type

Variable. An abstraction of data.

Type. An abstraction of functionality given to the data.

Dynamic Type. Same variable can hold values of different type.

Static Type. Variable can only hold values of the same type of the variable. Type of a variable cannot be changed. Type checking is done during compilation and runtime.

Compile-time Type. Type of the variable declared. It is the only type that the compiler is aware of.

Primitive Types. Numerical and boolean values.

Type	Size (bit)
byte	8
short	16
int	32
long	64
float	16
double	32

Subtypes. Let *S* and *T* be two types. *T* is a subtype of *S* if a piece of code written for variable of *S* can also safely be used on variables of type *T*.
T <: *S* denote *T* is subtype of *S* and *S* is a supertype of *T*.
Subtype relationship is transitive and reflexive.

Subtype Between Java Primitive Types.
byte <: short <: int <: long <: float <: double
char <: int

Unit 3: Functions

Functions as an Abstraction over Computation. Allow us to group a set of *instructions* and give it a name. Functions serve any purposes.
1. Allows programmers to compartmentalize computation and its effects.
2. Allows programmers to hide how a task is performed.
3. Reduce repetition in our code with code reuse.

Abstraction Barrier. The barrier between the code that calls the function and the code that defines the function body. Separates the role of the programmer into *implementer* and the *client*.

Unit 4: Encapsulation

Composite Data Types. Groups primitive data types together to form a complex data type. Allows the programmer to abstract away how a complex data type is represented. Representation and manipulation of the data type should fall on the same side of the abstraction barrier.

Class. Class is a data type with a group of functions associated with it. Functions are called methods and data are called fields. Maintains the abstraction barrier, only exposing the right method interfaces for others to use.

Encapsulation. The idea to keep all the data and the functions operating on the data within an abstraction barrier.

Objects. Instances of a class, containing the same methods and variables of the same type, but storing different values.

Object-Oriented Programming. Instantiates objects of different classes and orchestrates their interaction with each other by calling each other's methods. Typically, nouns are classes, properties of the nouns are fields and verbs are methods.

Reference Type. Everything that is not a primitive type. Variables do not store the data, only the reference to the data. Hence, two reference variables can share the same value.

Special Reference Value: *null*. Any reference variable that is not initialized will have the special reference value of *null*.

Unit 5: Information Hiding

Data Hiding. Using access modifiers, we can control what fields or methods can be accessed outside of the class. This functionality protects the abstraction barrier and is enforced by the *compiler* at compile time but are also enforced during runtime.

Constructor. Method that initializes an object.

```
class Circle {
  private double x;
  private double y;
  private double r;

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

The *this* Keyword. *this* is a reference variable that refers back to self. It can be used to distinguish between variables of the same name. In *this.x = x*, the *this.x* refers to the object field and the x refer to the parameter passed into the constructor.

Unit 6: Tell, Don't Ask

Accessors and Mutators. A class should also provide methods to retrieve and modify properties of the object. The client should tell the class what to do rather than performing the computation on behalf of the object. Try to minimise using accessor and modifier to private field.

Unit 7: Class Fields

Class Fields. Fields that do not belong to specific class and is universal (such as π).

The *static* Keyword. Associates the method or field with a class (rather than an object's instance fields).

The *final* Keyword. Indicate that the field will not change.

Accessing Class Fields. Accessed through the class name rather than an object. For instance, π is accessed with *java.lang.Math.PI* without instantiating the class.

Unit 8: Class Methods

Class Methods. Methods that do not belong to a specific class and is universal (such as *sqrt()*). Class methods cannot access any instance fields or call other instance methods. this has no meaning within a class method. Class methods are accessed through the class.

The main Method. Serves as the entry point to the program. Must be defined in the following way.

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

Unit 9: Composition

Composition. Used to represent a *HAS-A* relationship between two entities. For instance, a circle *has a* point.

Sharing References (aliasing). Two objects may share the same reference, causing unintended side effect. For instance, if two objects share the same reference, changing something in object1 may result in unintended changes to object2. Therefore, we should avoid sharing references as must as possible. Another solution is *immutability*.

Unit 10: Inheritance

Inheritance. Used to represent an *IS-A* relationship between two entities. For instance, a ColoredCircle is a Circle.

***extends* Keyword.** Used to give a class a subtype relationship.

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

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

Since ColoredCircle extends from Circle, ColoredCircle <: Circle.

Public fields of Circle and public method are accessible to ColoredCircle. However, anything private in the parent remains inaccessible to the child, maintaining the abstraction barrier.

***super* Keyword.** Calls the constructor of the superclass.

Inheritance Warnings. Be careful not to use inheritance when it is inappropriate as it may lead to unwanted behavior. Ensure inheritance preserves the meaning of subtyping.

Unit 11: Overriding

Object Class. Every class in Java inherits from the *Object* class implicitly. It has methods like *equals()* and *toString()*.

***toString* Method.** Inherited from the `Object` class and it is invoked implicitly by Java to convert a reference object to a String object.

Customizing *toString* for Circle. We can define our own *toString()* method by writing a *toString()* method in the class.

```
class Circle{
  ⋮
  @Override
  public String toString(){
    return "I am a Circle";
  }
}
```

Method Overriding. We can alter the behavior of an existing class. This occurs when a subclass defines an instance method with the same *method signature* as an instance method in the parent class. In addition, the return type of the overriding method needs to be a subtype of the overridden method.

Method Signature. Defined by the method name and the number, type, and order of its parameters.

Method Descriptor. Method signature and the return type.

***@Override* Notation.** A hint to the compiler that the method below intends to override the method in the parent class.

Unit 12: Polymorphism

Taking on Many Forms. Method override enables polymorphism, allowing us to change how existing code behaves without changing a single line of code.

Dynamic Binding. The method that is invoked is decided during run-time and is depended on the run-time type. Hence, the same method invocation can cause two different methods to be called. This process happens *only* during run-time.

Typecasting. Done during a narrowing type conversion, where we convert *T* to *S* even though *S* <: *T*. Typecasting will lead to a run-time error if not used appropriately. Type casting is done during compile time but is checked during run-time.

Unit 13: Liskov Substitution Principle

Liskov Substitution Principle. "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."

In other words, if we substitute a superclass object reference with an object of any of its subclasses, then program should not break.

LSP cannot be enforced by the compiler.

***final* Keyword for Classes and Methods.** The *final* keyword prevents classes from being *inherited* from and prevents methods from being *overridden*.

Unit 14: Abstract Classes

Abstract Classes. A class that has been made into something so general that it cannot and should not be instantiated (such as Shape). Usually means that one of its instance methods cannot be implemented without further details.

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

Not all methods have to be abstract. A class that is not an abstract class is a concrete class.

Unit 15: Interface

Interface. An interface is also a type and is declared with the keyword *interface*, with a name that usually ends with the -able suffix. Acts as a blueprint for its subclasses.

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

Both abstract classes and concrete classes can implement an interface. For a class to implement an interface and be concrete, it must override all abstract methods from the interface and provide an implementation to each. Otherwise, the class becomes abstract.

The *implements* Keyword. Used to show that subclass is implementing an interface. A class can only extend from one superclass but can implement multiple interfaces. Interfaces can extend from other interfaces but cannot extend from another class.

Interface as Supertype. A type can have many supertypes since a class can implement many interfaces.

Impure Interfaces. Interfaces that provide a default implementation of methods that all implementation subclasses will inherit (unless they override). Tagged using the *default* keyword.

Unit 16: Wrapper Class

Wrapper Classes. A wrapper class is a class than encapsulates a type rather than fields and methods. They are created with "*new*" and instances are stored on the heap. All primitive wrapper class objects are immutable and cannot be changed.

Auto-boxing and Unboxing. Performs type conversion between primitive type and its wrapper class.

Performance of Wrapper Class. A wrapper class comes with a memory overhead and is less memory efficient that primitive types.

Unit 17: Run-Time Class Mismatch

Cast Carefully. There may be a need for narrowing type conversion when writing code that depends on higher-level abstraction. However, typecasting must be done carefully since they cause run-time errors (bad) if not done properly.

Unit 18: Variance

Covariant Java Arrays. Since Java arrays are covariant, it is possible to assign an instance with a run-time type `Integer[]` to a variable with a compile-time type `Object[]`.

Variance of Types. Subtype relationship between *complex* types such as arrays are non-trivial. The variance of types refer to how the subtype relationship between complex types relates to the subtype relationship between components.

Let *C(S)* be some complex type based on type *S*.
We say a complex type is
(a) *covariant* if *S* <: *T* implies *C(S)* <: *C(T)*
(b) *contravariant* if *S* <: *T* implies *C(T)* <: *C(S)*
(c) *invariant* if it is neither covariant nor contravariant

Java Array is Covariant. This means that if *S*<:*T*, then *S*[]<:*T*[]. Consequently, it is possible for run-time error to occur even without typecasting (!!).

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

The compiler does not realize that the runtime type of *objArray* is actually *Integer[]*. However, this code compiles perfectly since *Integer* <: *Object* and *String* <: *Object.*

Unit 19: Exceptions

Try-Catch-Finally Syntax. The general syntax of a try-catch-finally are as follows.

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

You can have multiple catch statements, each catching a different type of exception. Information about an exception is encapsulated in an exception instance and passed into the catch block. In the example above, `e` is the variable containing an exception instance. You can combine multiple exceptions into one catch block with the " | " operator
i.e. *catch (Exception1 | Exception2) {…}*

Throwing Exceptions. To throw exceptions, we need to do two things.
1. We need to declare that the construct is throwing an exception with the *throws* keyword.
2. we need to create a new Exception object and throw it with the *throw* keyword.

```
public Circle(Point c, double r) throws SomeException {
  if (r < 0) {
    throw new SomeException("Negative Radius")
  }
  ⋮
}
```

Checked vs Unchecked Exceptions. *Unchecked exceptions* are caused by programmer error and should not happen if perfect code was written. Unchecked exceptions are usually not caught or thrown. *Checked exceptions* are exceptions that the programmer has no control over. The programmer needs to actively anticipate the exception and handle them when they occur. Unchecked exceptions are subclasses of `RuntimeException`.

Creating Our Own Exceptions. You can create your own exceptions by inheriting from one of the existing exceptions. You must throw the same or more specific exception. (following LSP)

Do Not Catch-Them-All. Do not catch all exceptions and do nothing. In doing so, all exceptions will be silently ignored.

Do Not Exit the Program Because of an Exception. That would prevent the function from cleaning up their resources.

Do Not Break Abstraction Barrier. Try to handle implementation-specific exceptions within the abstraction barrier.

Do Not Use Exceptions as Control Flow. Do not use exception to handle the logic of the program.

Unit 20: Generics

Generics. Java allows us to define a generic type that takes in other types as type parameters.

Creating Generics. We can create generics by specifying the type parameters between the `<` and `>`. By convention, we use a single capital letter to name each type parameters.

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

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

  S getFirst(){
    return this.first;
  } ⋮
}
```

Instantiating a Generic Type. In order to use a generic type, you have to pass in type parameters. Once a generic type is declared, it is called a *parameterized type*. Only reference types can be used as type arguments.

Generic Methods. Methods can also be parameterized with a type parameter.

```
public <T> boolean contains(T[] arr, T obj){⋯}
```

Bounded Type Parameters. We can put constraints on the type parameters, by using the `extends` keyword to ensure that the type parameter fulfills some condition.

```
public <T extends GetAreable> T contains(⋯){⋯}
```

We can use bounded type parameters for declaring generic classes as well (not only generic methods).

```
class Pair<S extends Comparable<S>, T> implements Comparable<Pair<S, T>>
{⋯}
```

This line shows that S must be a subtype of Comparable<S> and that two pair instances are comparable to each other. Note that if `T extends S`, `T` can be ` or any subtype of `S`.

Unit 21: Type Erasure

Implementing Generics. Java erases the type parameters and type arguments after type checking during compilation. Hence, there is only one representation of the generic type in the generated code, representing all the instantiated generic types regardless of type arguments.

Type Erasure. Type erasure removes wildcards and bounded type parameters during compilation after compile-time verification.

T extends Integer	→	Integer
? extends Integer	→	Integer
? super Integer	→	Object

Generics and Arrays Can't Mix. Due to type erasure and the covariance of Java arrays, it is possible to put generics with different type parameters into the same array (since the type parameters are erased) possibly causing `ClassCastExceptions`.

Unit 22: Unchecked Warnings

Unchecked Warnings. A warning from the compiler that due to type erasure, it is possible that a run-time error could occur.

`@SuppressWarnings` Annotation. It suppresses warning message from the compiler. *`@SuppressWarnings`* can apply to declaration of different scope, hence we should always use it at its most limited scope. We only use it in situations where we are sure that it will not cause a type error. A comment should be added to justify its use. Lastly, it cannot be used on assignment, only declaration.

```
@SuppressWarnings("unchecked")
```

Raw Types. A generic type being used without type arguments. The compiler is unable to do any type-checking. Mixing raw types and parameterized types may also cause errors. The only time when raw type can be used is as an operand of the *`instanceof`* operator.

Unit 23: Wildcards

Upper-Bounded Wildcards. Denoted by *`Array<? extends SomeType>`*. The `?` can be substituted in with either *`SomeType`* or any subtype of *`SomeType`*. The upper-bounded wildcard has the following properties.

1) If `S` <: `T`, then `A<? extends S>` <: `A<? extends T>`.
2) For any type `S`, `A<S>` <: `A<? extends S>`

Lower-Bounded Wildcards. Denoted by `Array<? super SomeType>`. The `?` can be substituted in with either `SomeType` or any supertype of `SomeType`. The lower-bounded wildcard has the following properties.

3) If `S` <: `T`, then `A<? super T>` <: `A<? super S>`.
4) For any type `S`, `A<S>` <: `A<? super S>`

PECS. *Producer extends, consumer super*. If the variable is a producer that returns a variable of type T, then should be declared with the wildcard "*`? extends T`*". In contrast, if a variable is a consumer that accepts a variable of type T, it should be declared with the wildcard "*`? super T`*"

Unbounded Wildcards. Denoted by `<?>`. Used in mainly in two scenarios. Can result in very restrictive methods.

1. If you are writing a method that can be implemented using functionality provided in the `object` class
2. When the code is using methods in the generic class that do not depend on the type parameter. E.g., Array size.

Unit 24: Type Inference

Type Inference. Java will look among the matching type that would lead to successful type checks and pick the *most specific* ones. Done during compilation.

Diamond Operator. Indicates to the compiler that type inference should be used. This can only be used to instantiate a type and never as a type.

Target Typing. Type inference that involves the type of the expression.

Unit 25: Immutability

Immutability. An instance of an immutable class cannot have any visible changes *outside of its abstraction barrier*. Hence, every call of the instance's method must behave the same way throughout the life of the instance.

***`final`* Keyword on Variables.** Use the final keyword on field to signal our intention that we do not intend to change them.

```
final private double x;
```

To prevent subclasses from overriding the methods, it is good practice to also make immutable classes final to disallow inheritance.

```
final class Circle {⋯}
```

Ease of Understanding. Immutable objects are easier to reason with and easier to understand. Unless explicitly reassigned, variable can be guaranteed to remain the same.

Enabling Safe Sharing of Objects. Since immutable objects cannot change, we can safely share instances of the class, reducing the need to create multiple copies of the same object.

Enabling Safe Sharing of Internals. Since immutable objects cannot change, internal information can also be shared to other classes (or subclasses) safely with the guarantee that the information referenced in the parent instance will never change. (e.g., subarray method in ImmutableArray)

Enabling Safe Concurrent Execution. Allows for us to reduce bugs in concurrent code since we can guarantee the correctness of our objects no matter the code order.

`Varargs` Syntax. Java syntax for a variable number of arguments of the same type. Syntactic sugar for passing in an array of items to a method.

```
@SafeVarargs
public static <T> ImmutableArray<T> of(T... items) {⋯}
```

`@SafeVarargs` notation. Since varargs is just an array, and arrays and generics do not mix well in java, the compiler would throw us an unchecked warning. If we are sure only one type can be placed in the array, we can use the `@SafeVarargs` to tell the compiler that this varargs is safe.

Unit 26: Nested Class

Nested Class. A nested class is a class defined within another containing class. It is used to group logically relevant classes together. Typically, a nested class would have no use outside of the container class, acting as a "helper" class that serves a specific purpose.

Access of a Nested Class. A nested class can access the fields and the methods of the container class, *including those declared as `private`*.

Keeping a Nested Class Within the Abstraction Barrier. We can keep the nested class within the abstraction barrier by declaring the nested class as *`private`* if there is no need for it to be exposed to the client outside the barrier.

The Same Encapsulation as Container Class. Since a nested class can access the private fields of the container class, we should only introduce a nested class if it belongs in the same encapsulation. Otherwise, the container class would leak its implementation details to the nested class.

Nested Static Classes. A static nested class is associated with the containing class, *not an instance*. It can only access static fields and static methods of the containing class.

Qualified *`this`*. Used from within the nested class to refer to the instance of the containing class. A regular *`this`* would refer to the instance of the nested class.

```
ContainingClass.this.field
```

Local Class. A class that is declared within a function, like a local variable. Local classes are scoped within the method. Just like a nested class, a local class has access to the variables of the enclosing class through the qualified *`this`* reference. It also has access to local variables of the enclosing method.

Variable Capture. When a method returns, all local variables of the method are removed from the *stack*. But an instance of that local class might still exist. The local class might reference local variable on the stack that may or may not been removed. To prevent such errors, the local class makes a copy of local variables inside itself.

Variable Captures are Effectively Final. To prevent bugs, Java only allows a local class to access variables that are explicitly declare *`final`* or implicitly *`final`*.

Anonymous Class. An anonymous class is one where you declare and instantiate the class in a single statement. It is anonymous since we do not even have to give the class a name. It cannot have a constructor and is just like a local class, capturing the variables of the enclosing scope as well as having the same rules regarding variable access. An anonymous class has the following format.

```
new X (arguments) { body }
```

`X` is a class that the anonymous class extends or an interface that the anonymous class implements. `X` cannot be empty. `X` cannot be more than one interface or class. For example,

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

Unit 27: Side Effect-Free Programming

Function as First Class Citizens. Functions can be assigned to a variable, passed as parameters, and returned from another function, just like an integer.

Pure Functions. A pure function is a function that does not cause any side effects. It cannot print to screen, write to files, throw exceptions, modify other variables, or modify the values of the arguments. A pure function must also be deterministic. Given the same input, the function must produce the same output *every single time*, ensuring referential transparency.

Functional Programming. The style where we build programs from pure functions.

Functional Interface. A functional interface is an interface with only one abstract method. It is recommended to annotate a functional interface with *`@FunctionalInterface`* notation.

Lambda Expressions. Lambda expressions allow us to write classes that implements functional interfaces in a concise manner.

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

Method Reference. A double-colon notation is used to specify a method reference.

```
1.  Box::of   // of method of the class box
2.  Box::new  // x -> new Box(x)
```

Curried Functions. A method of transforming a general *n-ary* function to a *sequence of n unary functions*.

```
x -> y -> (x + y);
```

Lambda as Closure. Lambda expressions do not simply store the function to invoke. They also store the data from the environment where it is defined. This construct that stores a function and the enclosing environment is called a closure. A lambda expression is hence able to save the current execution environment and continue to compute it later.

Unit 28: Box and Maybe

Lambda as a Cross-Barrier State Manipulator. Instance methods which accept functions as parameters allows the client to manipulate the data behind the abstraction barrier without knowing the internals of the object. We can treat lambda expressions as "manipulators" that we pass behind the abstraction barrier and modify the internal arbitrarily for us.

Unit 29: Lazy Evaluation

Lambda as Delayed Data. Lambda expression allows us to delay the execution of code, since the body of the function is not executed until we invoke the function.

Lazy Evaluation. We can build up a sequence of complex computations without executing them. These computations are evaluated on demand when needed.

Memoization. Speeds up programs by storing the result of expensive function calls and returning the cached result when the same inputs occur again.

Unit 31: Stream

Building a Stream.

1. Using the static factory method `of`
2. Using the *`generate`* and *`iterate`* methods
3. Convert from an array to a *`Stream`* using *`Arrays::stream`*
4. Convert from a *`List`* instance into a `Stream` using a *`List::stream`*

Terminal Operations. An operation on the stream that triggers the evaluation of the stream. E.g., *`forEach`*.

Intermediate Stream Operations. Operations that return another `Stream`. These include *`map`*, *`filter`* and *`flatMap`*. Most intermediate operations are lazy and do not cause the *`Stream`* to be evaluated.

Stateful and Bounded Operations. Stateful operations are intermediate operations that are stateful and need to keep track of some states to operate. (Such as *`sorted`* and *`distinct`*) Bounded operations are operations that should only be called on a finite stream.

Truncating an Infinite Stream. Operations that convert an infinite stream to a finite stream.

1. *`limit`* takes in an *`int n`* and returns a stream containing the first `n` elements of the stream.
2. *`takeWhile`* takes in a predicate and returns a stream containing the elements of the stream, until the predicate becomes false. It may return an infinite stream if the predicate never becomes false.

Peeking with a Consumer. *`peek`* is a useful intermediate operation of *`Stream`* which takes in a *`Consumer`* and allow us to apply a lambda on a "fork" of the stream.

Reducing a Stream. Terminal operation that applies a lambda repeated on the elements of a stream to reduce it into a single value.

Element Matching. Terminal operations for testing if the elements pass a given predicate.

1. *`noneMatch`* returns true if none of the elements pass the given predicate.
2. *`allMatch`* returns true if every element passes the given predicate.
3. `anyMatch` returns true if at least one element passes the given predicate.

Consumed Once. A stream can only be operated on once. We will have to recreate the stream if we want to operate on the stream more than once.

Unit 32: Loggable

The Importance and Uses of *`flatMap`*. *`flatMap`* allows us to manipulate the contents as well as the side information stored.

Unit 33: Monad

Identity Laws. The `of` method of a monad should behave like an identity and should not do anything extra to the value and the side information. It should simply wrap the value into the monad. Similarly, the `flatMap` method should not do anything extra to the value and the side information and should simply apply the given lambda expression to the value.

Left Identity Law:
`Monad.of(x).flatMap(x -> f(x)) ≡ f(x)`

Right Identity Law:
`someMonad.flatMap(x -> Monad.of(x)) ≡ someMonad`

Associative Law. It should not matter if we group functions together into another function before applying it to a value or if we compose the functions by chaining the `flatMap`. Formally,
`someMonad.flatMap(x -> f(x)).flatMap(x -> g(x)) ≡ someMonad.flatMap(x -> f(x).flatMap(y -> g(y)))`

Functors. A simpler construction than a monad in that it only ensures lambdas can be applied sequentially to the value, without worrying about side information. We can think of a functor as an abstraction that supports map.

Identity:
`someFunctor.map(x -> x) ≡ someFunctor`

Composition:
`someFunctor.map(x -> f(x)).map(x -> g(x)) ≡`
`someFunctor.map(x -> g(f(x)))`

Unit 34: Parallel Streams

Parallelism. Parallel computing refers to the scenario where multiple subtasks are truly running at the same time.

Concurrency. A program runs concurrently by dividing the computation into subtasks called threads. It has 2 main advantages.
1. It allows programmers to separate unrelated tasks into threads and write each thread separately.
2. It improves the utilization of the processor. (non-blocking)

Parallel and Concurrent Programs. All parallel programs are concurrent, but all concurrent programs are parallel.

Parallel Streams. The Java Stream allows for parallel operations on the elements of the stream in one single line of code.

```
someStream.filter(···).parallel().count()
```

***`.parallel`* Method.** This method enables parallel processing of the stream. *`parallel()`* is a lazy operation and merely marks the stream to be processed in parallel. As such, you can insert *`.parallel()`* anywhere in the chain.

What Can Be Parallelized? Stream operations must not interfere with stream data. Most of the time they are stateless. Side-effects should be kept to a minimum.

Interference. Stream operations that modify the source of the stream during the execution of the terminal operation would cause a *`ConcurrentModifiedException`*.

Stateful vs. Stateless. A stateful lambda is one where the result depends on any state that might change during the execution of the stream.

Side Effects. Side effects are modifications outside the stream which may lead to incorrect behavior. For instance, adding to a non-thread-safe data structure.

Reduce Method. The reduce method has the following method signature.

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

`identity`	Both an initial seed value for the reduction and a default result if there are no input elements..
`accumulator`	Takes a partial result and the next element, and produces a new partial result.
`combiner`	Combines two partial results to produce a new partial result.

Conditions to Parallelize Reduce Method.
1. `combiner.apply(identity, i) ≡ i`
2. `combiner` and `accumulator` must be associative.
3. `combiner` and `accumulator` must be compatible.
   `combiner.apply(u, accumulator.apply(identity, t)) ≡ accumulator.apply(u, t)`

Overloaded Reduce Method. There is an overloaded reduce method with the method signature of

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

In this method, the combiner is also the accumulator, and the function can only return the same type as the stream.

Performance of Parallel Streams. Parallelizing a stream does not always improve the performance. In fact, creating a thread to run a task incurs some overhead which maybe sometimes outweigh the benefits of parallelization.

Ordered vs. Unordered Streams. Whether or not the stream elements are ordered or unordered also affects the performance of parallel stream operations. Some stream operations respect the encounter order. Some operation such as *`findFirst`*, *`limit`* and *`skip`* can be expensive to parallelize on an ordered stream since it needs to coordinate between the streams to maintain order.

`.unordered()` Method. If we have an ordered stream but respecting the original order is not important, we can call *`unordered()`* as part of the chain command to make parallel operations much more efficient.

Unit 35: Threads

Synchronous Programming. A programming model where tasks are performed one at a time and execution of a task cannot begin until the previous task has been completed.

Thread. A thread is a single flow of execution in a program.

Thread Constructor. The *`new Thread(..)`* is the usual constructor used to create a *`Thread`* instance. The constructor that's in a *`Runnable`* instance as an argument, which is a function interface with a *`run()`* method that take in no parameter and returns *`void`*.

Starting a Thread. With each *`Thread`* instance, we run *`start()`*, which causes the given *`Runnable`* to run. *`start()`* ***returns immediately*** and does not return only after the given lambda expression completes its execution.

Threads Example. The two threads below run in two separate sequence of execution. The operating system decides which thread to run when, and on which core. As a result, you may see different interleaving of executions every time you run the same program.

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

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

Thread Names. We can use the class(static) method *`Thread.currentThread()`* to get the reference to the current running thread and the instance method *`getName()`* to find out the name of the thread.

Sleep. The class method *`.sleep()`* causes the current execution thread to pause execution immediately for a given period (in milliseconds). After the sleep timer is over, the thread is ready to be chosen by the scheduler to run again.

```
Thread.sleep(1000) // sleep for 1 second
```

`isAlive()` Method. The instance method *`isAlive()`* checks if another thread is still running.

```
while (otherThread.isAlive()){···}
```

Unit 36: Asynchronous Programming

Limitations of *`Thread`*. *`Thread`* is a relatively low-level abstraction which requires a fair amount of effort to write complex multi-threaded programs. It has the following drawbacks.

1. No method in *`Thread`* returns a value, hence we need the threads to communicate through shared variables.
2. There is no mechanism to specify the execution order and the dependency among threads.
3. We must consider the possibility of exceptions in each of our tasks.
4. A creation of *`Thread`* instances takes up some resources in Java. Hence, we should reuse our *`Thread`* instances to run multiple tasks.

CompletableFuture. Completable future is a *monad* found in *`java.util.concurrent.CompletableFuture`*. *`CompletableFuture`* encapsulates the promise to return a value. A key property of *`CompletableFuture`* is whether the value it promises is ready.

Creating a `CompletableFuture`. There are several ways we can create a CompletableFuture<T> instance:

1. Use the `completedFuture` method that takes in a value of type T. This method is equivalent to creating a task that is already completed and return us a value.
2. Use the `runAsync` method that takes in a `Runnable` lambda expression. `runAsync` has the return type of `CompletableFuture<Void>`. The returned `CompletableFuture` instance completes when the *given lambda* expression finishes.
3. Use `supplyAsync` method that takes in a `Supplier<T>` lambda expression. `supplyAsync` has the return type of `CompletableFuture<T>`. The returned `CompletableFuture` instance completes when the given lambda expression finishes.

We can also create a `CompletableFuture` that relies on other `CompletableFuture` instances.

1. Use the `allOf` method that takes in a variable number of other `CompletableFuture` instances. The returned `CompletableFuture<Void>` is completed when all the given `CompletableFutures` complete.
2. Use the `anyOf` method that takes in a variable number of other `CompletableFuture` instances. The returned `CompletableFuture<Object>` is completed when any of the given `CompletableFutures` completes, with the same result.

Chaining `CompletableFuture`. The usefulness of `CompletableFuture` comes from the ability to chain them up and specify a sequence of computations to be run.

1. Use the `thenApply` method that takes in a `Function` lambda expression. It returns a new `CompletableFuture` that, when the current instance completes, is executed with the result of the current instance as the argument to the supplied function. *Analogous to `map`.*
2. Use the `thenCompose` method that takes in a `Function` lambda expression. *Analogous to `flatMap`.*
3. Use the `thenCombine` method that takes in another `CompletableFuture` and a `BiFunction`. *Analogous to `Combine`.*

There are also asynchronous versions of each of the methods above which may cause the given lambda expression to run in a different thread, resulting in more concurrency. (`thenApplyAsync, thenComposeAsync, thenCombineAsync`)

Getting the Result. After setting up our task to run asynchronously, we must wait for them to complete. We should only get the result as the final step in our code as getting the result is a *synchronous* call. The `get()` method throws two checked exceptions:

- `InterruptedException` when the thread is interrupted.
- `ExecutionException` when there are errors/exceptions during execution.

An alternative to `get()` is `join()` which behaves just like `get()` except that no checked exception is thrown.

Handling Exceptions. CompletableFuture has built-in functionality to handle exceptions. The handle method takes in a BiFunction which takes in the result and an exception as arguments. The BiFunction returns a value of type U and the handle method returns a CompletableFuture encapsulating that value. Only one of the first two parameters is not null. If value is null, that means that an exception has been thrown. Otherwise, the exception is null.

```
cf.thenApply(x -> x + 1)
  .handle((t, e) -> (e == null) ? t : 0)
  .join();
```

Unit 37: Fork and Join

Thread Pool. Recall that creating and destroying threads is not cheap, and we should reuse existing threads to perform different tasks. A thread pool consists of (i) a collection of threads, each waiting for a task to execute, (ii) a collection of tasks to be executed. Typically, the tasks are put in a queue, and an idle thread picks up a task from the queue to execute.

Fork-Join Model. The general idea for the fork-join model is to solve a problem by breaking up the problem into identical problems but with smaller size, then solve the smaller version o the problem recursively and combine the results.

`RecursiveTask<T>`. In Java, we can create a task that we can fork and join as an instance of the abstract class `RecursiveTask<T>`.

`RecursiveTask<T>` Methods. `RecursiveTask` supports the following methods.

- The `fork()` method submits a smaller version of the task for execution
- The `join()` method waits for the smaller tasks to complete and then it returns.
- The `compute()` method which we have to define to specify what computation we want to compute.

ForkJoinPool. The ForkJoinPool is how Java manages the thread pool with fork-join tasks. The few key points are as follows.

- Each thread has a queue of tasks.
- When a thread is idle, it checks its queue of tasks. If queue is not empty, it picks up a task at the *head* of the queue to execute (invoking its `compute` method). Otherwise, it picks up a task from the *tail* of the queue of another thread to run. The latter is a mechanism called work stealing.
- When `fork()` is called, the caller adds itself to the head of the queue of the executing thread. This is done so the most recently forked task get executed next, similar to normal recursive calls. It arranges to asynchronously execute this task in the *pool* the current task is running in
- When `join()` is called, several cases might happen.
  1. If the subtask to be joined has not been executed, `compute()` method is called and the subtask is executed.
  2. If the subtask to be joined has been completed (some other thread has stolen this and completed it), then the result is read, and `join()` returns.
  3. If the subtask to be joined has been stolen and is being executed by another thread, then the current thread finds some other task to work on either in its local queue or steal another task from another queue.

Threads always looks for something to do and they cooperate to get as much work done as possible.

Order of `fork()` and `join()`. The order in which we call `fork()` and `join()` would affect the efficiency of our code. Since the most recently forked task is likely to be executed next, we should `join()` the most recently forked task first. In other works, the order of forking should be the reverse of the order of joining.

`RecursiveTask<T>` Example.

```
import java.util.concurrent.RecursiveTask;
class Fib extends RecursiveTask<Integer> {
  final int n;

  Fib(int n) {
    this.n = n
  }

  @Override
  protected Integer compute() {
    if (n <= 1) {
      return n;
    }

    Fib f1 = new Fib(n-1);
    Fib f2 = new Fib(n-2);

    // try different variant here.
  }
}
```

Variant	Speed	Explanation
`f1.fork();` `return f2.compute() + f1.join();`	Fastest	`f2.compute()` and `f1.fork()` run concurrently on main and worker threads
`f1.fork();` `return f1.join() + f2.compute();`	Slow	`f2.compute()` has to wait for `f1.fork()` to complete before proceeding
`return f1.compute() + f2.compute();`	Medium	Everything is done in the main thread. Faster than the previous one because there is no overhead from forking and joining

`f1.fork();`
`f2.fork();`
`return f2.join() + f1.join()`
	Fast	Recommended way to write recursive tasks. Apart from the first recursion, main thread delegates all other work to worker threads in the common pool hence slower than first variant

`f1.fork();`
`f2.fork();`
`return f1.join() + f2.join()`
	Medium	Wrong order of fork and join

Notes:

1. Check carefully whether a method is being overridden.
2. You cannot instantiate a 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.