CS2030s Notes

Unit 0: Overview

Functions

1. Compartmentalize computations
2. Hide implementation details
3. Code reuse

The Abstraction Principle

> *"Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts."*

In the case of *functions*, the "*varying parts*" refers to the values on which we wish to perform the computation on.

In the case of *types*, abstracting them out as parameterized types or subtypes.

In the case of *sub-computation*, abstracting them out as first-class functions.

Erecting an Abstraction barrier

Separate the role of a programmer into *two*, the *implementer* and the *client*.

The *implementer* should:

- Compartmentalize the internal variables and the implementation of the function, hiding them behind the abstraction barrier
- Expose parameters and returns values as the only communication gateways across the barrier.

Maintaining this abstraction barrier by encapsulating functions, variables and computations as objects, and hiding details from the client through *access modifiers*.

Encapsulation and *abstraction* forms two of the core principles of OOP.

Code for Change

Two ways to modify the computation behind the abstraction barrier, without changing the code behind the barrier.

1. Inheritance and Polymorphism
2. Closure (aka lambda functions)
   > 1. This idea if carried to the extreme in terms of flexibility leads to the concept of monad.
   > 2. A monad is a computational structure that allows objects to be composed and manipulated in a succinct and powerful way.

Types

Changing the behavior of existing code without changing the code could lead to more bugs if not managed properly. Important to adhere to certain rules when extending or modifying the behavior of existing code.

Type systems is a set of rules that governs how variables, expressions, and functions interact with each other.

Two notions that are important to constrain how inheritance and polymorphism should be used to avoid bugs. It reduces possible interdependence between programming constructs.

1. Subtyping
2. Liskov Substitution Principle

Every attempt to break the constraint can be caught automatically. We can detect potential bugs before it manifests itself.

The concept of types, subtypes, compile-time vs runtime types, variants of types, parameterized types and type inferences;

Side effects

Functions should not have any side effects.

With *immutability* and pure functions, we can guarantee that the same function invoked on the same objects will always return the same value. This certainty can help in understanding and reasoning about the code behavior.

Note

Immutability: Once we create an object, the object cannot be changed. To update it, we simply create a new object and reassigns it.

Unit 1: Compiler

- recap some fundamental programming concepts, including the concept of a program, a programming language, a compiler, an interpreter
- be aware of two modes of running a Java program (compiled vs. interpreted)
- be aware that compile-time errors are better than run-time errors, but the compiler cannot always detect errors during compile time

Compiler compiles the code into bytecode before executing it at runtime. *Interpreter* interprets each line of code and execute it directly.

Better for errors to be caught during compile time

- it's still under the control of the programmer.
- it's still under development.

A compiler can only read and analyze the code without actually running it. Without running the program, the compiler cannot always tell if a particular statement in the source code will ever be executed. It cannot always tell what values a variable will take.

To deal with this, the compiler can either be *conservative* or, *permissive*. If there is a possibility that a particular statement is correct, it does not throw an error, but rely on the programmer to do the right thing.

Conservative	**Permissive**
report an error as long as *there is a possibility* that a particular statement is incorrect	reporting an error only if *there is no possibility* that a particular statement is correct

Unit 2: Variable and Types

- appreciate the concept of variables as an abstraction
- understand the concept of types and subtypes
- contrast between statically typed language vs. dynamically typed language
- contrast between strongly typed language vs. weakly typed language
- be familiar with Java variables and primitive types
- understand widening type conversion in the context of variable assignments and how subtyping dictates whether the type conversion is allowed.

The type communicates to the readers what data type the variable is an abstraction over, and to the compiler/interpreter what operations are valid on this variable and how the operation behaves.

Dynamic Type The same variable can hold values of different type, and checking if the right type is used is done during the execution of the program.

Static Type We need to declare every variable we use in the program and specify its type.

The compiler will check if the compile-time type matches when it parses the variables, expressions, values, and function calls, and throw an error if there is a type mismatch.

Strongly Typed vs Weakly Typed

A *strongly* typed language enforces strict rules in its type system to ensure type safety.

A *weakly* typed language is more permissive in terms of type checking. You can cast a String to an Integer if you wish to. The control is yours.

**Primitive Types in Java

Primitives are types that hold numeric values as well as Boolean values. These include `byte`, `short`, `int` and `long` for storing 8, 16, 32, 64 bits respectively.

The type `char` stores 16-bit unsigned integers representing UTF-16 Unicode characters. For storing floating-point values, Java provides two types, `float` and `double`, for 16-bit and 32-bit floating-point numbers.

Subtyping relationship between primitive types

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

Unit 3: Functions

- understand the importance of function as a programming constructor and how it helps to reduce complexity and mitigate bugs.
- be aware of two different roles a programmer can play: the implementer and the client
- understand the concept of abstraction barrier as a wall between the client and the implementer, including in the context of a function.

Function as an Abstraction over Computation

- Functions allow programmers to **compartmentalize** computation and its effects.
  - To isolate the complexity to within its body.
  - Interacts only with its parameters and return value, reduces the dependencies and complexity of code.
- Functions allow programmers to **hide** *how* a task is performed.
  - Reduce the amount of information that we need to communicate among programmers.
  - The implementation of a function may change. But, as long as the parameters and the return value of a function remains the same, the caller of the function does not have to update the code.
- Functions allows us to **reduce** repetition in our code through *code reuse*.
  - Reduces the amount of boiler-plate code, code complexity and bugs.
  - Makes code more succinct and reduces the number of places we need to modify should the software evolves (decreasing the chance of introducing new bugs).

Abstraction Barrier

The abstraction barrier separates the role of the programmer into two: (i) an *implementer*, who provides the implementation of the function, and (ii) a *client*, which uses the function to perform the task.

Unit 4: Encapsulation

- understand composite data type as a even-higher level abstraction over variables
- understand encapsulation as an object-oriented (OO) principle
- understand the meaning of class, object, fields, methods, in the context of OO programming
- be able to define a class and instantiate one as an object in Java
- appreciate OO as a natural way to model the real world in programs
- understand reference types in Java and its difference from the primitive types

Composite Data Types

A *composite data type* allows programmers to group *primitive types* together, give it a name (a new type), and refer to it later.

Defining composite data type allows programmers to abstract away (and be separated from the concern of) how a complex data type is represented.

Abstraction: Class and Object (or, Encapsulation)

We can bundle the composite data type *and its associated functions* on the same side of the abstraction barrier together, into another abstraction called a *class*.

A class is a data type with a group of functions associated with it. A well-designed class maintains the abstraction barrier, properly wraps the barrier around the internal representation and implementation, and exposes just the right *method interface* for others to use.

Important

The concept of keeping all the data and functions operating on the data related to a composite data type together within an abstraction barrier is called *encapsulation*.

Object-Oriented Programming

A program written in an *object-oriented language* consists of classes, with one main class as the entry point. One can view a running object-oriented (or OO) program as something that instantiates objects of different classes and orchestrates their interactions with each other by calling each others' methods.

Unit 5: Information Hiding

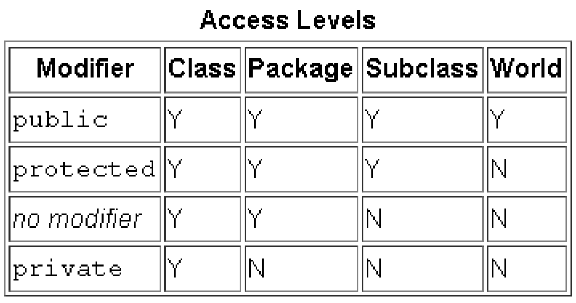
- understand the drawback of breaking the abstraction barrier
- understand the concept of information hiding to enforce the abstraction barrier
- understand how Java uses access modifiers to enforce information hiding
- understand what is a constructor and how to write one in Java

Breaking the abstraction barrier

When you break the abstraction barrier, you exposes the internal implementation of the composite data type. Ideally, we expect the end users to call the provided interface to interact/use the CDT. Suppose the you changed the internal implementation, end users, themselves, would have to change their code base as well!

Data Hiding

Java provides 4 basic access modifiers.



Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Constructor

Constructor method is a special method within the class. It cannot be called directly but is invoked automatically when an object is instantiated.

Note

It can be private (for a singleton) and can be overloaded like any other methods.

Unit 6: Tell, Don't ask

- understand what accessor and mutator are used for, and why not to use them
- understand the principle of "Tell, Don't Ask"

Accessor and mutator can potentially break encapsulation. Be careful how you use it!

The "Tell Don't Ask" Principle

If we need to know the internal and do something with it, then we are breaking the abstraction barrier. The right approach is to implement a method within the class that does whatever we want the class to do.

Unit 7: Class Fields

Unit 8: Class Methods

Unit 9: Composition

- how to compose a new class from existing classes using composition
- how composition models the HAS-A relationship
- how sharing reference values in composed objects could lead to surprising results

Classes model real-word entities in OOP. The composition models that **HAS-A** relationship between two entities.

Sharing References (aka Aliasing)

References Types may share the same reference values. This is called *aliasing*.

However, aliasing can have unexpected behaviors. Suppose we have an object P in both instances of an A class. If we change the value or update P, both instances of A would consist of the updated P object. This might not be what we want!

One solution is to avoid sharing preferences as much as possible.

The drawback of not sharing objects with the same content is that we will have a proliferation of objects and the computational resource usage is not optimized.

Another solution would be *immutability* (using `final`).

Unit 10: Heap and Stack

- understand when memory are allocated/deallocated from the heap vs. from the stack
- understand the concept of call stack in JVM

Heap and Stack

The Java Virtual Machine (JVM) manages the memory of Java programs while its bytecode instructions are interpreted and executed. A JVM implementation partitions the memory into several regions, including:

- *method area* for storing the code for the methods;
- *metaspace* for storing meta information about classes;
- *heap* for storing dynamically allocated objects;
- *stack* for local variables and call frames.

The *heap* is the region in memory where all objects are allocated in and stored, while the *stack* is the region where all variables (including primitive types and object references) are allocated in and stored.

When we invoke a method, JVM creates a *stack frame* for this instance method call.

Important

This stack frame is a region of memory that tentatively contains (i) the `this` reference, (ii) the method arguments, and (iii) local variables within the method, among other things. When a class method is called, the stack frame does not contain the `this` reference.

For example, `p1.distanceTo(p2)`.

`p1` and `this` point to the same object, and `p2` and `q` point to the same object. Within the method, any modification done to `this` would change the object referenced to by `p1`, and any change made to `q` would change the object referenced to by `p2` as well. After the method returns, the stack frame for that method is **destroyed.**

For primitive types, we copy the values onto the stack. If we change the values of the primitive types changed, the `x` and `y` of the calling function will not change.

To summarize, Java uses *call by value* for primitive types, and *call by reference* for objects.

Note

If we made multiple nested method calls, the stack frames get stacked on top of each other.

For static reference objects, draw the objects on the heap and reference it from the metaspace.

For nested/anon classes and lambdas, make sure to capture the necessary variables in the same environment (this include the arguments too!) and provide the qualified `this` keyword to the outer/enclosing class.

There is no need to draw the nested class (even though they're fields) if those classes are not instantiated.

Arguments to a lambda expression is not stored any where unless invoked.

Unit 11: Inheritance

- understand how inheritance models the IS-A relationship
- know how to use the `extends` keyword for inheritance
- understand inheritance as a subtype
- be able to determine the run-time type and compile-time type of a variable

We say that `S<:T` if any piece of code written for type `T` also works for type `S`.

We can invoke this subtype relationship by using the `extends` keyword. Unlike a parent-child relationship in real-life, however, anything private to the parent remains inaccessible to the child. This privacy veil maintains the abstraction barrier of the parent from the child, and creates a bit of a tricky situation -- i.e. a child `ColoredCircle` object has a center and a radius, but it has no access to it!

We use `super` to call the constructor of the superclass, to initialize its center and radius (since the child has no direct access to these fields that it inherited).

Models **IS-A** relationship.

Important

Use composition to model a has-a relationship; inheritance for a is-a relationship. Make sure inheritance preserves the meaning of subtyping.

Unit 12: Overriding

- be aware that every class inherits from `Object`
- be familiar with the `equals` and `toString` methods
- understand what constitutes a method signature
- understand method overriding
- appreciate the power of method overriding
- understand what Java annotations are for, and know when to use `@Override`
- be exposed to the `String` class and its associated methods, especially the `+` operator

Object and String

In Java, every class that does not extend another class inherits from the class **`Object`** implicitly. `Object` is at the root of the class hierarchy.

The `Object` class does not encapsulate anything in particular. It is a very general class that provides useful methods common to all objects. The two useful ones are:

- `equals(Object obj)`, which checks if two objects are equal to each other, and
  - Note that `x == y` is not the same as `equals()`
  - Primitives objects are copy by value, and not by reference.
- `toString()`, which returns a string representation of the object as a `String` object.

Method Overriding

Inheritance is not only good for extending the behavior of an existing class but through method overriding, we can *alter* the behavior of an existing class as well.

The *method signature* of a method refers to the method name and the number, type, and order of its parameters

The *method descriptor* of a method is the method signature plus the return type.

An instance method with the same *method descriptor* as an instance method in the parent class overrides the parent's method.

Method overriding is an example of Run Time Polymorphism.The version of a method that is executed is determined by the *target*.

Rules for method overriding

1. **Overriding Access-Modifiers :** The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.
2. **Final methods can not be overridden**
3. **Static methods can not be overridden (Method Overriding vs Method Hiding) :** When you define a static method with same signature as a static method in base class, it is known as method hiding
4. **Private methods can not be overridden**
5. **The overriding method must have same return type (or subtype).** This phenomena is known as **covariant return type**. It follows directly from LSP.
   1. Existing code that has been written to invoke the superclass' method would still work if the code invokes the subclass' method instead after the subclass inherits from the superclass. The new return type is assignable to the return type that you are overriding.
6. **Invoking overridden method from sub-class**. We can call parent class method in overriding method using super keyword.

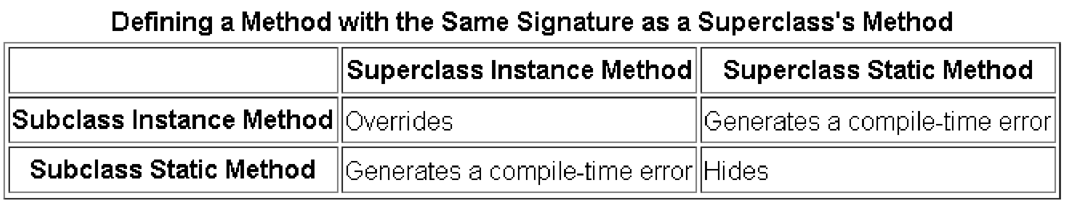
7. **Overriding constructor :** We can not override constructor as parent and child class can never have constructor with same name.
8. **Overriding Exception-Handling :**
    1. **Rule#1 :** If the super-class overridden method does not throw an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.
    2. **Rule#2 :** If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error. Also there is no issue if subclass overridden method is not throwing any exception.
9. **Overriding abstract method:** Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.

> *What if you implement two interfaces with the method `void f()` declared? What should happen? Who do Java call?*

In this case, when you override the method `f()` in the class, the issue is resolved as the two `f()` will be overridden at once.

Note

This also work if ONLY one of the interfaces implemented has a default method called f(). This will not work if both of this interfaces consist of a default implementation of f().



Defining a Method with the Same Signature as a Superclass's Method

	Superclass Instance Method	Superclass Static Method
Subclass Instance Method	Overrides	Generates a compile-time error
Subclass Static Method	Generates a compile-time error	Hides

Unit 13: Overloading

- understand what is overloading
- understand how to create overloaded methods

Method overloading

Unlike *method overriding*, *method overloading* is when we have two or more methods in the same class with the same name but a differing *method signature*1. In other words, we create an overloaded method by changing the type, order, and number of arguments of the method but keeping the method name identical.

Note

Note that only differing method signatures are needed to overload a method. You can return a different type!

```java
// The following has different method signatures
public void methodA(int A, int B) { ... }
public void methodA(Integer A, Integer B) { ... }

// The following has the same method signatures
public void methodA(int A, int B) { ... }
public void methodA(int B, int A) { ... }
```

You can overload static methods in the same way! But you cannot turn a static method to a non-static method via overloading.

You can overload `main` method like any other methods.

You cannot overload Operators.

Unit 14: Polymorphism

- understand dynamic binding and polymorphism
- be aware of the `equals` method and the need to override it to customize the equality test
- understand when narrowing type conversion and type casting are allowed

Method overriding enables *polymorphism*. 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).

Which method is invoked is decided *during run-time*, depending on the run-time type of the `obj`. This is called *dynamic binding* or *late binding* or *dynamic dispatch*.

Unit 15: Method Invocation

- understand the two step process that Java uses to determine which method implementation will be executed when a method is invoked
- understand that Class Methods do not support dynamic binding

1. The static binding occurs at compile time while dynamic binding happens at runtime.
2. Since static binding happens at an early stage of the program's life cycle, it also is known as early binding. Similarly, **dynamic binding is also known as late binding because it happens late when a program is actually running**.

During Compile Time

During compilation, Java determines the method descriptor of the method invoked, using the compile-time type of the target.

To determine the method descriptor, the compiler searches for all methods that can be correctly invoked on the given argument on **the target class and its entire hierarchy**.

Once the method is determined, the method's descriptor (return type and signature) is stored in the generated code.

During Run Time

During execution, when a method is invoked, the method descriptor from Step 1 is first retrieved. Then, the run-time type of the target is determined. Let the run-time type of the target be R. Java then looks for an accessible method with the matching descriptor in R. If no such method is found, the search will continue up the class hierarchy, first to the parent class of R, then to the grand-parent class of R, and so on, until we reach the root `Object`. The first method implementation with a matching method descriptor found will be the one executed.

Usage of Return Type in Dynamic Binding During Run-Time

Let’s consider the method invocation:

```
A a = new A();
T t = a.f();

// B <: A
R r = b.f() // possible?
```

Let’s suppose that the method descriptor chosen in Step 1 is “a method named `f`, without any parameter, and return a `T`” (i.e., `T f()`). If `T` is used in Step 2, then `T` is needed to disambiguate between different methods named `f` and without any parameter. Then, it must be possible for us to define another method named `f`, without any parameter, but returns something else other than `T`.

But is it possible to have another method with `f`, without any parameter, with a different return type?

There are only two ways to define a method named `f` in `A`: overriding and overloading. To override, our overriding method must have the same descriptor as the overridden method. So the return type must match. To overload, the method signature must be different. This means that the parameters suffice to disambiguate the methods – we do not need the return type.

So, we do not need `T` to find a matching method. As long as the method signature matches, it is fine.

Why then, does Java store the method descriptor, rather than just the method signature? It turns out that, there is an uncommon situation where the return type is required. Let's consider the following situation.

```
// In `A.java`,
class A {
  String f() {
    return "";
  }
}


// In `Main.java`,
class Main {
  public static void main(String[] args) {
    String s = new A().f();
  }
}
```

During compile-time, the invocation of `f` causes the method descriptor `String f()` to be stored in `Main.class`. Now, when we run: `java Main`, Step 2 looks for `String f()` in the run-time type of the target, `A`, (i.e., in `A.class`) and finds it.

Now, change the return type of `f()` in `A.java` to void and recompile `A.java` (without recompiling `Main.java`). Again, Step 2 looks for `String f()` in `A.class`, but all it can find is `void f()`, not `T f()`. Java gets confused and throws an error.

Here, the return type of `f()` is used – as a sanity check that the method descriptor found during compilation is still around during execution.

Invocation of Class Methods

The description above applies to instance methods. Class methods, on the other hand, do not support dynamic binding. The method to invoke is resolved statically during compile time. The same process in Step 1 is taken, but the corresponding method implementation in class C will always be executed during run-time, without considering the run-time type of the target.

Unit 16: Liskov Substitution Principle

- understand the type of bugs that reckless developers can introduce when using inheritance and polymorphism
- understand the Liskov Substitution Principle and thus be aware that not all IS-A relationships should be modeled with inheritance
- know how to explicitly disallow inheritance when writing a class or disallow overriding with the `final` keyword

Note

Liskov Substitution Principle (LSP), which says: "Let ϕ(x) be a property provable about objects x of type T. Then ϕ(y) should be true for objects y of type S where S<:T."

LSP cannot be enforced by the compiler[1]. The properties of an object have to be managed and agreed upon among programmers.

LSP Through the Lens of Testing

Designed by specification, not by implementation details.

Note

A *subclass* should not break the expectations set by the *superclass*. If a class B is substitutable for a parent class A then it should be able to pass all test cases of the parent class A. If it does not, then it is not substitutable and the LSP is violated.

Preventing Inheritance and Method Overriding

Not allowing inheritance would make it much easier to argue for the correctness of programs, something that is important when it comes to writing secure programs.

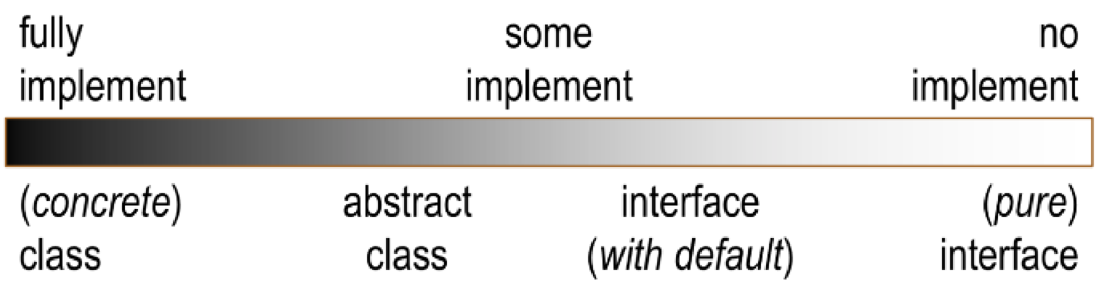
Use the keyword `final`.

Unit 17: Abstract Class

- be familiar with the concept of an abstract class
- know the use of the Java keyword `abstract` and the constraints that come with it
- understand the usefulness of defining and using an abstract class
- understand what makes a class concrete

Following are some important observations about abstract classes in Java.

1. An instance of an abstract class can not be created.
2. Constructors are allowed.
3. We can have an abstract class without any abstract method.
4. Abstract classes can not have final abstract methods because when you make a method final you can not override it but the abstract methods are meant for overriding.
5. We can define static methods in an abstract class.



Unit 18: Interface

- understand interface as a type for modeling "can do" behavior
- understand the subtype-supertypes relationship between a class and its interfaces

Since an interface models what an entity **can do**, the name usually ends with the -able suffix1 . All methods declared in an interface are `public abstract` by default.

Note

- A class can only extend from one superclass, but it can implement multiple interfaces.
- An interface can extend from one or more other interfaces, but an interface cannot extend from another class.

Interface as Super type

If a class C implements an interface I, `C<:I`. This definition implies that a type can have multiple super types.

Unit 19: Wrapper Class

- be aware that Java provides wrapper classes around the primitive types
- be aware that Java will transparently and automatically box and unbox between primitive types and their corresponding wrapper classes

Making Primitive Types Less Primitive

Java provides wrapper classes for each of its primitive types. A *wrapper class* is a class that encapsulates a *type*, rather than fields and methods.

All primitive wrapper class objects are *immutable* -- once you create an object, it cannot be changed.

Auto-boxing and Unboxing

As conversion back-and-for between a primitive type and its wrapper class is pretty common, Java provides a feature called auto-boxing/unboxing to perform type conversion between primitive type and its wrapper class.

From `int => Integer`, this is auto-boxing. From `Integer => int`, this is auto-unboxing.

Note

In general, a method that has `f(int a, int b)` is not the same as `f(Integer a, Integer b)`! They have different method signatures. However, if only one of the above methods are available, that is, only `f(int a, int b)`, and you invoke the method with `f(new Integer(1), new Integer(2))`, then java auto-unboxes the argument.

Performance

Because using an object comes with the cost of allocating memory for the object and collecting garbage afterward, it is less efficient than primitive types.

Unit 20: Run-Time Class Mismatch

- Understand the need for narrowing type conversion and type casting when writing code that depends on higher-level abstraction
- Understand the possibility of encountering run-time errors if typecasting is not done properly.

Narrowing conversion is not allowed in Java. Typecasting is a way for programmers to ask the compiler to trust the object returned by a certain method has a run-time type of `Type` (or its subtype). This also means that since the **correctness depends on the run-time type**, the compiler cannot help us. It is then up to the programmers to not make mistakes. (Permissive mindset)

Unit 21: Variance

- understand the definition of the variance of types: covariant, contravariant, and invariant.
- be aware that the Java array is covariant and how it could lead to run-time errors that cannot be caught during compile time.

Variance of Types

The *variance of types* refers to how the subtype relationship between complex types relates to the subtype relationship between components. An array of type S is an example of a complex type.

We say a complex type is:

- *covariant* if S<:T implies C(S)<:C(T)
- *contravariant* if S<:T implies C(T)<:C(S)
- *invariant* if it is neither covariant nor contravariant.

Arrays are *covariant*. Generics are *invariant*. Wildcards can be both *covariant* and *contravariant*.

Covariance of arrays can lead to *runtime errors* (*ArrayStoreException*).

Unit 22: Exceptions

- understand about handling java exceptions and how to use the `try`-`catch`-`finally` blocks
- understand the hierarchy of exception classes and the difference between checked and unchecked exceptions
- be able to create their own exceptions
- understand the control flow of exceptions
- be aware of good practices for exception handling

The `try`/`catch`/`finally` keywords group statements that check/handle errors together making code easier to read.

The error handling comes under the `catch` clauses, each handling a different type of exception. In Java, exceptions are instances that are a subtype of the `Exception` class. Information about an exception is encapsulated in an exception instance and is "passed" into the `catch` block.

With the exception, we no longer rely on a **special return value** (like null) from a function nor a global variable to indicate exceptions. What to do in the case of Functional Programming?

Finally, we have the optional `finally` clause for house-keeping tasks. This statement is *always* executed first before the next method is invoked!

In cases where the code to handle the exceptions is the same, you can avoid repetition by combining multiple exceptions into one catch statement using `|`.

Throwing Exceptions

Declare that the construct is throwing an exception, with the `throws` keyword. Create a new `IllegalArgumentException` object and throw it to the caller with the `throw` keywords.

Checked vs Unchecked Exceptions

Two types of exceptions: checked or unchecked.

An unchecked exception is an exception caused by a programmer's errors. `IllegalArgumentException`, `NullPointerException`, `ClassCastException` are examples of unchecked exceptions. They indicate that something is wrong with the program and cause run-time errors. In Java, unchecked exceptions are subclasses of the class `RuntimeException`. They **are not** explicitly thrown or caught. It is not reasonable to attempt to handle/recover from a runtime error that is due to the programmer's mistake.

A checked exception is an exception that a programmer has no control over. The programmer should thus actively anticipate the exception and handle them. For instance, when we open a file, we should anticipate that in some cases, the file cannot be opened. `FileNotFoundException` are two examples of checked exceptions. A checked exception must be either handled, or else the program will not compile.

Passing the buck

Do not pass the exception down like a waterfall. No one takes the responsibility to handle it and the user ends up with the exception. The ugly internals of the program (such as the call stack) is then revealed to the user.

A good program always handle checked exception gracefully and hide the details from the users.

Creating Our Own Exceptions

If you find that none of the exceptions provided by Java meet your needs, you can create your own exceptions, by simply inheriting from one of the existing ones. But, you should only do so if there is a good reason, for instance, to provide additional useful information to the exception handler.

Here is an example:

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

Overriding Method that Throws Exceptions

The overriding method must throw only the same, or a more specific checked exception. This follows from LSP. The caller of the overridden method cannot expect any new checked exception beyond what has already been "promised" in the method specification.

Common errors

1. Catching them all but doing nothing aka Pokémon exception handling. Since `Exception` is the superclass of all exceptions, every exception that is thrown, checked or unchecked, is now silently ignored! You will not be able to figure out if something is wrong with your program.
2. Overreacting with `Sys.exit(0)`
3. Do not break the abstraction barrier by exposing error handling. We should as much as possible handle the implementation-specific exceptions within the abstraction barrier.
4. Do not use `try-catch` as control flow!
    > 1. Such should be used to handle unexpected errors, and not to manage the logic of the program.

Note

Error <: Throwable

Unchecked Exceptions <: RunTimeExceptions <: Exceptions <: Throwable

Checked Exceptions <: Exceptions <: Throwable

Unit 23: Generics

- know how to define and instantiate a generic type and a generic method
- be familiar with the term parameterized types, type arguments, type parameters
- appreciate how generics can reduce duplication of code and improve type safety

To use Java generics effectively, you must consider the following restrictions:

- Cannot Instantiate Generic Types with Primitive Types
- Cannot Create Instances of Type Parameters
- Cannot Declare Static Fields Whose Types are Type Parameters
- Cannot Use Casts or `instanceof` With Parameterized Types
- Cannot Create Arrays of Parameterized Types
- Cannot Create, Catch, or Throw Objects of Parameterized Types
- Cannot Overload a Method Where the Formal Parameter Types of Each Overload Erase to the Same Raw Type

Note

You cannot have two of the same generic class in the same hierarchy

What is wrong with just using Object in Java?

Isolated from context - no difference. On both `t` and `obj` you can invoke only the methods of `Object`.

But with context - if you have a generic class:

```java
MyClass<Foo> my = new MyClass<Foo>();
Foo foo = new Foo();

// Generics
Foo newFoo = my.doSomething(foo);

// Object
Foo newFoo = (Foo) my.doSomething(foo);
```

Two advantages:

- No need of explicit casting (the compiler hides this from you)
- If the `Object` version is used, you won't be sure that the method always returns `Foo`. If it returns `Bar`, you'll have a `ClassCastException`, at runtime. The compiler has enough type information to check and give us an error. The compiler type-checks, before casting, to ensure no runtime errors.

How does Generics ensure Type Safety?

Generics allow classes and 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 results in a compilation error.

Differentiating Generic Type Parameters, Arguments and Parameterized Types

Parameterized Type refer to generic type that have been instantiated.

Type parameter refer to the `T` in `Array<T>`.

Type argument refer to the `String` in `Array<String>`.

Note

Object is superclass of all objects and can represent any user defined object. Since **all primitives doesn't inherit from "Object"** so we can't use it as a generic type.

Bounded Type Parameters

We can add constraint to the Type Parameter by allowing it to extend from another type. For example:

```
jshell> class Pair<T extends GetAreable, Helloable> {}
|  created class Pair
```

Here, T is a subtype of **both** GetAreable and Helloable.

Note

DO NOT say that superClassName extends superClassName, and is thus accepted. Should explain it as superClassName is upper-bounded by ? extends superClassName. The reverse is true for super.

Limitations

You cannot set a variable of a generic type as a static field for a class.

This is obvious. How would the types be decided across different instances? Recall that static methods are global. They are shared across all instances of the generic class.

Unit 24: Type Erasure

- understand that generics are implemented with type erasure in Java
- understand that type information is not fully available during run-time when generics are used, and problems that this could cause
- be aware that arrays and generics don't mix well in Java
- know the terms reifiable type and heap pollution.

Type erasure happens at **compilation time**.

Java adopts a *code sharing (erase the type parameters and arguments during compilation after type-checking) approach* as opposed to *code specialization (Creates a new type after every instantiation)* .

In the case of *code sharing*, there is only one representation of the generic type in the generated code, representing all the instantiated generic types, regardless of type arguments. This is possible due to *Type Erasure*.

One advantage of using this is the memory allocated/needed to store the same class of different types.

All instances of the Type Parameter (depending on the constraints that were added) are transformed to `Object` during compilation. When a method is invoked that returns a type `S`, Java does type-checking under the hood at **compile time**, to ensure that the underlying object is indeed of type `S` (the type you specified in the argument) before casting it.

Note that because of Type Erasure, the following not possible:

```java
// The following methods are the same after type erasure.
void foo(List<String> a) { ... }
void foo(List<Integer>a) { ... }
```

During the invocation of a method,

```java
Pair<String, Circle> p = new Pair<>();
p.getFirst();

// After type erasure, this looks

Pair p = new Pair();
// This type conversion only happens if Java compiler thinks that
// it is safe to do so.
(String)p.getFirst();
```

As mentioned, Java compiler has enough type information at compile time to ensure **type safety**.

1. Implicit casting is done for you. Despite narrowing conversion and the issues that comes with it, this is completely safe with Generics. Type conversion happens after type-checking iff Type-casting does not cause a runtime error.
2. Java does not allow Arrays and Generics to mix.
3. Generics are invariant to prevent heap pollution.

Consider this (Invalid) example:

```
Pair<String,Integer>[] pairArray = new Pair<String,Integer>[2];

// pass around the array of pairs as an array of object
Object[] objArray = pairArray;

// put a pair into the array -- no ArrayStoreException!
objArray[0] = new Pair<Double,Boolean>(3.14, true);
```

It checks that we have an array of pairs and we are putting another pair inside. Everything checks out. This would have caused a *heap pollution*, **a term that refers to the situation where a variable of a parameterized type refers to an object that is not of that parameterized type**. This would have resulted in a RunTimeException!

Generics are invariant to prevent the same problem from arising. Although the following is still possible:

```
// Creating a List of type String
List<String> listOfString = new ArrayList<>();
listOfString.add("Geeksforgeeks");

// This line would raise an unchecked warning
List<Integer> listOfInteger = (List<Integer>)(Object)listOfString;

Integer firstElement = listOfInteger.get(0); // ClassCastException here
System.out.println(firstElement);
```

Understanding the difference between reifiable and non-reifiable types

Reifiable types are types whose information are fully available during run time.

Non-reifiable types are types whose information are **not fully** available during run time.

Type information of Generics are lost after Type erasure.

Reifiable	Non-reifiable
Arrays	Generics
Wildcards	-

Unit 25: Unchecked Warnings

- be aware of how to use generics with an array
- be aware of unchecked warnings that compilers can give when we are using generics
- be able to make arguments why a piece of code is type-safe for simple cases
- know how to suppress warnings from compilers
- be aware of the ethics when using the @SuppressWarnings("unchecked") annotation
- know what is a raw type
- be aware that raw types should never never be used in modern Java

An unchecked warning is basically a message from the compiler that it has done what it can, and because of type erasures, there could be a run-time error that it cannot prevent.

Consider this example:

```
// version 0.1
class Array<T> {
  private T[] array;

  Array(int size) {
                 // Implemented as Object to be as flexible as possible
    this.array = (T[]) new Object[size];
  }

  public void set(int index, T item) {
    this.array[index] = item;
  }

  public T get(int index) {
    return this.array[index];
  }

  public T[] getArray() {
    return this.array;
  }
}
```

Where could the run time error possible come up? In this case, the underlying array is implemented as an `Object` and we choose to type-cast it to `T`. Is this safe? Can we be sure?

Because of `getArray()`, end-users can access the under-lying array and add in a type that is not allowed!

```
Array<String> array = new Array<String>(4);
Object[] objArray = array.getArray(); // ok Object -> Object
objArray[0] = 4; // ok Integer <: Object
String s = array.get(0);  // Not ok. ClassCastException, tries to convert int to String.
```

Raw Types

Another common scenario where we can get an unchecked warning is the use of *raw types*. A raw type is a generic type used without type arguments. **The compiler cannot do type-checking for us as it does not have sufficient information during compile-time regarding the type.**

By default, the type is `Object`, but without the compiler type-safety.

```
Array<String> a = new Array<String>(4);
populateArray(a); // (Array a) -> a.set(0, 1234)
String s = a.get(0);
```

Important

Raw types should never be used!

`Instanceof` is an example of type-checking at runtime!

Unit 26: Wildcards

- be aware of the meaning of wildcard `?` and bounded wildcards
- know how to use wildcards to write methods that are more flexible in accepting a range of types
- know that upper-bounded wildcard is covariant and lower-bounded wildcard is contravariant
- know the PECS principle and how to apply it
- be aware that the unbounded wildcard allows us to not use raw types in our programs

```
// In general, try to add constraints at the front, and wildcards in the argument (if possible)
public static <S, T extends S> boolean contains(Array<T> array, S obj) { .. }
```

Upper-Bounded Wildcards

An example of Upper-Bounded Wildcards

```
Array<? extends T> { ... }
```

- If `S` <: `T`, then `A<? extends S>` <: `A<? extends T>` (covariance)
- For any type `S`, `A<S>` <: `A<? extends S>`

The wildcard declaration of `List<? extends Number> foo3` means that any of these are legal assignments:

```
List<? extends Number> foo3 = new ArrayList<Number>();
List<? extends Number> foo3 = new ArrayList<Double>();
List<? extends Number> foo3 = new ArrayList<Integer>();
```

Reading - Given the above possible assignments, what type of object are you guaranteed to read from `foo3` ?

- You can read a Number because any of the lists that could be assigned to `foo3` must contain a Number or a subclass of Number.
- You can't read an Integer, because `foo3` can be pointing to a List of Doubles
- You can't read a Double, because `foo3` can be pointing to a List of Integers.

Writing - Given the above possible assignments, what type of object could you add to `foo3` that would be legal for all the above assignments ?

- You cant add anything. It may be pointing to a List of Integers when you're trying to add a `Number`.

In general, you cant add any object, because you can't guarantee what kind of List it is really pointing to, so you can't guarantee that the object is allowed in that List. The only "guarantee" is that you can only read from it and you'll get a T or a subclass of T.

Lower-Bounded Wildcards

An example of Lower-Bounded Wildcards

```
Array<? super T> { ... }
```

The lower-bounded wildcard is an example of contravariance. We have the following subtyping relations:

- If `S` <: `T`, then `A<? super T>` <: `A<? super S>` (contravariance)
- For any type `S`, `A<S>` <: `A<? super S>`

```
List<? super Integer> foo3 = new ArrayList<Number>();
List<? super Integer> foo3 = new ArrayList<Double>();
List<? super Integer> foo3 = new ArrayList<Integer>();
```

Reading - Given the above possible assignments, what type of object are you guaranteed to read from `foo3` ?

- You aren't guarantee to read an Integer because foo3 can be pointing to any of its superclass.
- The only guarantee is that you will get an instance of an Object or a subclass of an Object

Writing - Given the above possible assignments, what type of object could you add to `foo3` that would be legal for all the above assignments ?

- You can add any subclass of Integer or an Integer without issue.
- You cannot add a Double/Number/Integer/Object because `foo3` may be pointing to a subclass. No narrowing conversion!

PECS

If the variable is a producer that returns a variable of type `T`, it should be declared with the wildcard `? extends T`. Otherwise, if it is a consumer that accepts a variable of type `T`, it should be declared with the wildcard `? super T`.

If you need to produce and consume, it is best to leave it exactly with no wildcards, i.e. `List<Integer>`

This rule can be remembered with the mnemonic PECS, or "Producer Extends; Consumer Super".

Unbounded Wildcards

The unbounded Wildcards is the super type of every parameterized type.

`Array<?>` is useful when you want to write a method that takes in an array of some specific type, and you want the method to be flexible enough to take in an array of any type.

```
void foo(Array<?> array) { ... }

// We can call it like this:
Array<Circle> ac;
Array<String> as;
foo(ac); // ok
foo(as); // ok

// Pretty restrictive however
void foo(Array<?> array) {
        :
        x = array.get(0); // Only safe choice for x is Object
        array.set(0, y);  // Only safe choice to set is null.
}

// To get around the issue, you can use a helper method
void foo(Array<?> array) {
        fooHelper(array);
}

<T> void fooHelper(Array<T> arr) {
                arr.set(0, arr.get(0)); // ok due to type-inference
}
```

Unlike `Array<?>`, raw types prevent the compiler from type-checking as it does not have enough information about the type of the component of the array. It is up to the programmer to ensure type safety. For this reason, we must not use raw types.

Intuitively, we can think of `Array<?>`, `Array<Object>`, and `Array` as follows:

- `Array<?>` is an array of objects of some specific, but unknown type;
- `Array<Object>` is an array of `Object` instances, with type checking by the compiler;
- `Array` is an array of `Object` instances, without type checking.

You can instantiate an array with `<?>` like this :

```
new Comparable<?>[10]; // ok
```

The main reason behind this is `<?>` is reifiable. Since we do not know the type of ?, no information is lost during runtime. Since no information is lost, it retains full information and hence, it is reifiable.

In general, the use of wildcards allow for greater flexibility and allows functions/classes that operates the superclass/subclass of T to be passed into the method.

Unit 27: Type Inference

- be familiar how Java infers missing type arguments

Type Inference happens at **Compilation Time**.

Diamond Operator

One example of type inference is the diamond operator `<>` when we `new` an instance of a generic type:

Type Inferencing

```
public static <S> boolean contains(Array<? extends S> array, S obj) { ... }
// Instead of doing this:
A.<Shape>contains(circleArray, shape);

// you can do this
A.contains(circleArray, shape);
```

Java could still infer that `S` should be `Shape`. The type inference process looks for all possible types that match. In this example, the type of the two parameters must match. Let's consider each individually first:

- An object of type `Shape` is passed as an argument to the parameter `obj`. So `S` might be `Shape` or, if widening type conversion has occurred, one of the other supertypes of `Shape`.
- An `Array<Circle>` has been passed into `Array<? extends S>`. A widening type conversion occurred here, so we need to find all possible `S` such that `Array<Circle>` <: `Array<? extends S>`. This is true only if `S` is `Circle`, or another super type of `Circle`.

Intersecting the two lists, we know that `S` could be `Shape` or one of its supertypes: `GetAreable` and `Object`. The most specific type among these is `Shape`. So, `S` is inferred to be `Shape`.

Type inferencing can have unexpected consequences.

Consider this:

```
// Here, T is inferred to be `Object`.
public static <T> boolean contains(T[] array, T obj) { .. }

String[] strArray = new String[] { "hello", "world" };
A.<String>contains(strArray, 123); // type mismatch error
A.contains(strArray, 123); // ok, runs during compile and runtime
```

Target Typing

Type inferencing can involve the type of the expression as well. This is known as *target typing*.

Consider this:

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

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

We have a few more constraints to check:

- Due to target typing, the returning type of `T` must be a subtype of `Shape` (including `Shape`)
- Due to the bound of the type parameter, `T` must be a subtype of `GetAreable` (including `GetAreable`)
- `Array<Circle>` must be a subtype of `Array<? extends T>`, so `T` must be a super type of `Circle` (including `Circle`)

Intersecting all these possibilities, only two possibilities emerge: `Shape` and `Circle`. The most specific one is `Circle`

Unit 28: Immutability

Four ways of dealing with software complexity:

1. by encapsulating and hiding the complexity behind abstraction barriers,
2. by using a language with a strong type system and adhering to the subtyping substitution principle, and

3. applying the abstraction principles and reusing code written as functions, classes, and generics types.
4. Making your classes immutable.

Aliasing is a common problem with mutable code. A solution to this is to add `final` to your classes to prevent immutable code from being overridden.

> **Note**
>
> Making your class immutable should not have any visible change outside of the abstraction barrier. Every call of the instance method must behave the same way throughout the lifetime of the instance.

For example, consider this:

```
// Creating a new object to avoid mutating the current state, will require you to reassign the
variable.
// For example:

// MoveTo method returns a new circle object with a different center.
// Original C1 object remains to have the same center (as before)
// By reassigning, we are relying on the garbage collector to manage our memory
c1 = c1.moveTo(1,1);
```

Consider the 3 approaches we have seen so far:

1. Sharing aliases
2. Using `final` on fields to prevent override
3. Reassigning

The first shares all the references and is bug-prone. The second creates a new copy of the instance every time and is resource-intensive. The third approach, using immutable classes, allows us to share all the references until we need to modify the instance, in which case we make a copy (and reassigns to the original reference, the old reference is removed by the garbage collector).

Such a *copy-on-write* semantic allows us to avoid aliasing bugs without creating excessive copies of objects.

Advantages of being immutable

Ease of Understanding

With immutability, we ensure that the object instance `c` that we have created remains the same for its entire lifetime, unless we explicitly reassigns the value. This makes it easier to read, understand, and debug our code.

Without this property, we have to trace through all the methods that we pass `c` to, and each call of `c`'s methods to make sure that none of these codes modifies `c`.

Enabling Safe Sharing of Objects

Making a class immutable allows us to safely share instances of the class and therefore reducing the need to create multiple copies of the same object. For instance, think about the CACHED_NONE used in lab 5 and 6. If the instance is immutable, we can just create and cache a single copy, and always return this copy when the value is required.

> **Note**
>
> Such a design pattern is only safe when the class is **immutable**. Consider the mutable version of `Point` -- calling `Point.of(0, 0).moveTo(1, 1)` would change every reference to the origin to (1, 1), causing chaos in the code!

Enabling Safe Sharing of Internals

Immutable instances can also share their internals freely. Consider an immutable implementation of our `Array<T>`, called `ImmutableArray<T>`.

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

        // T... syntactic sugar for passing in an array of items to a method.
        // Safe as there is no way for us to modify the contents of the array.
  @SafeVarargs
  public static <T> ImmutableArray<T> of(T... items) {
    return new ImmutableArray<>(items, 0, items.length-1);
  }

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

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

        // Create a new immutableArray that can only access a sub-portion
  public ImmutableArray<T> subarray(int start, int end) {
     return new ImmutableArray<>(this.array, this.start + start, this.start + end);
  }
}
```

@SafeVarargs Since the varargs is just an array, and array and generics do not mix well in Java, the compiler would throw us an unchecked warning. In this instance, however, we know that our code is safe because there is no way to modify the items of type `T` in the array. We can use the `@SafeVarargs` annotation to tell the compiler that we know what we are doing and this varargs is safe.

A typical way to implement `subarray` is to allocate a new `T[]` and copy the elements over. This operation can be **expensive** if our `ImmutableArray` has millions of elements. But, since our class is immutable and the internal field `array` is guaranteed not to mutate, we can safely refer to the same `array` from `a`, and only store the starting and ending index.

Enabling Safe Concurrent Execution

Concurrent programming allows multiple threads of code to run in an interleaved fashion, in an arbitrary interleaving order. If we have complex code that is difficult to debug to begin with, imagine having code where we have to ensure its correctness regardless of how the execution interleaves!

Immutability helps us ensure that regardless of how the code interleaves, our objects remain unchanged.

Unit 29: Nested Class

A nested class is a **field** of the containing class.

- **It is a way of logically grouping classes that are only used in one place**: They usually have no use outside of the container class. Nested classes can be used to encapsulate information within a container class when the implementation of the container class becomes too complex.

- **It increases encapsulation**: Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared `private`. By hiding class B within class A, A's members can be declared private and B can **access** them. In addition, B itself can be hidden from the outside world. Only do this if B is in the same encapsulation as A, otherwise, the container class would leak its implementation details to B.

- **It can lead to more readable and maintainable code**: Nesting small classes within top-level classes places the code closer to where it is used.

> **Note**
>
> Note that the containing class is able to access the private fields of the nested class

A nested class can be either static or non-static. Just like static fields and methods, a *static nested class* is associated with the containing *class*, **NOT** an *instance*. So, it can only access static fields and static methods of the containing class. A *non-static nested class*, on the other hand, can access all fields and methods of the containing class. A *non-static nested class* is also known as an *inner class*.

Local Class

We can also declare a class within a function, just like a local variable.

Suppose we have a list of strings, and we want to sort them in the order of their length, we can write the following method:

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

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

  names.sort(new NameComparator()); // sort method takes in an object
}
```

This makes the code easier to read since we keep the definition of the class and its usage closer together.

Classes like `NameComparator` that are declared inside a method (inside a block of code between `{` and `}`) is called a *local class*. Just like a local variable, a local class is scoped within the method. Like a nested class, a local class has access to the variables of the enclosing class through the qualified `this` reference. Further, it can access the local variables of the enclosing method.

Variable Capture + Effectively final

Recall that when a method returns, all local variables of the methods are removed from the stack. But, an instance of that local class might still exist. Consider the following example:

```
class A {
  int x = 1;

  C f() {
    int y = 2; // effectively final

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

    B b = new B();
    return b;
  }
}
A a = new A();
```

```
// After method execution, local method variables are removed.
C b = a.f();
b.g(); // How is this safe?
a.x;
```

Calling will give us a reference to an object of type `B` now. But, if we call `b.g()`, what is the value of `y`?

The function may have completed its execution and removed from the process stack, with all the variables destroyed but it may be the case that objects of the inner class are still on the heap referencing a particular local variable of that function. To counter this, Java **makes a copy** (*variable capture*) of the local variable and gives that as a reference to the inner class.

Note

Local classes only capture the variables it needs!

To maintain consistency between the 2 copies, the local variable is mandated to be "**final**" and non-modifiable. An implicitly final variable does not change after initialization. Therefore, there is no confusion whether changes to one variable would affect the variable inside the local class.

Effectively final are only for variables on the same stack frame! This does not include the enclosing class fields.

Note

This is only effective for immutable objects. You can still mutate mutable objects!

Anonymous class

An anonymous class is one where you declare a class and instantiate it in a single statement. It's anonymous since We do not even have to give the class a name.

An anonymous class has the following format: `new X (arguments) { body }, where:

- *X* is a class that the anonymous class extends or an interface that the anonymous class implements. X cannot be empty. This syntax also implies an anonymous class cannot extend another class and implement an interface at the same time.
- *arguments* are the arguments to the constructor of the anonymous class. If the anonymous class is extending an interface, then there is no constructor, but we still need `()`.
- *body* is the body of the class, except that we cannot have a constructor for an anonymous class.

An anonymous class is just like a local class, it captures the variables of the enclosing scope as well -- the same rules to variable access as local classes applies.

```
class A {
        private int x = 1; // not effectively final
        void foo() {
                int y = 1; // effectively final

                Hello h = new Hello() {
                        void sayHi() {
                                System.out.println(y + x);
                        }
                };

                h.sayHi();
        }
        public static void main(String[] args) { new A().foo(); } }

interface Hello { void sayHi(); }
```

Anonymous classes also have the same restrictions as local classes with respect to their members:

- You cannot declare static initializers or member interfaces in an anonymous class.
- An anonymous class can have static members provided that they are constant variables.

Note that you can declare the following in anonymous classes:

- Fields
- Extra methods (even if they do not implement any methods of the supertype)
- Instance initializers
- Local classes

However, you cannot declare constructors in an anonymous class.

Unit 30: Side Effect-Free Programming

- the concept of functions as side-effect-free programming constructs and its relation to functions in mathematics.
- understand the importance of writing code that is free of side effects
- how functions can be first-class citizens in Java through using local anonymous class
- how we can succinctly use a lambda expression or a method reference in place of using local anonymous class
- how we can use currying to generalize to functions with higher arity
- how we can create a closure with lambda and environment capture

Functions

1. Free from side-effects
2. Referential Transparency

Mathematical functions are free from *side-effects*. They simply compute and return the value. These functions do not change `x` nor any other unknowns `y`, `z` etc.

Let f(x) = a. With *referential transparency*, any occurrences of f(x), we can replace them with a. The converse holds true.

Writing code that are free from side-effects allows us to reason it the same way as we reason about mathematical functions. However, the mutable nature of certain complex types does not ensure this.

Consider the example: `T t = a.get(0);

The program above is not always true as the underlying array is mutable and maybe change. This violates property 2 (we cannot replace all occurrences of `a.get(0)` with just `t`)

With the above stated properties, our programs can be chained and composed together. To achieve this, functions need to be first class citizens in our program, so that we can assign functions to a variable, pass it as parameters and return a function from another function, etc. Just like an integer.

Pure functions

Returns an output without any of the following:

1. Print to the screen
2. Write to files
3. Throw any possible exceptions
4. Change other variables
5. Modify the values of the arguments
6. Non-deterministic

A pure function must also be deterministic. Given the same input, the function must produce the same output, *every single time*. This deterministic property ensures referential transparency

In the OOP paradigm, we commonly need to write methods that update the fields of an instance or compute values using the fields of an instance. Such methods are not pure functions. On the other hand, if our class is immutable, then its methods must not have side effects and thus is pure.

Functions as first class citizens

```
// Transform an object of T to an object of R
<T, R, S> Transformer<T,R> chain(Transformer<? super T, ? extends S> t1,

Transformer<? super S,? extends R> t2) {
  return new Transformer<T,R>() {
    public R transform(T value) {
      return t2.transform(t1.transform(value));
    }
  }
}
```

We can think of **an instance of the anonymous class as the function**. Since a function is now just an instance of an Object in java, we can pass it around, return it from a function, and assign it to a variable, just like any other reference types. We can now compose two functions together like we would in mathematics.

Lambda Expression

An interface in Java with only one abstract method is called a *functional interface*. If a programmer intends an interface to be a functional interface, they should annotate the interface with the `@FunctionalInterface` annotation.

A key advantage of a functional interface is that there is **no ambiguity** about which method is being overridden by an implementing subclass.

We can simplify the above example to:

```
// Transform an object of T to an object of R
<T, R, S> Transformer<T,R> chain(Transformer<? super T, ? extends S> t1,

Transformer<? super S,? extends R> t2) {
  return (T value) -> t2.transform(t1.transform(value));
}
```

Method Reference

The double-colon notation `::` is used to specify a *method reference*. We can use method references to refer to a (i) static method in a class, (ii) instance method of a class or interface, (iii) constructor of a class. Here are some examples (and their equivalent lambda expression)

```
Box::of              // x -> Box.of(x)
Box::new             // x -> new Box(x)
x::compareTo         // y -> x.compareTo(y)
A::foo               // (x, y) -> x.foo(y) or (x, y) -> A.foo(x,y)
```

The last example shows that the same method reference expression can be interpreted in two different ways.

The actual interpretation depends on how many parameters `foo` takes and whether `foo` is a class method or an instance method. When compiling, Java searches for the matching method, performing type inferences to find the method that matches the given method reference. A compilation error will be thrown if there are multiple matches or if there is ambiguity in which method matches.

An example:

```
class A {
        public Integer addOne(Integer y) {
                return y + 1;
 }

 public static void main(String[] args) {
   A a = new A();
   Transformer<Integer, Integer> dist = A::addOne;
   Integer x = dist.transform(a, 2);
   System.out.println(x);
 }
}

@FunctionalInterface
public interface Transformer<T, U> { public U transform(A a, T t); }
```

Lambda as Closure

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

the **variable `origin` is captured** by the lambda expression `dist`. Just like in local and anonymous classes, a captured variable must be either explicitly declared as `final` or is effectively final.

A lambda expression stores more than just the function to invoke -- it also stores the data (including any declared variables wider than its current scope!) from the environment where it is defined. We call such a construct that stores a function together with the enclosing environment a *closure*.

This allows us to save the current execution environment, then continue to compute it later.

- Allows for cleaner code with fewer parameters to pass around and less duplicated code.
- Separate the logic to do different tasks in a different part of program easier.

Currying

```
Transformer<Integer, Transformer<Integer, Integer>> add = x -> y -> (x + y);
// Compute the first argument first
Transformer<Integer,Integer> incr = add.transform(1);
```

The technique that translates a general n-ary function to a sequence of n unary functions is called *currying*. After currying, we have a sequence of *curried* functions.

Currying is useful if one of the argument

1. is not available until later on
2. does not change very often
3. expensive to compute

We can save the partial results as a function and continue applying later. We can dynamically create functions as needed, save them, and invoke them later.

Unit 31: Box and Maybe

- the generality of the class `Box<T>` and `Maybe<T>`
- how passing in functions as parameter can lead to highly general abstractions
- how `Maybe<T>` preserves the "maybe null" semantics over a reference type by internalizing checks for `null`

Lambda as a Cross-Barrier State Manipulator

Altering the internal state without the use of setters and getters. We can do this by providing methods that accept a lambda expression, and apply the expression on the item.

Allows the client to manipulate data behind the abstraction barrier without knowing the internals of the object, treating lambda expressions as "manipulators" that they can pass in and modify the internals, while maintaining encapsulation.

Options type

The `Maybe<T>` abstraction allows us to write code without mostly worrying about the possibility that our value is missing.

Suppose we invoked `Counter c = shop.findCounter()`.

The `findCounter` method is a mapping from the domain of shops to counters. If we implement `findCounter` such that it returns `null` if no counter is available, then `findCounter` is not a **pure function** anymore. The return value `null` is not a counter, as we cannot do things that we can normally do on counters to it. So `findCounter` now maps to a value outside its codomain. This violation of the purity of function adds complications to our code, as we now have to **specifically** filter out `null` value, and is a common source of bugs.

This **violates** property 2 (referential transparency), as replacing occurrences of `c` might lead to a `nullPointerException`.

1. One way to fix this is to have a special counter (say, `class NullCounter extends Counter`) that is returned whenever there is no available counters. However, this means that everywhere we return null in place of a non-null instance we have to create a special subclass.
2. Another way is to expand the codomain of the function to include `null`, and wrap both `null` and `Counter` under a type called `Maybe<Counter>`. We make `findCounter` returns a `Maybe<Counter>` instead

With this design, the domain of `findCounter` is now `Shop` to codomain `Maybe<Counter>`, and it is pure.

Note

Another way to view the `Maybe<T>` class is that it internalizes all the checks for `null` on the client's behalf. `Maybe<T>` ensures that if `null` represents a missing value, then the semantics of this missing value is preserved throughout the chain of `map` and `filter` operations. Within its implementation, `Maybe<T>` do the right thing when the value is missing to prevent us from encountering `NullPointerException`. There is a check for `null` when needed, internally, within `Maybe<T>`. This internalization removes the burden of checking for `null` on the programmer and removes the possibility of run-time crashes due to missing `null` checks.

Lazy evaluation

- what is lazy evaluation and how lambda expression allows us to delay the execution of a computation
- how memoization and the `Lazy<T>` abstraction allows us to evaluate an expression exactly once.

Lambda as Delayed Data

When a lambda function is declared, nothing is executed yet. We are just saving them to be executed later. This enables **Lazy Evaluation**. We can build up a sequence of complex computations, without actually executing them, until we need to. Expressions are evaluated on demand when needed.

A good design is to wrap the expression within a lambda expression, so that it does not get evaluated eagerly when we pass it in as parameter.

Memoization

Not repeat ourselves. If we have computed the value of the function before, we can cached the value, keep it somewhere, so that we don't need to compute it again.

Note

Only useful if the function is pure -- deterministic, and invoking it has no side effects on the execution of the program.

Unit 33: Infinite List

Unit 34: Streams

Terminal Operations

A Stream is lazy, just like `InfiniteList`. A terminal operation is an operation on the stream that **triggers the evaluation of the stream**.

- chain a series of intermediate operations together (**ALL LAZY OPERATIONS**)
- end with a terminal operation.

The `forEach` method is a terminal operation that applies a lambda expression to each element of the stream. The lambda expression to apply does not return any value. Java provides the `Consumer<T>` functional interface for this.

Intermediate Stream Operations

An intermediate operation on stream returns another **Stream**. Java provides `map`, `filter`, `flatMap`, and other intermediate operations. Intermediate operations are lazy and do not cause the stream to be evaluated.

FlatMap a Stream

Takes a lambda expression that transforms every element in the stream into another stream. The resulting stream of streams is then flattened and concatenated together.

Stateful and Bounded Operations

Some intermediate operations are stateful -- they need to keep track of some states to operate.

- **sorted** returns a stream with the elements in the stream sorted.
  - Without argument, it sorts according to the natural order as defined by implementing the Comparable interface.
- **distinct** returns a stream with only distinct elements in the stream.

distinct and **sorted** are also known as bounded operations, since they should only be called on a finite stream -- calling them on an infinite stream is a bad idea!

Truncating an Infinite List

Convert from infinite stream to finite stream. These are intermediate operations.

- `limit` takes in an `int` and **returns a stream** containing the first `n` elements of the stream;
- `takeWhile` takes in a predicate and **returns a stream** containing the elements of the stream, until the predicate becomes false. The resulting stream might still be infinite if the predicate never becomes false.

Peeking with a Consumer

A particularly useful **intermediate** operation of Stream is `peek`. `peek` takes in a `Consumer`, allowing us to apply a lambda on a "fork" of the stream.

Reducing a Stream

One of the more powerful terminal operations in Stream is `reduce`. The `reduce` operation applies a lambda repeatedly on the elements of the stream to reduce it into a single value.

Element Matching

Stream also provides **terminal** operations for testing if the elements pass a given predicate:

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

Consumed Once

A stream can only be operated on once. We cannot iterate through a stream multiple times. Doing so would lead to an `IllegalStateException` being thrown. We have to recreate the stream if we want to operate on the stream more than once.

Unit 35: Loggable Class

Unit 36: Monads

- Understand what are functors and monads
- Understand the laws that a functor and monad must obey and be able to verify them

Note

Whether a function is pure or not has nothing to do with whether the class is a monad!

Generalizing `Loggable<T>`

We have just created a class with a flatMap method that allows us to operate on the value encapsulated inside, along with some "side information".

- an `of` method (or equivalent) to initialize the value and side information.
  - For example, `Lazy(( ) -> x)` is an appropriate unit operator.
- have a `flatMap` method (or equivalent) to update the value and side information.

Different classes above have different side information that is initialized, stored, and updated when we use the of and flatMap operations.

- `Maybe<T>` stores the side information of whether the value is there or not there.
- `Lazy<T>` stores the side information of whether the value has been evaluated or not.
- `InfiniteList<T>` stores the side information that the values in the list may or may not be evaluated. (Might not be a monad as it does not support the of nor the flatMap method)
- `Loggable<T>` stores the side information of a log describing the operations done on the value.

Our `of` method **should not do anything extra** to the value and side information -- it should simply wrap the value into the `Monad`.

Our `flatMap` method **should not do anything extra** to the value and the side information, it should simply apply the given lambda expression to the value.

Identity Laws

To be more general, let monad be a type that is a Monad and monad be an instance of it.

The left identity law says:

`Monad.of(x).flatMap(x => f(x))` must be the same as `f(x)`

The right identity law says:

`monad.flatMap(x => Monad.of(x))` must be the same as `monad`

Associative Law

Regardless of how we group that calls to flatMap, their behavior must be the same. This law is called the associative law. More formally, it says:

```
monad.flatMap(x -> f(x)).flatMap(x -> g(x)) == monad.flatMap(x -> f(x).flatMap(y -> g(y)))
```

If our monads follow the laws above, we can safely write methods that receive a monad from others, operate on it, and return it to others. We can also safely create a monad and pass it to the clients to operate on. Our clients can then call our methods in any order and operate on the monads that we create, and the effect on its value and side information is as expected.

Functors

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

A functor needs to adhere to two laws:

- preserving identity: `functor.map(x -> x)` is the same as functor
- preserving composition: `functor.map(x -> f(x)).map(x -> g(x)) is the same as functor.map(x -> g(f(x))`.

Our classes from cs2030s.fp, `Lazy<T>`, `Maybe<T>`, and `InfiniteList<T>` are functors as well.

Why do we need monads?

1. We want to program **only using functions**.
2. Then, we have a first big problem.

   How can we say **what is to be executed first**? How can we form an ordered sequence of functions (i.e. a program) *using no more than functions*?

   Solution: compose functions. If you want first `g` and then `f`, just write `f(g(x,y))`. This way, "the program" is a function as well: `main = f(g(x,y))`.
3. More problems: some functions **might fail** (i.e. `g(2,0)`, divide by 0). We have **no "exceptions"** in FP (an exception is not a function). How do we solve it?

   Solution: Let's **allow functions to return two kind of things**: instead of having `g : Real, Real -> Real` (function from two reals into a real), let's allow `g : Real, Real -> Real | Nothing` (function from two reals into (real or nothing)).
4. But functions should (to be simpler) return only **one thing**.

   Solution: let's create a new type of data to be returned, a "**boxing type**" that encloses maybe a real or be simply nothing. Hence, we can have `g : Real, Real -> Maybe Real`.
5. What happens now to `f(g(x,y))`? `f` is not ready to consume a `Maybe Real`. And, we don't want to change every function we could connect with `g` to consume a `Maybe Real`.

   Solution: let's **have a special function to "flatMap"/"compose"/"link" functions**. That way, we can, behind the scenes, adapt the output of one function to feed the following one.

   In our case: `g >>= f` (connect/compose `g` to `f`). We want `>>=` to get `g`'s output, inspect it and, in case it is `Nothing` just don't call `f` and return `Nothing`; or on the contrary, extract the boxed `Real` and feed `f` with it. (This algorithm is just the implementation of `>>=` for the `Maybe` type). Also note that `>>=` must be written **only once** per "boxing type" (different box, different adapting algorithm).
6. Many other problems arise which can be solved using this same pattern: 1. Use a "box" to codify/store different meanings/values, and have functions like `g` that return those "boxed values". 2. Have a composer/linker `g >>= f` to help connecting `g`'s output to `f`'s input, so we don't have to change any `f` at all.
7. Remarkable problems that can be solved using this technique are:

   > - having a global state that every function in the sequence of functions ("the program") can share: solution `StateMonad`.
   > - We don't like "impure functions": functions that yield *different* output for *same* input. Therefore, let's mark those functions, making them to return a tagged/boxed value: `IO` monad.

Unit 37: Parallel Streams

- be aware that a program can be broken into subtasks to run parallelly and/or concurrently
- be aware of the issues caused by running subtasks parallelly and concurrently.
- be aware that there exist tradeoffs in the number of subtasks and the processing overhead.
- be familiar with how to process a stream parallelly and correctly.

Parallel and Concurrent Programming

What is concurrency?

A single-core processor can only execute one instruction at one time -- this means that only one process can run at any one time.

We can write a program so that it runs concurrently -- by dividing the computation into subtasks called threads. Such multi-thread programs are useful in two ways:

1. It allows us, the programmers, to separate unrelated tasks into threads, and write each thread separately;
2. it improves the utilization of the processor. For instance, if I/O is in one thread, and UI rendering is in another, then when the processor is waiting for I/O to complete, it can switch to the rendering thread to make sure that the slow I/O does not affect the responsiveness of UI.

What is parallelism?

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

All parallel programs are concurrent, but not all concurrent programs are parallel.

That is,

$$Parallel \subseteq Concurrency$$

Parallel Stream

You can convert a sequential stream into a parallel one using `.parallel().`

The order of execution should not be critical in determining the success of the program.

Parallel Stream breaks the list into multiple subsequences, and run the code for each subsequence in parallel. Since there is no coordination among the parallel tasks on the order of the printing, whichever parallel tasks that complete first will output the result to screen first, causing the sequence of numbers to be reordered.

If you want to produce the output in the order of input, use `forEachOrdered` instead of `forEach`, we will lose some benefits of parallelization because of this.

```
IntStream.range(2_030_000, 2_040_000).filter(x -> isPrime(x)).parallel().count();
```

The code above produces the same output regardless if it is being parallelized or not.

The task above is stateless and does not produce any side effects. Each element is processed individually without depending on other elements. Such computation is sometimes known as **embarrassingly parallel**. The only communication needed for each of the parallel subtasks is to combine the result of count() from the subtasks into the final count (which has been implemented in Stream for us).

What can be parallelized?

To ensure that the output of the parallel execution is correct, the stream operations must not interfere with the stream data, and most of the time must be stateless. Side-effects should be kept to a minimum.

Order of execution

Each element should be processed independent from other elements. Embarrassingly Parallel.

Interference

Interference means that one of the stream operations modifies the source of the stream DURING the execution of the terminal operation. For instance:

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

would cause `ConcurrentModificationException` to be thrown. Note that this non-interference rule applies even if we are using stream() instead of `parallelStream()`.

Note that the following interpretation is wrong:

> *"Interference means that one of the stream operations modifies the source of the stream BEFORE the executions of the terminal operation.", which is why we get a `ConcurrentModificationException`?*

Since intermediate operations are lazy, nothing gets executed until we invoke a terminal operation. So, before a terminal operation is called, nothing gets modified.

Stateful vs. Stateless

A stateful lambda is one where the result depends on any state that might change during the execution of the stream. May or may not be critical. `reduce(identity, (result, x) -> result + x )`

For instance, the **generate and map operations below are stateful**, since they depend on the state of the standard input. To ensure that the output is correct, additional work needs to be done to ensure that state updates are visible to all parallel subtasks.

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

Side effects

Side-effects **can lead to incorrect results** in parallel execution but **not necessary**. Consider the following code:

```
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 what we call a non-thread-safe data structure. If two threads manipulate it at the same time, an incorrect result may result.

There are two ways to resolve this.

1. the `.collect` method.
2. use a thread-safe data structure.

```
// First
list.parallelStream() .filter(x -> isPrime(x)) .collect(Collectors.toList())

// Second
List<Integer> result = new CopyOnWriteArrayList<>();
list.parallelStream() .filter(x -> isPrime(x)) .forEach(x -> result.add(x));
```

Associativity

The reduce operation is inherently parallelizable, as we can easily reduce each sub-stream and then use the combiner to combine the results. Consider this example:

```
// identity, accumlator, combiner
Stream.of(1,2,3,4).reduce(1, (x, y) -> x * y, (x, y) -> x * y);
```

There are several rules that the identity, the accumulator, and the combiner must follow:

- `combiner.apply(identity, i)` must be equal to `i`.
- The combiner and the accumulator must be **associative** -- **the order of applying must not matter.** That is, (x *y)* z equals x *(y* z).
- The combiner and the accumulator must be compatible -- `combiner.apply(u, accumulator.apply(identity, t))` must equal to `accumulator.apply(u, t)` => u *(1* t) equals u * t

Performance of Parallel Streams

Parallelizing a stream does not always improve the performance. Creating a thread to run a task incurs some overhead, and the overhead of creating too many threads might outweigh the benefits of parallelization.

Ordered vs. Unordered Source

Whether or not the stream elements are ordered or unordered also plays a role in the performance of parallel stream operations. Streams created from iterate, ordered collections (e.g., List or arrays), from of, are ordered. Stream created from generate or unordered collections (e.g., Set) are unordered.

Some stream operations respect the encounter order. For instance, both **distinct** and **sorted** are stable operations (preserve the original order of elements).

The parallel version of `findFirst`, `limit`, and `skip` can be expensive on an ordered stream, since it needs to **coordinate between the streams to maintain the order**.

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

Unit 38: Threads

Synchronous Programming

We wait for the method to complete its execution while our program stalls. Only after the method returns can the execution of our program continue. We say the method blocks until it returns. Such is known as *synchronous programming*.

However, this is not very efficient especially when there are frequent method calls that takes a very long time (hence, blocking the execution of the rest of our programs.)

Threads

One way to achieve this is to use *threads*. A thread is a **single flow of execution** in a program.

Java provides a class called **`java.lang.Thread`** that we can use to encapsulate a function to run in a separate thread. The `new Thread(...)` is how we can construct a `Thread` instance. The constructor takes in a `Runnable` instance as an argument. `Runnable` is invoked by calling `run()` and returns `void`.

In each `Thread` instance, we run `start()`, which causes the given lambda expression to run. **Note that `start()` returns immediately** without waiting for the lambda expression to complete its execution. This is known as *asynchronous* execution.

The two threads above now run in two separate sequences of execution. The operating system has a scheduler that decides which threads to run when, and on which core (or which processor), resulting in different interleaving of executions every time you run the same program.

Names

Every thread in Java has a name, printing out its name is useful for peeking under the hood to see what is happening. We can use the instance method `Thread.currentThread().getName()` to get the reference of the current running thread.

The thread `main` is a thread that is created for us every time our program runs and the class method `main()` is invoked.

Sleep

Another useful method in the `Thread` class is `sleep`. You can cause 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.

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

Note

Two more things to note:

- We can use `isAlive()` to periodically check if another thread is still running.
- The program exits only after all the threads created run to their completion.

Unit 39: Asynchronous Programming

Limitations of Thread

Consider the situation where we have a series of tasks that we wish to execute ***concurrently*** and we want to organize them such that there are certain sets of dependencies.

1. Handle exceptions gracefully -- if one of the tasks encounters an exception, the other tasks not dependent on it should still be completed.
2. Implementing the above using `Thread` requires careful coordination.
    1. No methods in `Thread` that return a value. We need the threads to communicate through shared variables.
    2. There is no mechanism to specify the execution order and dependencies among them -- which thread to start after another thread completes.
    3. We have to consider the possibility of exceptions in each of our tasks.
3. Another drawback of using `Thread` is its overhead -- **the creation of `Thread` instances takes up some resources in Java**.
    1. As much as possible, we should reuse our `Thread` instances to run multiple tasks.

Managing the `Thread` instances itself and deciding which `Thread` instance should run which `Thread` is a gigantic undertaking.

A Higher-Level Abstraction

`java.util.concurrent.CompletableFuture` is a monad that provides a higher-level of abstraction, allowing programmers to focus on specifying the tasks and their dependencies, without worrying about the details.

```
CompletableFuture<Integer> foo(int x) {
    CompletableFuture<Integer> a = CompletableFuture.completedFuture(x);
    CompletableFuture<Integer> b = a.thenComposeAsync(i -> taskB(i));
    CompletableFuture<Integer> c = a.thenComposeAsync(i -> taskC(i));
    CompletableFuture<Integer> d = a.thenComposeAsync(i -> taskD(i));
    CompletableFuture<Integer> e = b.thenCombineAsync(c, (i, j) -> taskE(i, j));
    return e;
}
```

We can then run `foo(x).get()` to wait for all the concurrent tasks to complete and return us the value.

The `CompletableFuture` Monad

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

Creating a `CompletableFuture`

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

- Use the `completedFuture` method. This method is equivalent to creating a task that is **already completed** and return us a value.
- 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. **Similar to RUN**
- Use the `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. **Similar to PRODUCE**

A new `CompletableFuture` created with `allOf` is completed only when **all** the given `CompletableFuture` completes. On the other hand, a new `CompletableFuture` created with `anyOf` is completed when **any** one of the given `CompletableFuture` completes.

Chaining `CompletableFuture`

The usefulness of `CompletableFuture` comes from the ability to chain them up and specify a sequence of computations to be run. We have the following methods:

- `thenApply`, which is analogous to `map`
- `thenCompose`, which is analogous to `flatMap`
- `thenCombine`, which is analogous to `combine
- `thenRun` is a blocking call for the CompletableFuture that it is chained with.

The methods above run the given lambda expression in the same thread as the caller. There is also an asynchronous version (`thenApplyAsync`, `thenComposeAsync`, `thenCombineAsync`), which may cause the given lambda expression to run in a different thread (thus more concurrency).

What's the advantage of using one over the other? The `Async` version simply allows the current task to be picked up by another thread while the current thread does some other stuff.

If the task is dependent on the previous call, `thenApply` is the way to go. Otherwise, `thenApplyAsync` is a better choice as the tasks don't block the thread anymore.

```
// Both tasks dependent on a1, but not on each other.
// You can let them run in different threads.
a1.thenApplyAsync(x -> x.incr());
a1.thenApplyAsync(x -> x .inrc());
```

Getting The Result

We can call `get()` to get the result but `get()` is a **synchronous call**, i.e., it blocks until the `CompletableFuture` completes, to maximize concurrency, we should only call `get()` as the final step in our code.

The method `CompletableFuture::get` throws a couple of checked exceptions: `InterruptedException` and `ExecutionException`, which we need to catch and handle. The former refers to the exception that the thread has been interrupted, while the latter refers to errors/exceptions during execution.

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

```
CompletableFuture<Integer> ith = CompletableFuture.supplyAsync(() -> findIthPrime(i));
CompletableFuture<Integer> jth = CompletableFuture.supplyAsync(() -> findIthPrime(j));

CompletableFuture<Integer> diff = ith.thenCombine(jth, (x, y) -> x - y);

diff.join();
```

Handling Exceptions

One of the advantages of using `CompletableFuture<T>` instead of `Thread` to handle concurrency is its ability to handle exceptions. `CompletableFuture<T>` has three methods that deal with exceptions: `exceptionally`, `whenComplete`, and `handle`.

The computation is asynchronous and could run in a different thread, `CompletableFuture<T>` keeps things simpler by storing the exception and passing it down the chain of calls, until `join()` is called. `join()` might throw `CompletionException` and whoever calls `join()` will be responsible for handling this exception. The `CompletionException` contains information on the original exception.

For instance, the code below would throw a `CompletionException` with a `NullPointerException` contains within it.

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

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

We can use the `handle` method, to handle the exception. The `handle` method takes in a `BiFunction` (similar to `cs2030s.fp.Combiner`). The first parameter to the `BiFunction` is the value, the second is the exception, the third is the return value.

Only one of the first two parameters is not `null`. If the value is `null`, this means that an exception has been thrown. Otherwise, the exception is `null`2.

Question

Practice Question: What outputs is guaranteed?

```
printAsync(1);
CompletableFuture.allOf(printAsync(2), printAsync(3)).join();

The answer is 123, 321, 23, 32.

Note that printAsync(1) is not synchronized so it can get printed anytime, or not printed at
all before the program exits. Since there is a call to join() for printAsync(2) and
printAsync(3), we know for sure that 2 and 3 will be printed.
```

Unit 40: Fork and Join

Thread Pool

Creating and destroying threads is not cheap, hence we should reuse existing threads to perform different tasks. This goal can be achieved by using a *thread pool*.

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 and Join

Java implements a thread pool called `ForkJoinPool` that is fine-tuned for the fork-join model of recursive parallel execution.

The Fork-join model is essentially a parallel divide-and-conquer model of computation. 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 (*fork*), then solve the smaller version of the problem recursively, then combine the results (*join*). This repeats recursively until the problem size is small enough -- we have reached the base case and so we just solve the problem sequentially without further parallelization.

In Java, we can create a task that we can fork and join as an instance of abstract class `RecursiveTask<T>`. `RecursiveTask<T>` supports the methods `fork()`, which submits a smaller version of the task for execution, and `join()` (which waits for the smaller tasks to complete and return). `RecursiveTask<T>` has an abstract method `compute()`, which we, as the client, have to define to specify what computation we want to compute.

Here is a simple `RecursiveTask<T>` that recursively sums up the content of an array:

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

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

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

      int middle = (low + high) / 2;
      Summer left = new Summer(low, middle, array);
      Summer right = new Summer(middle, high, array);
      left.fork();
      return right.compute() + left.join();
    }
  }


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

The line `task.compute()` above is just like another method invocation. It causes the method `compute()` to be invoked, and if the array is big enough, two new `Summer` instances, `left` and `right`, to be created. `left`.

We then call `left.fork()`, which adds the tasks to a thread pool so that one of the threads can call its `compute()` method -- Use another thread to compute the parallelization.

We subsequently call `right.compute()` (which is a normal method call computed in the current thread).

Join the results of the two threads by calling `left.join()`, which blocks until the computation of the recursive sum is completed and returned. We add the result from `left` and `right` together and return the sum.

There are other ways we can combine and order the execution of `fork()`, `compute()`, and `join()`. Some are better than others. We will explore more in the exercises.

`ForkJoinPool`

Let's now explore the idea behind how Java manages the thread pool with fork-join tasks. The details are beyond the scope of this module, but it would be interesting to note a few key points, as follows:

- Each thread has a queue of tasks.

- When a thread is idle, it checks its queue of tasks. If the queue is not empty, it picks up a task at the head of the queue to execute (e.g., invoke its `compute()` method). Otherwise, if the queue is empty, 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 that the most recently forked task gets executed next, similar to how normal recursive calls.
- When `join()` is called, several cases might happen.
  - If the subtask to be joined hasn't been executed, its `compute()` method is called and the subtask is executed.
  - 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.
  - If the subtask to be joined has been stolen and is being executed by another thread, then the current thread finds some other tasks to work on either in its local queue or steal another task from another queue.

The beauty of the mechanism here is that the threads always look for something to do and they cooperate to get as much work done as possible.

Order of `fork()` and `join()`

One implication of how `ForkJoinPool` adds and removes tasks from the queue is the order in which we call `fork()` and `join()`. Since the most recently forked task is likely to be executed next, we should `join()` the most recent `fork()` task first. In other words, the order of forking should be the reverse of the order of joining.

```
left.fork();
right.fork();
return right.join() + left.join();
```

```
(a) f1.fork(); return f2.compute() + f1.join(); // This is the most efficient.

(b) f1.fork(); return f1.join() + f2.compute(); // This works, but slow, since in Java
subexpressions are evaluated left to right, i.e. for A + B, A is evaluated first before B (by the
away this has nothing to do with associativity). So f1.join() needs to wait for f1.fork() to
complete before f2.compute() can be evaluated. Compare this with (a) where f2.compute() proceeds
while f1.fork() is running.

(c) return f1.compute() + f2.compute(); // This is sequentially recursive. Not much different from
(b), there is no overhead involved in forking and joining. Everything is done by the main thread.

(d) f1.fork(); f2.fork(); return f2.join() + f1.join(); // Apart from the first recursion, main
thread delegates all other work to worker threads in the common pool.

(e) f1.fork(); f2.fork(); return f1.join() + f2.join(); // Looks the same as (d), but (d) still
preferred as it follows the convention of joins to be returned innermost first. Since a thread
forks tasks to the front of its own double-ended queue, the last task forked should be the one
that is joined when the thread becomes idle; tasks at the back of the dequeue are stolen by other
idle worker threads.

(f) Other non-functional combinations
• return f1.join() + f2.join(); // works
• return f1.fork() + f2.fork();
• return f1.compute() + f2.fork();
• return f1.fork() + f2.join();

// A fork() must be followed by a join() to get the result back. None of the options that uses
fork also join back the result. The only option that gives us the correct result is A. Note that
it computes the Fibonacci number sequentially.
```