Unit 1: Program and Compiler

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

Runtime Errors. Errors that are detected while the program is being executed.

Unit 2: Variable and Type

Variable. An abstraction of data.

Type. An abstraction of functionality given to the data.

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

Static Type. Variable can only hold values of the same type of the variable. Type of a variable cannot be changed.

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

Primitive Types. Numerical and boolean values.

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

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

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

Unit 3: Functions

Functions as an Abstraction over Computation. Allow us to group a set of *instructions* and give it a name. Functions serve any purposes.

1. Allows programmers to compartmentalize computation and its effects.
2. Allows programmers to hide how a task is performed.
3. Reduce repetition in our code with code reuse.

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

Unit 4: Encapsulation

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

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

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

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

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

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

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

Unit 5: Information Hiding

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

Constructor. Method that initializes an object.

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

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

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

Unit 6: Tell, Don't Ask

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

Unit 7: Class Fields

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

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

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

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

Unit 8: Class Methods

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

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

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

Unit 9: Composition

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

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

Unit 10: Inheritance

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

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

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

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

Since ColoredCircle extends from Circle, ColoredCircle <: Circle.

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

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

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

Unit 11: Overriding

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

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

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

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

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

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

Method Descriptor. Method signature and the return type.

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

Unit 12: Polymorphism

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

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

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

Unit 13: Liskov Substitution Principle

Liskov Substitution Principle. "Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S<:T."

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

LSP cannot be enforced by the compiler.

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

Unit 14: Abstract Classes

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

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

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

Unit 15: Interface

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

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

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

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

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

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

Unit 16: Wrapper Class

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

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

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

Unit 17: Run-Time Class Mismatch

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

Unit 18: Variance

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

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

Let $C(S)$ be some complex type based on type S.
We say a complex type is

(a) *covariant* if $S <: T$ implies $C(S) <: C(T)$
(b) *contravariant* if $S <: T$ implies $C(T) <: C(S)$
(c) *invariant* if it is neither covariant nor contravariant

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

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

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

Unit 19: Exceptions

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

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

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

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

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

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

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

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

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

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

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

Unit 20: Generics

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

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

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

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

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

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

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

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

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

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

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

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

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

Unit 21: Type Erasure

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

Type Erasure. Each type parameter are replaced with object. If the type parameter is bounded, its is replaced by the bounds instead.

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

Unit 22: Unchecked Warnings

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

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

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

Unit 23: Wildcards

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

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

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

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

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

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

1. If you are writing a method that can be implemented using functionality provided in the `object` class
2. When the code is using methods in the generic class that don't depend on the type parameter.

Unit 24: Type Inference

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

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

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

Notes:

1. Check carefully whether a method is being overridden.
2. You cannot instantiate a interface.
3. After `exception` is thrown, everything else in try is ignores, and the `catch` and `finally` block are run.
4. LSP: You need to write what property of the superclass that is no longer holds for the subclass.
5. Generics allow classes / methods that use any reference type to be defined without resorting to using the `Object` type. It enforces type safety by binding the generic type to a specific given type argument at compile time. Attempt to pass in an incompatible type would lead to compilation error.
6. `Object` is upper bounded by `? extends Object`
7. The method `grade` in Assessments can be overridden by individual subclasses – polymorphism can be used here.
8. Existing code that has been written to invoke A's copy would still work if the code invoked B's copy instead after B inherits from A.
9. Do not forget the `this` when drawing the call stack.
10. Do not forget to create that arguments are allocated on stack too.