Beyond Classes

Interface

```
public abstract interface Run { //abstract implicit modifiers

  public abstract void run();  //public abstract implicit modifiers
}
```

Concise form:

```
public interface Run {
  void run();
}
```

An interface cannot be marked as final

```
//does not compile
public final interface Walk {}
```

Interface variables are implicitly **public, static, final.**

```
public interface Weight {
  int max_weight = 5;
  public static final int max_height = 50;
  String color;  //does not compile as it's not initialized
}
```

Interfaces with the same default name

Class which implements two interfaces having the same(signature) default method:

```
@Override
public void go() {
    Run.super.go();
    //or
    Walk.super.go();
}
```

Full Example

Case abstract class

Abstract class default methods

default conflict abstract method

```
interface House {
  default void work(){}  //default method
}

interface Office {
  void work(); //abstract method
}

class HomeOffice implements Office, House {
  //I am forced to implement work()!
  @Override
  public void work() {}
}
```

default vs abstract conflict

private methods

A private interface method **cannot** be called in a method outside the interface declaration.

```
interface InterfaceWithAPrivateMethod {
  private void message();
  void hello();
}

class InterfaceWithAPrivateMethodImpl implements InterfaceWithAPriva
  @Override
  public void hello(){
    //does not compile!
    //message();
  }
}
```

Override

`@Override` annotation informs the compiler that the element is meant to override an element declared in a superclass and/or interface. While **it is not required** to use this

annotation when overriding a method, it helps to prevent errors.

Default Methods

First, if a class or interface inherits two interfaces containing default methods with the same signature, it must override the method with its own implementation.

Inherit two default methods with the same signature

Methods accessible by default method

A default method can invoke any other type of method within the interface:

- public static methods
- private static methods
- public instance methods
- private static methods
- other default methods

DefaultMethodCallingOtherMethods

static methods

```
interface Certifications {
  //public by default
  public static void ocp17() {}
}

class MyCertifications implements Certifications {
  public static void main(String[] args) {
    Certifications.ocp17();  //valid
    MyCertifications myCertifications = new MyCertifications();
    //I cannot invoke a static method of the interface from an insta
    myCertifications.ocp17();  //DOES NOT COMPILE!
  }
}
```

interface with static methods

Sealed

A sealed class requires at least one subclass to extend.

Rules

- A sealed class needs to be declared in the same package or named module as their direct subclasses.
- Direct subclasses of a sealed classes must be either `final` or `sealed` or `non-sealed`.
- The permits class is optional if the classes (sealed and subclasses) are declared in the same file.

Same file

The `permits` clause is optional if the subclass is nested or declared in the same file.

```
//same file
public sealed class Snake {}
final class Cobra extends Snake {}
```

Sealed interfaces

An interface can be declared sealed. The `permits` list can apply to a class that implements the interface or an interface that extend the interface.
Then a sealed interface can be extended only by:

- `sealed` interfaces
- `non-sealed` interfaces.

```
// Sealed interface
public sealed interface Pet permits Cat, Dog, Rabbit {}

// Classes permitted to implement sealed interface Pet
public final class Cat implements Pet {}
public final class Dog implements Pet {}

// Interface permitted to extend sealed interface pet
public non-sealed interface Rabbit extends Pet {}
```

Example of sealed interface

Different files

```
public sealed class Snake permits Cobra, Viper {}
//separated files, but same pacakge
final class Cobra extends Snake {}
non-sealed class Viper extends Snake {}
```

```
//same file
//no permits needed
sealed class HumanBeing {}

sealed class Male extends HumanBeing {}
non-sealed class EuropeanMale extends HumanBeing {}
final class AsianMale extends HumanBeing {}
```

A subclass (Male) of a sealed class (HumanBeing) must be marked either `final` or `sealed` or `non-sealed`.

Modules

Named Module: which allow sealed classes and their direct subclasses in different packages, provided they are in the same named module.

sealed nested

```
  sealed class Pet {
    public final class Dog extends Pet {}
  }
```

sealed nested

Final

```
final class AFinal {
  public static void main(String[] args) {
    //I can't override a final class
    //new AFinal() {}; //does not compile
  }
}
```

Record

Compact Constructor

```
public record Person(String firstName, String lastName) {

  //this is a compact constructor
  public Person {  //no parenthesis!!!
    if (firstName.isEmpty() || lastName.isEmpty()) {
      throw new IllegalArgumentException("invalid");
    }
  }
}
```

A compact constructor cannot set an instance variable through `this` but just with the normal assignment operator.

```
public record Name(String name) {
  public Name {
    name = "Enrico"; //this works
    //this.name = "Enrico"; //this does not compile
  }
}
```

A compact constructor must have the same access modifiers as the record itself.

```
//does not compile
public record Name(String name) {
  Name {
    name="John";
  }
}
```

Overloaded constructor

```
public record Person(String firstName, String lastName) {

  //this is an overloaded constructor
  public Person() {
    //he first line must be a call to another constructor,
    this("Enrico", "Giurin");
  }
}
```

Record with instance variables

```
record Game() {
  //does not compile
   final int score = 10;
}
```

It does not compile because records cannot include instance variables not listed in the declaration of the record, as it could break immutability.

enum

Abstract method

If an enum contains an abstract method, then every enum value must include an override of this abstract method.

```
public enum Gender {
  MALE {
    @Override
    public String description() {
      return "male";
    }
  }, FEMALE {
    @Override
    public String description() {
      return "female";
    }
  };
  public abstract String description();
}
```

Constructors

enum constructors are implicitly private!

```
  public SeasonWithValues(String description) {  //DOES NOT COMPILE
    this.description = description;
  }
```

Access to static field not final

It is illegal to access (not final) static member from enum constructor or instance initializer.

```
enum Api {
  UPDATE("update the data"), READ("read the data");
  private String description;
  private static String version = "2.0";
  private static final String firstVersion = "1.0";

  Api(String description) {
    this.description = description;
    System.out.println(firstVersion);  //this is fine
    //It is illegal to access (not final) static member 'version'
    //( JLS 8.9.2 )
    //System.out.println(version);  //does not compile
  }
}
```

Enum With Fields

Comparable

all the enum implements `Comparable`.

```
int compare = CardinalPoints.NORTH.compareTo(CardinalPoints.EAS

//so they can be used in a TreeSet
TreeSet<CardinalPoints> set = new TreeSet<>();
set.add(CardinalPoints.SOUTH);
set.add(CardinalPoints.NORTH);
set.add(CardinalPoints.WEST);
System.out.println(set); //[NORTH, WEST, SOUTH]
```

Nested Classes

Inner Class

Reference: `Outer.this.field`.

```
class University {
  int id;
  class Department {
    int id;
    class Professor {
      int id;
      void printID() {
        System.out.println("University id: " + University.this.id);
        //System.out.println("University id: "+this.id); //equivale
        System.out.println("Department id: " + University.Departmen
        // System.out.println("Department id: "+Department.this.id)
        System.out.println("professor id: " + University.Department
        //System.out.println("professor id: "+Professor.this.id);//
      }
    }
  }
}
```

University

Nested Local Class

A local class can access only final and effectively final local variables.

```
  public void sortArray() {
    int size = 0;
    class NestedLocal {
      public void sort() {
        //size is not effectively fianl, as it's changed in the las
        int c = size;  //does not compile!
      }
    }
    //here I modify size, so I make it not effectively final
    size = 0;
  }
```

Nested Local Class