Definiciones Básicas	
Clase	Describe un tipo de objeto en particular, puede tener métodos y atributos
Método	Una función definida en una clase. Los métodos implementan las responsabilidades de los objetos
Objeto	Las unidades básicas de construcción en Java. Los objetos tienen variables (información) y métodos (funcionalidad)

Operadores Matemáticos	
Suma	+
Resta	-
División	/
Multiplicación	*
Módulo	% (Devuelve el resto de una división entera)
Sumar 1	++
Restar 1	--
Operadores lógicos:	
Y	&& (Devuelve verdadero si las dos evaluaciones son verdaderas)
O	|| (Devuelve verdadero si una de las dos evaluaciones son verdaderas)
No	! (Devuelve lo opuesto al resultado de la evaluación)
Comparaciones:	
Mayor	>
Menor	<
Igual	== o .equals()
Mayor o igual	>=
Menor o igual	<=
No igual	!=

Tipos de datos	
Integer	Números enteros
Long	Enteros muy grandes
Float	Número con coma
Double	Número con coma y muchos decimales
String	Cadena de caracteres
Character	Un caracter (Ej: 'a')
Boolean	Verdadero o Falso

Operaciones condicionales:

```
if (condición){
  //código que se corre si la condición es verdadera
} else if(condición){
  //código que se corre si la primera condición no fue verdadera y la segunda sí es verdadera
}else {
  // código que se corre si ninguna condición anterior fue verdadera
}
```

Loops:

For:

```
for(Integer i = 0; i < valorMaximo; i++){
    // código que se ejecuta cada vez
}
```

For Each:

```
for(Object object : listaDeObjetos){
        //código que se va a ejecutar por cada objeto en la lista
}
```

While:

```
while(condición){
        //hacer este código
}
```

Métodos:

Definición: un método se define a través de su firma.

```
visibilidad TipoQueRetorna
nombre(TipoDelParametro parámetro){
      //codigo del método
}
```

Ejemplo: método que toma dos Strings y devuelve un String:

```
public String unirStrings (String s1,
String s2){
   return string1 + string2;
}
```

Ejemplo: método que **no devuelve nada** pero imprime un String por pantalla:

```
public void imprimirString (String
unString){
   System.out.println(unString);
}
```

Getters y Setters:

Son métodos públicos que nos permiten modificar y observar a los atributos privados de una clase:

Ejemplo de Getter:

```
public Object getAtributo(){
   return atributo;
}
```

Ejemplo de Setter:

```
public void setAtributo(Object atributo){
   this.atributo = atributo;
}
```

Visibilidad:

public: cualquiera puede acceder

private: sólo se puede acceder desde la misma clase

protected: sólo se puede acceder desde la misma clase o desde las clases que hereden de ella o que estén en el mismo paquete.

Clase

Definición:

```
public class NombreDeClase{
   // Atributos
   private TipoAtributo NombreDelAtributo;
   // Métodos
   public void Metodo(TipoParametro
Parametro){

   }
}
```

Ejemplo: creo la clase Animal con un único atributo que sea peso y una única responsabilidad que sea comer comida

```
public class Animal{
   // Atributos
   private Integer peso;
   // Métodos
   public void comer(Comida unaComida){

   }
}
```

Interfaz

Definición:

```
public interface NombreDeInterface{
   public TipoQueRetorna
metodoDeLaInterface(TipoParametro parametro);
}
```

Ejemplo: creo la interface Saltador cuya única responsabilidad es saltar

```
public interface Saltador{
    // Métodos
    public void saltar();
}
```

Implementación de Herencia

Definición:

Se utiliza la palabra reservada **extends** para indicarle a una clase quién es su clase padre.

```
public class Clase extends ClasePadre{

}
```

Ejemplo:

```
public class Perro extends Animal{

}
```

Implementación de Interfaces

Definición:

Se utiliza la palabra reservada **implements** para indicarle a una clase que implementa una interfaz.

```
public class Clase implements UnaInterface{

}
```

Ejemplo:

```
public class Perro implements
Saltador{

}
```

Combinación Interfaz y Herencia

Ejemplo la Clase Perro extiende de animal e implementa la interfaz saltador

```
public class Perro extends Animal implements
Saltador{

}
```

Constructores

Definición:

Se utiliza un constructor para crear instancias válidas del objeto. Es decir, creamos un objeto con los atributos ya inicializados..

```
public class Clase{

private String atributoString;
private  Integer atributoInteger;

public Clase(Integer unInteger, String unString) {
   atributoInteger = unInteger;
   atributoString = unString;
}
```

Ejemplo:

```
public class Perro{

private String nombre;
private Integer peso;

public Perro(String unNombre, Integer unPeso) {
   nombre = unNombre;
   peso = unPeso;
}
```

Equals

Definición:

Se le hace override al método equals para poder comparar objetos y definir cuando son iguales

```
@Override
public boolean equals(Object unObjeto) {
   Clase objClase = (Clase) unObjeto;
   if
(objClase.atributo.equals(unObjeto.getAtributo())){
       return true;
   }
   else {
       return false
   }
}
```

Ejemplo:

```
@Override
public boolean equals(Object obj) {
   Perro unPerro = (Perro) obj;
   if (unPerro.getnombre() == nombre &&
unPerro.getpeso() == peso){
       return true;
   }
   else {
       return false
   }
}

//Este código dice que dos perros son iguales
si tienen el mismo nombre y el mismo peso
```

Listas (List) - ArrayList

Definición:

Una lista es una colección de datos del mismo tipo, en donde importa el orden y puede haber repetidos.

Crear lista vacía:

```
List<Tipo> lista = new ArrayList<>();
```

Obtener elemento:

```
lista.get(posicion);
```

Agregar elemento:

```
lista.add(unElemento);
```

Remover elemento:

```
perrosList.remove(unElemento);
```

Recorrer una lista con for each:

```
public void recorrerLista(List<Tipo> lista){

   for (Tipo unObjeto : lista){
       //Hacer algo con el objeto
   }
}
```

Remove dentro de un ciclo for para varios elementos (útil también a la hora de filtrar una lista):

Para buscar y remover elementos de una lista, la mejor opción es crear una nueva lista con los elementos que quiero quedarme. Si usamos remove dentro del for, arruinamos la lista.

```
public void removerPerrosConNombre
(List<Perro> perros, String nombre){
   List<Perro> perrosFiltrados = new ArrayList<>();
   for (Perro unPerro : perros){
       if (unPerro.getNombre() != nombre){
           perrosFiltrados.add(unPerro);
       }
   }
   perros = perrosFiltrados;
}
```

Conjuntos (Set) - HashSet

Definición:

Un conjunto es una colección de datos del mismo tipo, en donde no importa el orden y no puede haber repetidos.

Crear conjunto vacío:

```
Set<Tipo> conjunto = new HashSet<>();
```

Agregar elemento:

```
conjunto.add(unElemento);
```

Recorrer un conjunto con for each:

```
public void recorrerLista(Set<Tipo> conjunto){

   for (Tipo unObjeto : conjunto){
       //Hacer algo con el objeto
   }
}
```

Remove dentro de un ciclo for:

Para buscar y remover elementos de un conjunto, la mejor opción es crear un nuevo conjunto con los elementos que quiero quedarme. Si usamos remove dentro del for, arruinamos el conjunto. (Similar a Listas)

```
public Set<Perro> removerPerrosConNombre
(String nombre){
   Set<Perro> perros = new HashSet<>();
   for (Perro unPerro : perros){
       if (unPerro.getNombre() != nombre){
           perros.add(unPerro);
       }
   }
   return perros;
}
```

Remove dentro de un ciclo for para eliminar un único elemento:

Para buscar y remover un elemento de una lista, la mejor opción es guardarnos una referencia del objeto a eliminar y fuera del for removerlo. Si usamos remove dentro del for, arruinamos la lista.

```
public void removerAlumnoConPadron
(List<Alumno> alumnos, int padron){
   Alumno alumnoAEliminar;
   for (Alumno unAlumno : alumnos){
       if (unAlumno.getPadron() == padron){
           alumnoAEliminar = unAlumno;
           break;
       }
   }
   alumnos.remove(alumnoAEliminar);
```

Diccionarios (Map) - HashMap

Definición:

Un diccionario es una colección de datos en donde existe una clave y un valor. A través de las claves podemos acceder al valor.

Crear diccionario nuevo:

```
Map<Integer, String> diccionario  = new HashMap<>();
```

Agregar una clave y un valor:

```
diccionario.put(clave, valor)
```

Reemplazar un valor:

```
diccionario.replace(clave, valor)
```

Obtener valores usando la clave:

```
diccionario.get(clave)
```

Obtener conjunto de todas las claves:

```
diccionario.keySet()
```

Recorrer una diccionario con for each:

En el caso del diccionario, para poder recorrerlo, debemos pedir el conjunto de todas las claves y trabajar con las claves para pedir los valores.

```
public void recorrerDiccionario(Map<Tipo> dicc){

   for (Tipo unaClave : dicc.keySet()){
       Tipo unValor = dicc.get(unaClave);
   }
}
```

Ejemplo:

```
Map<Integer, String> diccionario = new
HashMap<>();
for (Integer clave : diccionario.keySet()){
   //Imprimir por pantalla el valor
   System.out.println(diccionario.get(clave));
}
```

To String

Definición:

Se le hace override al método toString para customizar la forma en la que el método se representa como String

```
@Override
public String toString() {
    return String.format("unAtributo" +
getAtributo());
    }
```

Ejemplo:

```
public String toString() {
    return String.format("DNI: " + getDNI()
+ " Nombre: " + getNombre());
    }


//Este código imprime algo como: "DNI:
34455990 Nombre: Felipe Catania"
```

Excepciones

Definición:

Se denomina excepciones a los errores en tiempo de ejecución. Es decir, a los errores que ocurren cuando se está ejecutando el programa.

Estructura:

```
try {
 //Aca va el codigo que quiero intentar
} catch (Exception e){
 //Aca va el código que maneja la excepción
} finally{
// El finally es opcional.
//Aca va el codigo que quiero que se ejecute en
caso de que falle o no falle.
}
```

Ejemplo:

```
try {
        List unaListaDeNumeros;
        unaListaDeNumeros.add(4);
}catch (Exception e){
 e.printStackTrace();
}
```