Java 8

- **Lambda Expressions**
  - Works with Funtional Interfaces
    -> Lambda Expressions provides a body to an otherwise abstract method. If lambda expression is assigned to an interface with more that one abstract methods, only one of the method will be implemented, leaving the others unimplemented.
  - Compiler treats Lambda expression as a function and hence doesn't create .class file

- **Functional Interface**
  - Permits exactly one abstract method inside them.
  - Also called **Single Abstract Method (SAM)** interfaces.
  - Can have methods of object class.
    eg :-

```
@FunctionalInterface
interface Test {
void func1();         // Abstract method
String toString();  // Also an abstract method but is actually overriding method of Object class

    default void func2() {
        System.out.println("func2()");
        }
}
```

  - A functional interface can extends another interface only when it does not have any abstract method.
  - Some predefined functional interfaces :-
    - **Function** -> *R apply(T t);* -> takes one input parameter and returns a value.
    - **Predicate** -> *boolean test(T t);* -> takes one argument and returns boolean value.
    - **Consumer** -> *void accept(T t);* -> accepts single argument and returns no result.
    - **Supplier** -> *T get();* -> doesn't accept any input and returns a single output.

*T is input type and R is return type.

- **Streams**
  - Present in java.util.stream.
  - Egs :-

```
class Product {
```

```
        String name;
        int price;
  }
```

- **Max ->** product.stream().max((x,y) -> x.price > y.price ? 1 : -1).get();
- **Min ->** product.stream().min((x,y) -> x.price > y.price ? 1 : -1).get();
- **Sum ->** product.stream().collect(Collectors.summingDouble(x->x.price));

```
                product.stream().map(x->x.price).reduce((sum,x) -> sum +x);
                product.stream().map(x->x.price).collect(Collectors.reducing((x,y)->x+y));
```