# 함수형 인터페이스 소개

# 함수형 인터페이스 소개

- 미리 정의되어 있는 함수형 인터페이스
  - Collection<E> 인터페이스에는 removeIf 메소드가 존재하며 "default Boolean removeIf(Predicate<? Super E> filter)" 로 정의
  - removeIf 메소드의 사용을 위해 "Predicate<? Super E> filter"가 무엇인지 알아야 함
  - Predicate는 아래와 같이 정의되어 있는 제네릭 인터페이스이자 함수형 인터페이스

```
@FunctionalInterface
public interface Predicate<T> {
    Boolean test(T t);
}
```

  - 자바에서는 표준으로 정의된 함수형 대표적인 함수형 인터페이스 4개와 그 안에 선언된 추상 메소드
    - Predicate<T> : Boolean test(T t)
    - Supplier<T> : T get()
    - Consumer<T> : void accept(T t)
    - Function<T, R> : R apply(T t)

# 함수형 인터페이스 소개

- Predicate<T>
  - Predicate<T> 인터페이스에 "boolean test(T t)" 추상 메소드가 존재하며, 전달된 인자를 대상으로 true, false를 판단할 때 사용

```
public class PredicateDemo {
    public static int sum(Predicate<Integer> p, List<Integer>
list) {
        int s = 0;
        for (int n : list) {
            if (p.test(n)) {
                s += n;
            }
        }
        return s;
    }

    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 5, 7, 9, 11, 12);
        int s = sum(n -> n % 2 == 0, list);
        System.out.println("짝수 합 : " + s);
        s = sum(n -> n % 2 != 0, list);
        System.out.println("홀수 합 : " + s);
    }
}
```

  - "public static int sum(Predicate<Integer> p, List<Integer> list) " 메소드 에서 Predicate가 어떤 인터페이스인지 알고 있다면 "boolean test(Integer t)" 메소드 정의에 해당하는 람다식을 작성해서 전달해야 한다는 것을 알 수 있음

# 함수형 인터페이스 소개

1. 아래 코드에서 주석으로 표시된 내용의 출력을 보이도록 show 메소드를 정의해 보자.

```
public class PredicateShow {
    public static <T> void show(Predicate<T> p, List<T> list) {
        // 채워 넣을 부분
    }

    public static void main(String[] args) {
        List<Integer> list1 = Arrays.asList(1, 3, 8, 10, 11);
        show(n -> n % 2 != 0, list1);  // 홀수만 출력
        List<Double> list2 = Arrays.asList(-1.2, 3.5, -2.4, 9.5);
        show(n -> n > 0.0, list2);  // 0.0 보다 큰 수 출력
    }
}
```

# 함수형 인터페이스 소개

- Predicate<T>를 구체화하고 다양화 한 인터페이스들
  - Predicate<T>에서 T를 기본 자료형으로 결정하여 정의한 인터페이스들이 존재하며, 이들은 함수형 인터페이스지만 제네릭 아님
    - IntPredicate : boolean test(int value)
    - LongPredicate : boolean test(long value)
    - DoublePredicate : boolean test(double value)
  - Predicate<T>와 달리 두개의 인자를 받아서 true, false를 결정할 수 있는 제네릭 인터페이스
    - BiPredicate<T, U> : boolean test(T t, U u)

```
public class IntPredicateDemo {
    public static int sum(IntPredicate ip, List<Integer> list) {
        int s = 0;
        for (int n : list) {
            if (ip.test(n)) {
                s += n;
            }
        }
        return s;
    }

    public static void main(String[] args) {
        List<Integer> list = Arrays.asList(1, 5, 7, 9, 11, 12);
        int s = sum(n -> n % 2 == 0, list);
        System.out.println("짝수 합 : " + s);
        s = sum(n -> n % 2 != 0, list);
        System.out.println("홀수 합 : " + s);
    }
}
```

# 함수형 인터페이스 소개

- Supplier<T>
  - Supplier<T> 인터페이스에 "T get()" 추상 메소드가 존재하며, 단순히 무엇인가 반환할 때 사용

```
public class SupplierDemo {
    public static List<Integer> makeIntList(Supplier<Integer>
s, int n) {
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(s.get());  // 난수 생성 후 추가
        }
        return list;
    }
    public static void main(String[] args) {
        Supplier<Integer> spr = () -> {
          Random rand = new Random();
          return rand.nextInt(50);
        };
        List<Integer> list = makeIntList(spr, 5);
        System.out.println(list);
        list = makeIntList(spr, 10);
        System.out.println(list);
    }
}
```

# 함수형 인터페이스 소개

- Supplier<T>를 구체화 한 인터페이스들
  - Supplier<T>에서 T를 기본 자료형으로 결정하여 정의한 인터페이스들 존재
    - IntSupplier : int getAsInt()
    - LongSupplier : long getAsLong()
    - DoublePredicate : double getAsDouble()
    - BooleanSupplier : boolean getAsBoolean()

```
public class IntSupplierDemo {
    public static List<Integer> makeIntList(IntSupplier is, int n)
{
        List<Integer> list = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            list.add(is.getAsInt());
        }
        return list;
    }
```

```
 public static void main(String[] args) {
        IntSupplier ispr = () -> {
            Random rand = new Random();
            return rand.nextInt(50);
        };
        List<Integer> list = makeIntList(ispr, 5);
        System.out.println(list);
        list = makeIntList(ispr, 10);
        System.out.println(list);
    }
}
```

# 함수형 인터페이스 소개

- Consumer<T>
  - Consumer<T> 인터페이스에 "void accept(T t)" 추상 메소드가 존재하며, 전달 인자를 소비하는 형태로 매개변수와 반환형이 선언되어 있고, 전달된 인자를 기반으로 반환 이외의 다른 결과를 실행할 때 사용

```
public class ConsumerDemo {
    public static void main(String[] args) {
        Consumer<String> c = s -> System.out.println(s);
        c.accept("Pineapple");
        c.accept("Strawberry");
    }
}
```

# 함수형 인터페이스 소개

- Consumer<T>를 구체화하고 다양화 한 인터페이스들
  - Consumer<T>에서 T를 기본 자료형으로 결정하여 정의한 인터페이스와 매개변수의 선언을 다양화 한 인터페이스 존재
    - IntConsumer : void accept(int value)
    - ObjIntConsumer<T> : void accept (T t, int value)
    - LongConsumer : void accept(long value)
    - ObjLongConsumer<T> : void accept (T t, long value)
    - DoubleConsumer : void accept(double value)
    - ObjDoubleConsumer<T> : void accept (T t, double value)
    - BiConsumer<T, U> : void accept(T t, U u)

```
public class ObjIntConsumerDemo {
    public static void main(String[] args) {
        ObjIntConsumer<String> c = (s, i) -> System.out.println(i + ". " + s);
        int n = 1;
        c.accept("Toy", n++);
        c.accept("Book", n++);
        c.accept("Candy", n);
    }
}
```

# 함수형 인터페이스 소개

- Function<T, R>
  - Function<T, R> 인터페이스에 "R apply(T t)" 추상 메소드가 존재하며, 전달 인자와 반환 값이 모두 존재할 때 사용
  - 전달 인자와 반환 값이 있는 가장 보편적인 형태로 프로그래머가 흔히 사용할 수 있는 인터페이스

```
public class FunctionDemo {
    public static void main(String[] args) {
        Function<String, Integer> f = s -> s.length();
        System.out.println(f.apply("Robot"));
        System.out.println(f.apply("Toy"));
    }
}
```

```
public class FunctionDemo2 {
    public static void main(String[] args) {
        Function<Double, Double> cti = d -> d * 0.393701;
        Function<Double, Double> itc = d -> d * 2.54;
        System.out.println("1 cm = " + cti.apply(1.0) + " inch"); // 센티미터를 인치로 계산
        System.out.println("1 inch = " + itc.apply(1.0) + " cm"); // 인치를 센티미터로 계산
    }
}
```

# *함수형 인터페이스 소개*

- Function<T, R>을 구체화하고 다양화 한 인터페이스들
  - Function<T, R>에서 T, R을 기본 자료형으로 결정하여 정의한 인터페이스 존재 (메소드 명 규칙 확인)
    - IntToDoubleFunction : double applyAsDouble(int value)
    - DoubleToIntFunction : in applyAsInt(double value)
  - Function<T, R>에서 T, R의 자료형이 같아야 한다면 아래의 인터페이스 활용 (메소드 명 규칙 확인)
    - IntUnaryOperator : int applyAsInt(int operand)
    - DoubleUnaryOperator : double applyAsDouble(double operand)
  - Function<T, R>에 위치한 추상 메소드의 매개변수 선언과 반환형을 다양화 한 인터페이스
    - BiFunction<T, U, R> : R apply(T t, U u)
    - IntFunction<R> : R apply(int value)
    - DoubleFunction<R> : R apply(double value)
    - ToIntFunction<T> : int applyAsInt(T value)
    - ToDoubleFunction<T> : double applyAsDouble(T value)
    - ToIntBiFunction<T, U> : int applyAsInt(T t, U u)
    - ToDoubleBiFunction<T, U> : double applyAsDouble(T t, U u)

# 함수형 인터페이스 소개

```
public class ToIntFunctionDemo {
    public static void main(String[] args) {
        ToIntFunction<String> f = s -> s.length();
        System.out.println(f.applyAsInt("Robot"));
        System.out.println(f.applyAsInt("Box"));
    }
}
```

- 앞서 작성하였던 FunctionDemo2 에서 사용한 인터페이스를 DoubleUnaryOperator로 수정

```
public class DoubleUnaryOperatorDemo {
    public static void main(String[] args) {
        DoubleUnaryOperator cti = d -> d * 0.393701;
        DoubleUnaryOperator itc = d -> d * 2.54;
        System.out.println("1 cm = " + cti.applyAsDouble(1.0) + " inch"); // 센티미터를 인치로 계산
        System.out.println("1 inch = " + itc.applyAsDouble(1.0) + " cm"); // 인치를 센티미터로 계산
    }
}
```

# *함수형 인터페이스 소개*

- removeIf 메소드를 사용해 보자
  - removeIf 메소드의 선언 : default boolean removeIf(Predicate<? super E> filter)
  - 매개 변수 선언이 "Predicate<? super E> filter"이므로 ArrayList<Integer>의 인스턴스를 생성하면, 그안에 존재하는 removeIf 메소드의 E는 Integer로 결정
  - 즉, public boolean removeIf(Predicate<? super Integer> filter) {...}로 결정되며 매개변수 선언에 <? super Integer>가 존재하므로, 아래 참조변수를 대상으로 람다식을 작성하여 메소드의 인자로 전달 가능
    - Predicate<Integer> f = ...
    - Predicate<Number> f = ...
    - Predicate<Object> f = ...
  - 다음은 removeIf 메소드의 기능에 대한 정의를 자바 문서에서 발췌
    - Removes all of the elements of this collection that satisfy the given predicate
    - 컬렉션 인스턴스에 저장된 인스턴스 중 Predicate<T> 인터페이스의 test 메소드의 인자로 전달했을때 true로 반환되는 인스턴스 모두 삭제

# 함수형 인터페이스 소개

```
public class RemoveIfDemo {
    public static void main(String[] args) {
        List<Integer> list1 = Arrays.asList(1, -2, 3, -4, 5);
        list1 = new ArrayList<>(list1);

        List<Double> list2 = Arrays.asList(-1.1, 2.2, 3.3, -4.4, 5.5);
        list2 = new ArrayList<>(list2);

        list1.removeIf(n -> n.doubleValue() < 0);
        list2.removeIf(n -> n.doubleValue() < 0.0);

        System.out.println(list1);
        System.out.println(list2);
    }
}
```