|  |  |
| --- | --- |
| **import** java.io.File;  **import** java.io.FileFilter;  **public** **class** INDEX {  **public** INDEX() {  }  **public** **static** **void** main(String[] args) {  File file = **new** File(".");  File[] files = file.listFiles(**new** FileFilter() {  **public** **boolean** accept(File pathname) {  **return** pathname.isHidden();  }  });    **files = file.listFiles(File::isHidden);**  **files = file.listFiles((File)-> !File.isHidden() );**  **files = file.listFiles((File a)-> a.isHidden() );**    **for** (File atfile : files) {  System.***out***.println(atfile.getName());  }    }  } | |
| files = file.listFiles(File::isHidden); | **메서드 레퍼런스로 넘겨준다** |
| files = file.listFiles((File)-> !File.isHidden() ); | **람다식으로 익명함수를 넘겨준다** |
| files = file.listFiles((File a)-> a.isHidden() ); | **람다식으로 변수명지정해서 익명함수에서 사용** |
|  |  |

메서드 레퍼런스, 람다식

Predicate 인터페이스를 이용하여 처리

|  |  |
| --- | --- |
| **import** java.io.File;  **import** java.util.ArrayList;  **import** java.util.function.Predicate;  **public** **class** INDEX {  **public** INDEX() {  }  **public** **static** **void** main(String[] args) {  File file = **new** File(".");  File[] files = **null**;  files = file.listFiles();    // **ArrayList<File>hiddenFiles = getHidden(files);**  **// ArrayList<File>hiddenFiles = getFilter(files,File::isHidden);**  **ArrayList<File>hiddenFiles = *getFilter*(files,(File a)->a.isHidden());**    **for** (File atfile : hiddenFiles) {  System.***out***.println(atfile.getName());  }    }    **public** **static** ArrayList<File> getFilter(File[] files, Predicate<File> p){  ArrayList<File> result = **new** ArrayList<File>();  **for** (File file : files) {  **if**(p.test(file))  result.add(file);  }  **return** result;  }    **public** **static** ArrayList<File> getHidden(File[] files){  ArrayList<File> result = **new** ArrayList<File>();  **for** (File file : files) {  **if**(file.isHidden())  result.add(file);  }  **return** result;  }  } | |
| **getHidden(files);** | 히든만 찾을수 있다. |
| **getFilter(files,File::isHidden);** | 중복성을 제거하여 메서드레퍼런스로 처리가능 |
| ***getFilter*(files,(File a)->a.isHidden());** | 중복성제거 람다식으로 처리가능 |
| @FunctionalInterface  **public** **interface** Predicate<T> {  **boolean** test(T t);  ...  } | |

스트림을 이용하여 간편하게 사용

|  |  |
| --- | --- |
| **import** java.util.ArrayList;  **import** java.util.List;  **public** **class** 스트림 {  **public** 스트림() {  }  **public** **static** **void** main(String[] args) {  List<Integer> list = **new** ArrayList<Integer>();  list.add(3);  list.add(4);  list.add(1);    Object[] a = list.stream().filter((Integer i)-> i>1).toArray();  **for** (Object object : a) {  System.***out***.println((Integer)object);  }  }  } | |
| .stream().filter((Integer i)-> i>1).toArray(); | 스트림생성->중계연산->중단연산 |

디폴트 매서드

자바8은 구현 클래스에서 구현하지 않아도 되는 메서드를 인터페이스가 포함할수 있는 기능도 제공한다 메서드 바디는 클래스 구현이 아니라 인터페이스 일부로 포함한다.

그래서 이를 디폴트 메서드 라고 부른다.

자바8에서는 인터페이스 규격명세에 default라는 새로운 키둬드를 지원한다.

동작파라미터화 코드 전달하기(아직 어떻게 실행할 것인지 결정하지 않은 코드 블록)

소비자 요구사항 항상 바뀐다. 효과적으로 대응할수 있게 처리할수있는 방법이다.

위에서 설명한것들이 다들 동작파라미터화 이다.

Comparator로 정렬하기

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Comparator;  **public** **class** 정렬 {  **public** 정렬() {  }  **public** **static** **void** main(String[] args) {  ArrayList<Integer> ilist = **new** ArrayList<Integer>();  ilist.add(4);  ilist.add(1);  ilist.add(3);  ilist.add(2);  **ilist.sort(new Comparator<Integer>() {**  **public int compare(Integer o1, Integer o2) {**  **return o1.compareTo(o2);**  **}**  **});**  **ilist.sort((Integer a, Integer b) -> a.compareTo(b));**  **for** (Integer integer : ilist) {  System.***out***.println(integer);  }  }  } |

Runnable로 코드 블록 실행하기.

|  |
| --- |
| **public** **class** Runnable블록화 {  **public** Runnable블록화() {  }  **public** **static** **void** main(String[] args) {  **Thread t = new Thread(()->**  **{**  **while(true)**  **System.*out*.println("aaaa");**  **}**  **);**  **t.start();**          **Runnable r = ()->{**  **while(true)**  **System.*out*.println("rrr");**  **};**  **new Thread(r).start();**  }  } |

1. 동작 파라미터화에서 메서드 내부적으로 다양한 동작을 수행할 수 있도록 코드를 메서드 인수로 전달한다.

2. 동작 파라미터화를 이용하면 변화하는 요구사항에 더 잘 대응할 수 있는 코드를 구현할 수 있으며 나중에 엔지니어링 비용을 줄일수 있다.

3. 코드 전달 기법을 이용하면 동작을 메서드의 인수로 전달할 수 있다 하지만 자바 8이전에는 코드를 지저분하게 구현해야 해다. 익명 클래스로도 어느 정도 코드를 깔끔하게 만들수 있지만 자바8에서는 인터페이스를 상속받아 여러 클래스를 구현해야 하는 수고를 없앨수 있는 방법을 제공한다

4. 자바API의 많은 메서드는 정렬, 쓰레드,GUI처리 등을 포함한 다양한 동작으로 파라미터화할 수 있다.

람다표현식

1. 익명 : 보통의 메서드와 달리 이름이 없으므로 익명이라 표현한다. 구현해야 할 코드에 대한 걱정거리가 줄어든다.
2. 함수 : 람다는 메서드처럼 특정 클래스에 종속되지 않으므로 함수라고 부른다. 하지만 메서드처럼 파라미터 리스트바디, 변환형식, 가능한 예외리스트를 포함한다.
3. 전달 : 람다 표현을 메서드 인수로 전달하거나 변수로 저장할 수 있다.
4. 간결성 : 익명 클래스처럼 많은 자질구레한 코드를 구현할 필요가 없다.

**함수형 인터페이스의 추상메서드 스그너처를 함수 디스크립터 라고한다.**

|  |  |
| --- | --- |
| 사용사례 | 람다예제 |
| 불린 표현식 | (List<String list)->list.isEmpty() |
| 객체생성 | ()->new Apple(10) |
| 객체에서 소비 | (Apple a) ->{  System.out.println(a.getWeight());  } |
| 객체에서 선택/추출 | (String s)->s.length() |
| 두값을 조합 | (int a, int b) ->a\*b |
| 두객체비교 | (Apple a, Apple b)-> a.compareTo(b)) |

람다전달

|  |
| --- |
| @FunctionalInterface  **interface** BuffreadReaderProcessor{  String process(BufferedReader b) **throws** IOException;  } |
| **public** **class** 람다전달 {  **public** 람다전달() {  }    **public** **static** String processFile(BuffreadReaderProcessor p) **throws** IOException{  **try**(BufferedReader br = **new** BufferedReader(**new** FileReader("vv.txt"))){  **return** p.process(br);  }  }    **public** **static** **void** main(String[] args) **throws** FileNotFoundException, IOException {  **String str = *processFile*((BufferedReader br)->br.readLine());**  System.***out***.println(str);  }  } |

함수형 인터페이스 사용

Predicate, Consumer, Function(이것도있는데 다음에)

|  |
| --- |
| **import** java.io.File;  **import** java.util.ArrayList;  **import java.util.function.Consumer;**  **import java.util.function.Predicate;**  //Funtion은 다음에  **public** **class** 함수형인터페이스\_사용 {  **public** 함수형인터페이스\_사용() {  }    **public** **static** **void** main(String[] args) {  File file = **new** File(".");  File[] files = **null**;  files = file.listFiles();  ArrayList<File>hiddenFiles = *getPredicateFilter*(files,(File a)->a.isHidden());      **System.*out*.println("---Predicate---");**  **for (File atfile : hiddenFiles) {**  **System.*out*.println(atfile.getName());**  **}**    **System.*out*.println("---Consumer---");**  ***getConsumerFilter*(files,(File a)->System.*out*.println(a.getName()));**    }      **public** **static** ArrayList<File> getPredicateFilter(File[] files, Predicate<File> p){  ArrayList<File> result = **new** ArrayList<File>();  **for** (File file : files) {  **if(p.test(file))**  **result.add(file);**  }  **return** result;  }    **public** **static** **void** getConsumerFilter(File[] files, Consumer<File> p){  **for** (File file : files) {  **p.accept(file);**  }  }      }  http://stackoverflow.com/questions/18198176/java-8-lambda-function-that-throws-exception |

String option = paramList.stream().map(Object::toString).collect(Collectors.joining(" "));

public Set<String> findLongTracks(List<Album> albums) {  
return albums.stream()  
.flatMap(album -> album.getTracks())  
.filter(track -> track.getLength() > 60)  
.map(track -> track.getName())  
.collect(toSet());  
}

public Set<String> findLongTracks(List<Album> albums) {  
Set<String> trackNames = new HashSet<>();  
albums.stream()  
.flatMap(album -> album.getTracks())  
.filter(track -> track.getLength() > 60)  
.map(track -> track.getName())  
.forEach(name -> trackNames.add(name));  
return trackNames;  
}

public Set<String> findLongTracks(List<Album> albums) {  
Set<String> trackNames = new HashSet<>();  
albums.stream()  
.forEach(album -> {  
album.getTracks()  
.forEach(track -> {  
if (track.getLength() > 60) {  
String name = track.getName();  
trackNames.add(name);  
}  
});  
});  
return trackNames;  
}

BinaryOperator<Long> addLongs = (x, y) -> x + y;

long x = 1;

long y = 2;

System.out.println( addLongs.apply(x, y) );

public static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator) {

Objects.requireNonNull(comparator);

return (a, b) -> comparator.compare(a, b) >= 0 ? a : b;

}

​

​

​

​

public interface Comparator<T> {

​

int compare(T o1, T o2);

}

BinaryOperator<Integer> add = (x, y) → x + y

BinaryOperator<Long> addLongs = (x, y) -> x + y;