

PROGRAMMING III

OOP. JAVA LANGUAGE

COURSE 5



PREVIOUS COURSE CONTENT

Generics

- Defining a generic
- Run-time behavior

Collections

- List
- Set
- Map

COUSE CONTENT

Collections

- Utilities classes
- Aggregate Operations

Generics

- Wild Cards
- Restrictions

Comparing objects

GENERICS

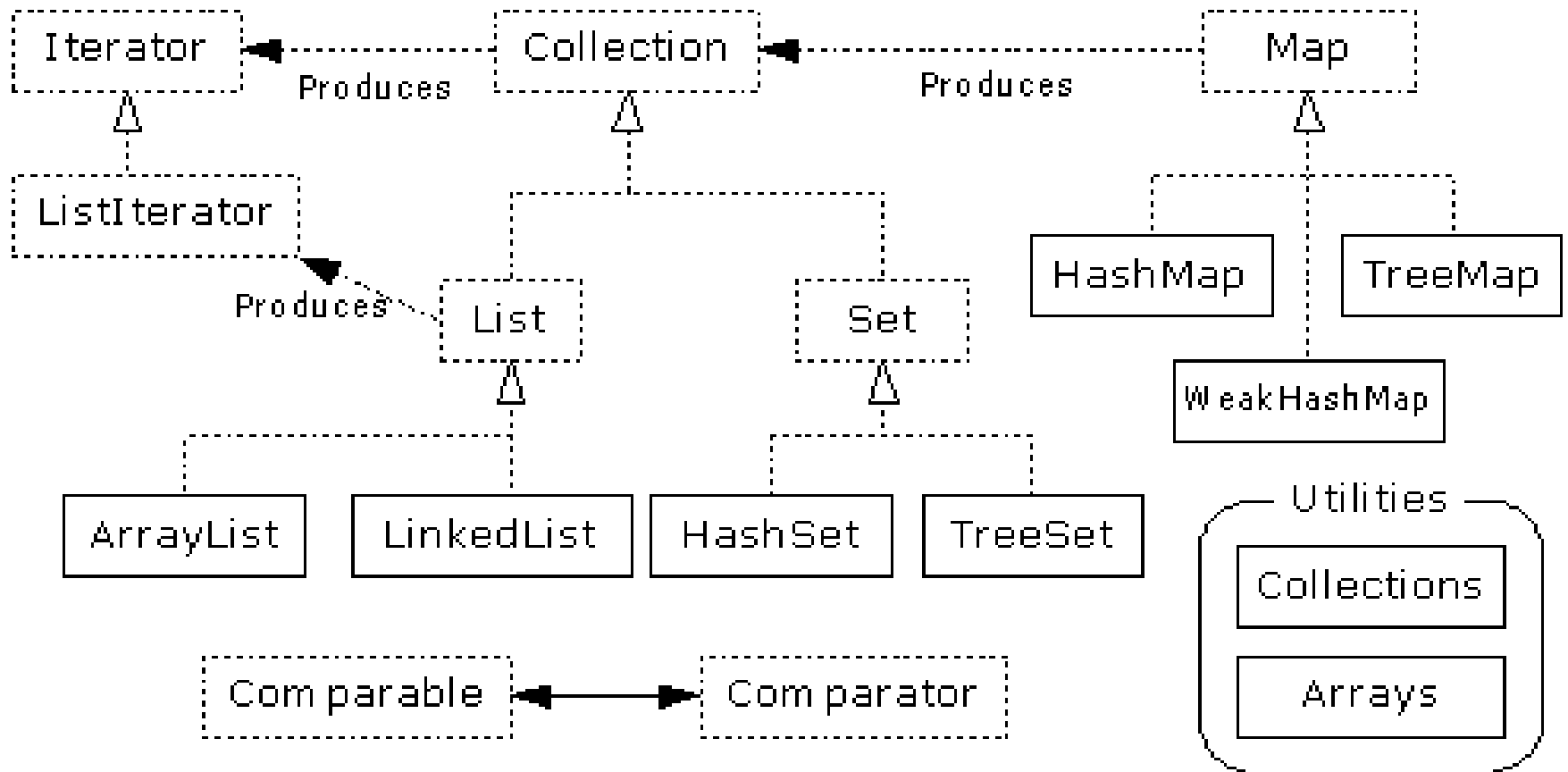
- ❑ Introduced in Java 1.5
- ❑ Allows class and methods definitions with parameters for types
 - ❑ Classes or methods that have type parameters are called *parameterized class* or *generic definitions*, or, simply, *generics*
- ❑ Can be
 - ❑ defined by Java libraries
 - ❑ user defined

COLLECTIONS

❑ What is a collection in Java?

- ❑ are containers of Objects which by polymorphism can hold any class that derives from Object
- ❑ GENERICS make containers aware of the type of objects they store
 - ❑ from Java 1.5

COLLECTIONS



COUSE CONTENT

Collections

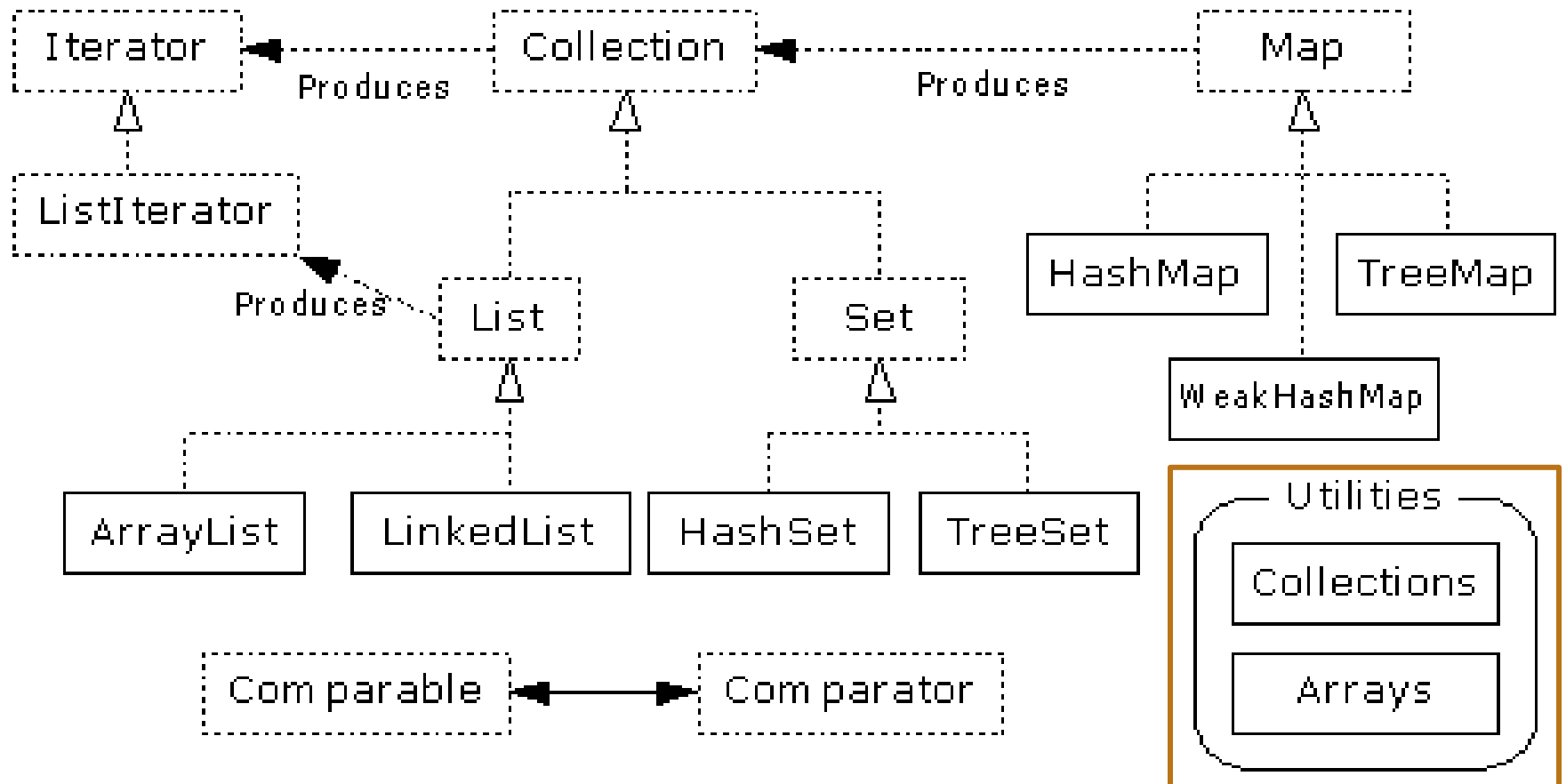
- Utilities classes
- Aggregate Operations

Generics

- Wild Cards
- Restrictions

Comparing objects

COLLECTIONS



COLLECTION. UTILITIES CLASS

❑ Algorithms

- ❑ These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
- ❑ The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

❑ **The Collections class provides a number of static methods for fundamental algorithms**

❑ **Most operate on Lists, some on all Collections**

- ❑ sort, search, shuffle
- ❑ reverse, fill, copy
- ❑ min, max

COLLECTIONS. OTHER CLASSES

- ❑ **Still available**
- ❑ **Don't use for new development**
- ❑ **Hashtable**
 - ❑ use HashMap
- ❑ **Enumeration**
 - ❑ use Collections and Iterators
- ❑ **Vector**
 - use ArrayList
- ❑ **Stack**
 - use LinkedList
- ❑ **BitSet**
 - use ArrayList of boolean, unless you can't stand the thought of the wasted space
- ❑ **Properties**

PROPERTIES CLASS

- ❑ **Located in java.util package**
- ❑ **Special case of Hashtable**
 - ❑ Keys and values are Strings
 - ❑ Tables can be saved to/loaded from file
- ❑ **Java VM maintains set of properties that define system environment**
 - ❑ Set when VM is initialized
 - ❑ Includes information about current user, VM version, Java environment, and OS configuration
 - ❑ Example:

```
Properties prop = System.getProperties();
Enumeration e = prop.propertyNames();
while (e.hasMoreElements()) {
    String key = (String) e.nextElement();
    System.out.println(key + " value is " + prop.getProperty(key));
}
```

COUSE CONTENT

Collections

- Utilities classes
- Aggregate Operations

Generics

- Wild Cards
- Restrictions

Comparing objects

COMPARABLE AND COMPARATORS

- ❑ **Some classes provide the ability to sort elements.**
 - ❑ How is this possible when the collection is supposed to be de-coupled from the data?

COMPARABLE AND COMPARATORS

- ❑ **Some classes provide the ability to sort elements.**
 - ❑ How is this possible when the collection is supposed to be decoupled from the data?
- ❑ **Java defines two ways of comparing objects**
 - ❑ The objects implement the Comparable interface
 - ❑ A Comparator object is used to compare the two objects
- ❑ **If the objects in question are Comparable, they are said to be sorted by their "natural" order.**
- ❑ **Comparable object can only offer one form of sorting. To provide multiple forms of sorting, Comparators must be used.**

COMPARABLE INTERFACE

- ❑ **The Comparable interface contains the *compareTo()* method.**
 - ❑ `int compareTo(T obj)`
- ❑ **This method returns**
 - ❑ 0 if the objects are equal
 - ❑ <0 if this object is less than the specified object
 - ❑ >0 if this object is greater than the specified object.
- ❑ **In order to provide a natural ordering for objects, you must implement the Comparable Interface**
- ❑ **Any object which is "Comparable" can be compared to another object of the same type.**
 - ❑ There is only one method defined within this interface.
 - ❑ Therefore, there is only one natural ordering of objects of a given type/class.

COMPARATOR INTERFACE

❑ **The Comparator interface defines two methods:**

❑ `int compare(T obj1, T obj2)`

❑ 0 if the Objects are equal

❑ <0 if the first object is less than the second object

❑ >0 if the first object is greater than the second object.

❑ `boolean equals(T obj)`

❑ returns true if the specified object is equal to this comparator.
ie. the specified object provides the same type of comparison
that this object does.

COMPARABLE AND COMPARATORS

- ❑ **Comparators are useful when objects must be sorted in different ways.**
- ❑ **For example**
 - ❑ Employees need to be sorted by first name, last name, start date, termination date and salary.
 - ❑ A Comparator could be provided for each case
 - ❑ The comparator interrogates the objects for the required values and returns the appropriate integer based on those values.
- ❑ **The appropriate Comparator is provided a parameter to the sorting algorithm.**

COUSE CONTENT

Collections

- Utilities classes
- Aggregate Operations

Generics

- Wild Cards

Comparing objects

GENERIC. WILDCARDS

❑ Bounded Type Parameters

- ❑ restrict the types that can be used as type arguments in a parameterized type

- ❑ `<T extends B1 [& B2 [& B3 ...]]>`

❑ Wildcards

- ❑ Wildcard - ?

- ❑ Represents an unknown type

- ❑ Can be used as the type of a

- ❑ Parameter

- ❑ Field

- ❑ Local variable

- ❑ Sometimes as a return type

GENERIC. WILDCARDS

❑ Upper Bounded Wildcards

❑ `public static void process(List<? extends Foo> list)`

❑ Unbounded Wildcards

❑ `public static void printList(List<?> list)`

❑ Lower Bounded Wildcards

❑ `public static void addNumbers(List<? super Integer> list)`

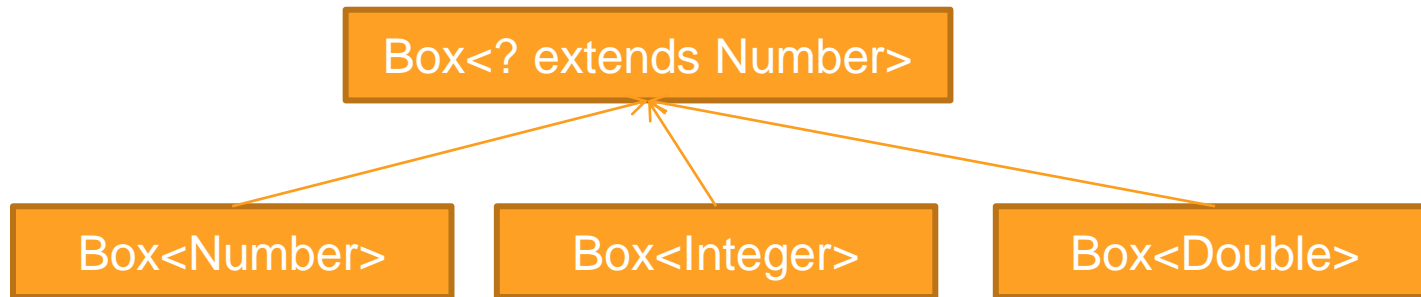
GENERICS. WILDCARDS. UPPER BOUNDED

❑ Upper Bounded

❑ defines a type that is bounded by the superclass

❑ Example

❑ Create a class box that can contain only objects that are subtypes of class number



❑ `Box<? extends Number> box = new Box<Integer>()`

GENERICS. WILDCARDS. UPPER BOUNDED

```
public class Box<E> {  
    public void copyFrom(Box<E> b) {  
        this.data = b.getData();  
    }  
}
```

```
Box<Integer> intBox = new Box<>();  
Box<Number> numBox = new Box<Number>();  
numBox.copyFrom(intBox);
```

Does the code execute?

GENERICS. WILDCARDS. UPPER BOUNDED

```
public class Box<E> {  
    public void copyFrom(Box<E> b) {  
        this.data = b.getData();  
    }  
}
```

```
public class Box<E> {  
    public void copyFrom(Box<E extends Number> b) {  
        this.data = b.getData();  
    }  
}
```

```
Box<Integer> intBox = new Box<>();  
Box<Number> numBox = new Box<Number>();  
numBox.copyFrom(intBox);
```

GENERICS. WILDCARDS. UNBOUNDED

□ Unbounded Wildcards

□ Print any list of objects

```
□ public static void printList(List<Object> list)
{
    for (Object obj: list)
        System.out.println(obj);
}
```

□ Call:

```
List<Object> listObject = new ArrayList<>();
printList(listObject);
```

```
List<String> listString = new ArrayList<>();
printList(listString); => compilation error
```

□ How to resolve?

GENERICS. WILDCARDS. UNBOUNDED

❑ Unbounded Wildcards

❑ Print any list of objects

```
public static void printList(List<?> list)
{
    for (Object obj: list)
        System.out.println(obj);
}
```

❑ Call:

```
List<Object> listObject = new ArrayList<>();
printList(listObject);
```

```
List<String> listString = new ArrayList<>();
printList(listString);
```

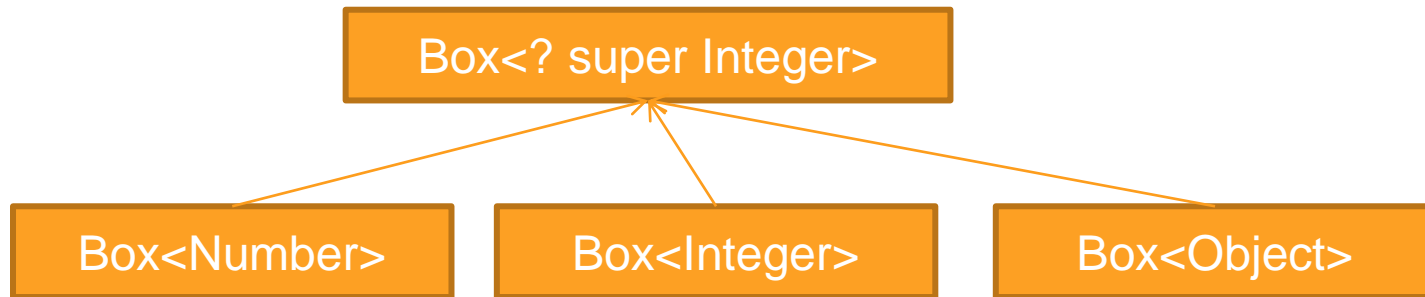
GENERICS. WILDCARDS. LOWER BOUNDED

❑ Lower Bounded

❑ defines a type that is bounded by the subclass

❑ Example

❑ Create a class box that can contain only objects that are subtypes of class number



❑ `Box<? super Integer> box = new Box<Number>()`

GENERICS. WILDCARDS. LOWER BOUNDED

❑ **Suppose we want to write `copyTo()` that copies data in the opposite direction of `copyFrom()`.**

`copyTo()` copies data from the host object to the given object.

```
public void copyTo(Box <E>b) {  
    b.data = this.getData();  
}
```

❑ **Above code is fine as long as `b` and the host are boxes of exactly same type. But `b` could be a box of an object that is a superclass of `E`.**

```
public void copyTo(Box<? Super E> b) {  
    b.data = this.getData  
}
```

```
Box <Integer> intBox = new Box<>();  
Box <Number> numBox = new Box<>();  
intBox.copyTo(numBox);
```

COUSE CONTENT

Collections

- Utilities classes
- Aggregate Operations

Generics

- Wild Cads
- Restrictions

Comparing objects

GENERIC. RESTRICTIONS

❑ Java, generic types are compile-time entities

- ❑ C++, instantiations of a class template are compiled separately as source code, and tailored code is produced for each one

- ❑ Primitive type parameters (List<int>) not allowed

 - ❑ in C++, both classes and primitive types allowed

 - ❑ Java – auto boxing

- ❑ Objects in JVM have non-generic classes

```
Pair<String> strPair = new Pair . . ;
```

```
Pair<Number> numPair = new Pair . . ;
```

```
b = strPair.getClass () == numPair.getClass ();
```

```
assert b == true; // both of the raw class Pair
```

 - ❑ but byte-code has reflective info about generics

GENERICS. RESTRICTIONS

❑ Instantiations of generic parameter T are not allowed

- ❑ `new T ()` // ERROR: whatever T to produce?
- ❑ `new T [10]`

❑ Arrays of parameterized types are not allowed

- ❑ `new Pair<String> [10];` // ERROR
- ❑ since type erasure removes type information needed for checks of array assignments

❑ Static fields and static methods with type parameters are not allowed

- ❑ `class Singleton {`
 - ❑ `private static T singleOne;` // ERROR
- ❑ since after type erasure, one class and one shared static field for all instantiations and their objects

❑ Cannot Create, Catch, or Throw Objects of Parameterized Types

GENERICCS

□ Why generic programming

- supports *statically-typed* data structures
 - *early detection* of type violations
 - cannot insert a string into `ArrayList <Number>`
 - also, hides automatically generated casts
- *superficially* resembles C++ templates
 - C++ templates are factories for ordinary classes and functions
 - a new class is always instantiated for given distinct generic parameters (type or other)
- in Java, generic types are factories for *compile-time* entities related to types and methods

COUSE CONTENT

Collections

- Utilities classes
- Aggregate Operations

Generics

- Wild Cards

Comparing objects

LAMBDA EXPRESSIONS

❑ A Java 8 lambda is basically a method in Java without a declaration usually written as (parameters) -> { body }.

❑ Examples

- ❑ `(int x, int y) -> { return x + y; }`
- ❑ `x -> x * x`
- ❑ `() -> x`

❑ A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context.

❑ Parenthesis are not needed around a single parameter.

❑ `()` is used to denote zero parameters.

❑ The body can contain zero or more statements.

❑ Braces are not needed around a single-statement body.

LAMBDA EXPRESSIONS

❑ Example of lambda usage for iterating through a list

❑ `List<Integer> intSeq = Arrays.asList(1, 2, 3);`

❑ `intSeq.forEach(z -> System.out.println(z));`

❑ `x -> System.out.println(x)` is a lambda expression that defines an anonymous function with one parameter named `x` of type `Integer`

❑ How could lambda be used to iterate through a map

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

❑ `items.put("A", 10);`

❑ `items.put("B", 20);`

LAMBDA EXPRESSIONS

❑ Example of lambda usage for iterating through a list

❑ `List<Integer> intSeq = Arrays.asList(1, 2, 3);`

❑ `intSeq.forEach(z -> System.out.println(z));`

❑ `intSeq.forEach(System.out::println);`

❑ `x -> System.out.println(x)` is a lambda expression that defines an anonymous function with one parameter named `x` of type `Integer`

❑ How could lambda be used to iterate through a map

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

❑ `items.put("A", 10);`

❑ `items.put("B", 20);`

❑ `items.forEach((k, v) -> System.out.println("key : " + k + " value : " + v));`

FUNCTIONAL INTERFACES

- ❑ **Interfaces with only one explicit abstract method**
 - ❑ AKA SAM interface (Single Abstract Method)
- ❑ **Optionally annotated with @FunctionalInterface**
 - ❑ Do it, for the same reason you use @Override
- ❑ **Some Functional Interfaces you know –**
 - ❑ `java.lang.Runnable`
 - ❑ `java.util.concurrent.Callable`
 - ❑ `java.util.Comparator`
 - ❑ `java.awt.event.ActionListener`
 - ❑ Many, many more in package `java.util.function`

METHOD REFERENCES

❑ An alternative to lambda

- ❑ An instance method of a particular object (bound)
 - ❑ `objectRef::methodName`
- ❑ An instance method, whose receiver is unspecified (unbound)
 - ❑ `ClassName::instanceMethodName` – The resulting function has an extra argument for the receiver
- ❑ A static method
 - ❑ `ClassName::staticMethodName`
- ❑ A constructor
 - ❑ `ClassName::new`

JAVA 8 STREAMS

What are streams?

JAVA 8 STREAMS

❑ What are streams?

- ❑ Streams are not related to InputStreams, OutputStreams, etc.
- ❑ Streams are NOT data structures but are wrappers around Collection that carry values from a source through a pipeline of operations.
- ❑ Stream represents a sequence of objects from a source, which supports aggregate operations

JAVA 8 STREAMS

❑ Streams characteristics

- ❑ Sequence of elements – A stream provides a set of elements of specific type in a sequential manner. A stream gets/computes elements on demand. It never stores the elements.
- ❑ Source – Stream takes Collections, Arrays, or I/O resources as input source.
- ❑ Aggregate operations – Stream supports aggregate operations like filter, map, limit, reduce, find, match, and so on.
- ❑ Pipelining – Most of the stream operations return stream itself so that their result can be pipelined. These operations are called intermediate operations and their function is to take input, process them, and return output to the target. `collect()` method is a terminal operation which is normally present at the end of the pipelining operation to mark the end of the stream.
- ❑ Automatic iterations – Stream operations do the iterations internally over the source elements provided, in contrast to Collections where explicit iteration is required.

STREAMS

□ Stream types

- `stream()` – Returns a sequential stream considering collection as its source.
- `parallelStream()` – Returns a parallel Stream considering collection as its source.

Example

```
List<String> strings = Arrays.asList("abc", "", "bc", "efg", "abcd", "",  
    "jkl");  
List<String> filtered = strings.stream().filter(string ->  
    !string.isEmpty()).collect(Collectors.toList());
```

CREATING STREAMS

❑ From individual values

- ❑ `Stream.of(val1, val2, ...)`

❑ From array

- ❑ `Stream.of(someArray)`

- ❑ `Arrays.stream(someArray)`

❑ From List (and other Collections)

- ❑ `someList.stream()`

- ❑ `someOtherCollection.stream()`

CREATING STREAMS

❑ *Stream.builder()*

```
Stream<String> streamBuilder  
= Stream.<String>builder().add("a").add("b").add("c").build();
```

❑ *Stream.generate()*

```
Stream<String> streamGenerated = Stream.generate(() ->  
"element").limit(10);
```

❑ *Stream.iterate()*

```
Stream<Integer> streamIterated = Stream.iterate(40, n -> n +  
2).limit(20);
```

CREATING STREAMS

□ Stream of Primitives

```
IntStream intStream = IntStream.range(1, 3);
```

```
LongStream longStream = LongStream.rangeClosed(1, 3);
```

```
Random random = new Random();
```

```
DoubleStream doubleStream = random.doubles(3);
```

□ Stream of *String*

```
IntStream streamOfChars = "abc".chars()
```

```
Stream<String> streamOfString =
```

```
Pattern.compile(", ").splitAsStream("a, b, c");
```

STREAM PIPELINE

- ❑ Perform a sequence of operations over the elements of the data source and aggregate their results

- ❑ Parts

- ❑ source,

- ❑ intermediate operation(s)

- ❑ return a new modified stream

- ❑ can be chained

- ❑ terminal operation.

- ❑ Only one terminal operation can be used per stream.

- ❑ The result of a interrogation

- ❑ Example

- ❑ Predefined operation: *count()*, *max()*, *min()*, *sum()*

STREAM PIPELINE

```
List<String>strings = Arrays.asList("abc", "", "bc", "efg",  
"abcd", "", "jkl");
```

```
//get count of empty string
```

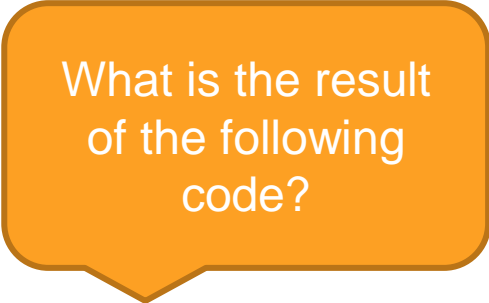
```
int count = strings.stream()  
    .filter(string -> string.isEmpty())  
    .count();
```

ORDER OF THE OPERATIONS

```
List<String> list = Arrays.asList("one", "two", "three", "four");
```

```
long size = list.stream().map(element -> {  
    System.out.println("Call map method");  
    return element.substring(0, 3);  
}).skip(2).count();  
System.out.println("size" + size);
```

```
size = list.stream().skip(2).map(element -> {  
    System.out.println("Call map method");  
    return element.substring(0, 3);  
}).count();  
System.out.println("size" + size);
```



What is the result of the following code?

ADVANCED OPERATIONS

❑ collect

- ❑ transform the elements of the stream into a different kind of result

❑ reduce.

- ❑ combines all elements of the stream into a single result

```
class Person {
    String name;
    int age;
    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return name;
    }
}

List<Person> persons =
    Arrays.asList( new Person("Max", 18),
                  new Person("Peter", 23),
                  new Person("Pamela", 23),
                  new Person("David", 12));
```


ADVANCED OPERATIONS.

COLLECT

```
List<Person> filtered = persons .stream()
    .filter(p -> p.name.startsWith("P"))
    .collect(Collectors.toList());
System.out.println(filtered);
```

```
Map<Integer, List<Person>> personsByAge = persons .stream()
    .collect(Collectors.groupingBy(p -> p.age));
personsByAge .forEach((age, p) -> System.out.format("age %s: %s\n", age, p));
```

```
Double averageAge = persons .stream()
    .collect(Collectors.averagingInt(p -> p.age));
System.out.println(averageAge);
```

```
IntSummaryStatistics ageSummary = persons .stream()
    .collect(Collectors.summarizingInt(p -> p.age));
System.out.println(ageSummary);
```

ADVANCED OPERATIONS. REDUCE

□/find the oldest person

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);
```

□determine the sum of ages from all persons

```
Integer ageSum = persons
    .stream()
    .reduce(0, (sum, p) -> sum += p.age, (sum1, sum2) -> sum1
    + sum2);
System.out.println(ageSum); // 76
```

EXAMPLE

Person result = persons.

```
.stream()  
.filter(x -> "michael".equals(x.getName()))  
.findAny()  
.orElse(null);
```

Person result = persons

```
.stream()  
.filter(x -> { if("michael".equals(x.getName()) &&  
21==x.getAge()){ return true; } return false; })  
.findAny()  
.orElse(null);
```