FACULTY OF INFORMATION TECHNOLOGY

DATA STRUCTURES (CTDL)

Semester 1, 2021/2022

Java Generics

https://docs.oracle.com/javase/tutorial/extra/generics/index.html

Java Collections

- Early versions of Java lacked generics…

```
interface Collection {
    /** Return true iff the collection contains obj */
    boolean contains(Object obj);
   /** Add obj to the collection; return true iff
      * the collection is changed. */
   boolean add(Object obj);
   /** Remove obj from the collection; return true iff
     * the collection is changed. */
   boolean remove(Object obj);
  ...
}
```

Java Collections (cont.)

- Lack of generics was painful because programmers had to manually cast.

```
Collection c = ...;
c.add("Hello");
c.add("World");
//...
for (Object ob : c) {
        String s = (String) ob;
        System.out.println( "s.length: "+    s.length());
}
```

- … and people often made mistakes!

Java Collections (cont.)

- Limitation seemed especially awkward because built-in arrays do not have the same problem!

```
String [] a = ...
a[0] = (“Hello”)
a[1] = (“World”);
//...
For (String s : a) {
        System.out.println(s);
}
```

- In late 1990s, Sun Microsystems initiated a design process to add generics to the language ...

Arrays → Generics

- One can think of the array "brackets" as a kind of parameterized type: a type-level function that takes one type as input and yields another type as output

```
Object []     a = ...
String []     b = ...
Integer []    c = ...
Button []     d = ...
```

- We should be able to do the same thing with object types generated by classes!

Generic Collections

- With generics, the Collection interface becomes…

```
Interface Collection<T> {
        /** Return true iff the collection contains x */
        boolean contains(T x);
        /** Add x          to the collection; return true iff
                *the collection is changed. */
        boolean add(T x);
        /** Remove x from the collection; return true iff
        *the collection is changed. */
        boolean remove(T x);
        ...
}
```

Generic Collections (cont.)

- With generics, no casts are needed...

```
Collection<String> c = ...;
c.add(“Hello”);
c.add(“World”);
//...
for (String s : c) {
        System.out.println( “s.length: ”+    s.length());
}
```

- … and mistakes (usually) get caught!

Generic Collections (cont.)

- Type checking as part of syntax check (compile time)
  - The compiler can automatically detect uses of collections with incorrect types…

```
Collection<String> c= ...
c.add("Hello")  /*      Okay   */
c.add(1979); /* Illegal: static error! */
```

Generally speaking,

Collection<String>

behaves like the parameterized type

Collection<T>

where all occurrences of T have been replaced byString.

Subtyping

- Subtyping extends naturally to generic types.

```
interface Collection<T> { ... }
interface List<T> extends Collection<T> { ...}
class LinkedList<T> implements List<T> { ...}
class ArrayList<T> implements List<T> { ... }

/* The following statements are all legal. */
List<String> l = new LinkedList<String>();
ArrayList<String> a = new ArrayList<String>();
Collection<String> c =  a;
l = a;
c = l;
```

Subtyping (cont.)

- String is a subtype of Object so...
- ...is LinkedList<String> a subtype of LinkedList<Object>?

```
LinkedList<String> ls = new LinkedList<String>();
LinkedList<Object> lo = new LinkedList<Object>();

lo = ls; // Suppose this is legal
lo.add(2110); //Type-checks: Integer subtype Object
String s = ls.get(0); // Type-checks: ls is a List<String>
```

- But what would happen at run-time if we were able to actually execute this code?

Array Subtyping

- Java's type system allows the analogous rule for arrays:

```
String[] as = new String[10];
Object[] ao = new Object[10];

ao = as; // Type-checks: considered outdated design
ao[0] = 2110; // Type-checks: Integer subtype Object
String s = as[0];//Type-checks: as is a String array
```

- What happens when this code is run? TRY IT OUT!
  - It throws an ArrayStoreException! Because arrays are built into Java right from beginning, it could be defined to detect such errors

ArrayStoreException

- An attempt has been made to store the wrong type of object into an array of objects

```
package lab1_arrays;

public class TestArrayStoreException  {
    public static void main(String[] args) {
        Object x[] = new String[7];
        x[0] = new Integer(0);
    }
}
```

A type parameter for a method

- Some comments to below method:

```
/** Replace all values x in list ts by y. */
public void replaceAll(List<Double> ts, Double x, Double
y) {
for (int i = 0; i < ts.size(); i = i + 1)
      if (Objects.equals(ts.get(i), x))
            ts.set(i, y);
}
```

- We would like to rewrite the parameter declarations so this method can be used for ANY list, no matter the type of its elements.

A type parameter for a method (cont.)

- Try replacing Double by some "Type parameter" T, and Java will still complain that type T is unknown.

```
/** Replace all values x in list ts by y. */
public void replaceAll(List<Double> ts, Double x, Double
y) {                                T          T         T
for (int i = 0; i < ts.size(); i = i + 1)
      if (Objects.equals(ts.get(i), x))
            ts.set(i, y);
}
```

- Somehow, Java must be told that T is a type parameter and not a real type.

A type parameter for a method (cont.)

- Placing <T> after the access modifier indicates that T is to be considered as a type parameter, to be replaced when method is called.

```
/** Replace all values x in list ts by y. */
public <T> void replaceAll(List<T> ts, T x, T y) {
      for (int i = 0; i < ts.size(); i = i + 1)
            if (Objects.equals(ts.get(i), x))
                  ts.set(i, y);
}
```

Printing Collections

- Suppose we want to write a method to print every value in a Collection<T>.

```
void print(Collection<Object> c) {
      for (Object x : c) {
            System.out.println(x);
      }
}
...
Collection<Integer> c = ...
c.add(42);
print(c);
```

- /*Illegal:Collection<Integer> is not a subtype of Collection<Object>! */

Wildcards: introduce wildcards

- To get around this problem, Java's designers added wildcards to the language

```
void print(Collection<?> c) {
      for (Object x : c) {
            System.out.println(x);
      }
}
...
Collection<Integer> c = ...
c.add(42);
print(c); /* Legal! */
```

- One can think of Collection<?> as a "Collection of some unknown type of values".

Wildcards

- We can't add values to collections whose types are wildcards ...

```
void doIt(Collection<?> c) {
c.add(42); /* Illegal! */
}
...
Collection<String> c = ...

doIt(c); /* Legal! */
```

42 can be added to
- Collection<Integer>
- Collection<Number>
- Collection<Object>

but c could be a Collection of anything, not just supertypes of Integer

Bounded Wildcards

- Sometimes it is useful to have some information about a wildcard. Can do this by adding bounds…

```
void doIt(Collection<? super Integer> c) {
c.add(42); /* Legal! */
}
// ...
Collection<Object> c1 = ...
doIt(c1); /* Legal! */
Collection<Float> c2 = ...
doIt(c2); /* Illegal! */
```

Now c can only be a Collection of some supertype of Integer, and 42 can be added to any such Collection

- "? super" is useful when you are only giving values to the object, such as putting values into a Collection

Bounded Wildcards (cont.)

- "? extends" is useful for when you are only receiving values from the object, such as getting values out of a Collection.

```
void doIt(Collection<? extends Shape> c) {
      for (Shape s : c)
            s.draw();
      }
// ...
Collection<Circle> c = ...
doIt(c); /* Legal! */
Collection<Object> c = ...
doIt(c); /* Illegal! */
```

Bounded Wildcards (cont.)

- Wildcards can be nested. The following receives Collections from an Iterable and then gives floats to those Collections.

```
void doIt(Iterable<? extends Collection<? super Float>>
cs) {
for (Collection<? super Float> c : cs)
c.add(0.0f);
}
// ...
List<Set<Float>> l = ...
doIt(l); /* Legal! */
Collection<List<Number>> c = ...
doIt(c); /* Legal! */
Iterable<Iterable<Float>> i = ...;
doIt(i); /* Illegal! */
ArrayList<?extends Set<?super Number>> a = ...
doIt(a); /* Legal! */
```

Generic Methods

- Here's the printing example again. Written with a method type-parameter.

```
<T> void print(Collection<T> c){//T is a type
parameter
      for(T x : c){
            System.out.println(x);
      }
}
...
Collection<Integer> c=...
c.add(42);
print(c);/*Moreexplicitly:this.<Integer>print(c)*/
```

- But wildcards are preferred when just as expressive.

Bounded vs unbounded wildcards in Generics

- Generics in Java is one of important feature added in Java 5 along with Enum, autoboxing and varargs, to provide compile time type-safety.

- bounded wildcards:

  <? extends T>: **all Types must be sub-class of T .** T represents the lower bound

  <? super T>: **all Types required to be the super class of T.** T represents the upper bound.

- unbounded wildcard:

  <?>: **any type**, similar to Object in Java

Interface Comparable

- Interface Comparable<T> declares a method for comparing one object to another.

```
interface Comparable<T>{
/*Return a negative number,0,or positive
number
*depending on whether this is less than,
*equal to,or greater than that*/
int compareTo(Tthat);
}
```

Integer, Double, Character, and String are all Comparable with themselves

Interface Comparable (cont.)

- Type parameter: anything T that implements Comparable<T>

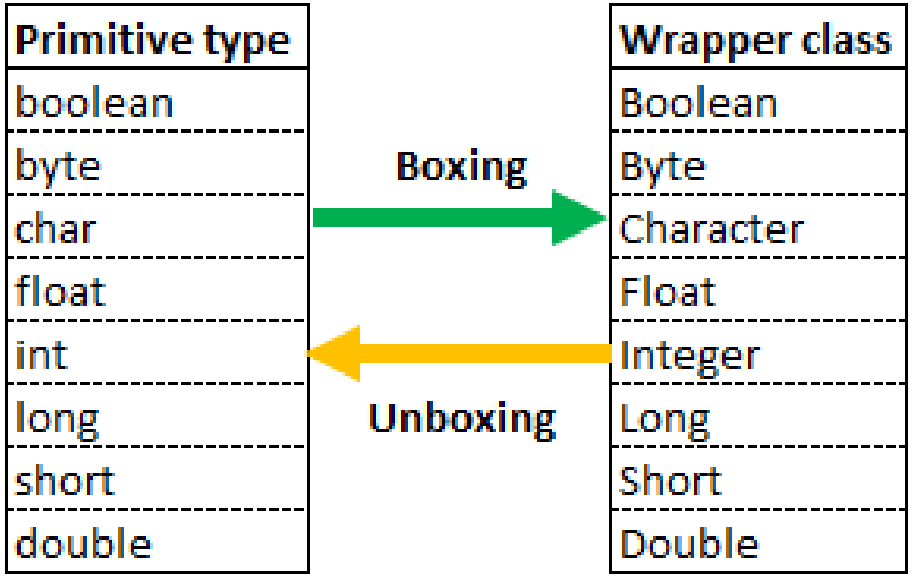
```
public static <T extends Comparable<T>>
int indexOf1(List<T> c, T x) {
}
```

```
// Remove every object, obj, from coll for which p.test(obj)
// is true. (This does the same thing as coll.removeIf(p).)
public static <T> void remove(Collection<T> coll, Predicate<T> p) {
    // TODO
    Iterator<T> iter = coll.iterator();
    while(iter.hasNext()) {
        T next = iter.next();
        if(p.test(next))
            iter.remove();
    }
}
```

Autoboxing

- *Autoboxing* is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.

- For example, converting an int to an Integer, a double to a Double,..



Autoboxing Example

- Consider the following code:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(i);
```

- The compiler converts the previous code to the following at runtime:

```
List<Integer> li = new ArrayList<>();
for (int i = 1; i < 50; i += 2)
    li.add(Integer.valueOf(i));
```

When autoboxing is applied?

- The Java compiler applies autoboxing when a primitive value is:
  - Passed as a parameter to a method that expects an object of the corresponding wrapper class.
  - Assigned to a variable of the corresponding wrapper class.

Unboxing

- Converting an object of a wrapper type (Integer) to its corresponding primitive (int) value.

- The Java compiler applies unboxing when an object of a wrapper class is:
  - Passed as a parameter to a method that expects a value of the corresponding primitive type.
  - Assigned to a variable of the corresponding primitive type.

Unboxing Example

```
import java.util.ArrayList;
import java.util.List;

public class Unboxing {

    public static void main(String[] args) {
        Integer i = new Integer(-8);

        // 1. Unboxing through method invocation
        int absVal = absoluteValue(i);
        System.out.println("absolute value of " + i + " = " + absVal);

        List<Double> ld = new ArrayList<>();
        ld.add(3.1416);    // Π is autoboxed through method invocation.

        // 2. Unboxing through assignment
        double pi = ld.get(0);
        System.out.println("pi = " + pi);
    }

    public static int absoluteValue(int i) {
        return (i < 0) ? -i : i;
    }
}
```

Advantages of Autoboxing / Unboxing

- Autoboxing and unboxing lets developers write cleaner code, making it easier to read.
- Support using primitive types and Wrapper class objects interchangeably;
- Do not need to perform any typecasting explicitly.

Varargs

- A feature that simplifies the creation of methods that need to take a variable number of arguments.
- This feature is called varargs and it is short-form for variable-length arguments.
- (Prior to JDK 5), variable-length arguments could be handled two ways:
  - Using overloaded method(one for each);
  - Putting the arguments into an array.

Varargs (cont.)

- Syntax of varargs:

```
public static void fun(int ... a)
{
    // method body
}
```

- We don't have to provide overloaded methods so less code.

Varargs (cont.)

- Example:

```
// A method that takes variable number of integer
// arguments.
static void fun(int... a) {
    System.out.println("Number of arguments: " + a.length);

    // using for each loop to display contents of a
    for (int i : a)
        System.out.print(i + " ");
    System.out.println();
}
// Driver code
public static void main(String args[]) {
    // Calling the varargs method with different number
    // of parameters
    fun(100); // one parameter
    fun(1, 2, 3, 4); // four parameters
    fun(); // no parameter
}
```

Varargs quizz

Given:

```
1. class Voop {
2.   public static void main(String [] args) {
3.     doStuff(1);
4.     doStuff(1, 2);
5.   }
6.   // insert code here
7. }
```

Which, inserted independently at line 6, will compile? (Choose all that apply.)

A. `static void doStuff(int... doArgs) { }`

B. `static void doStuff (int [] doArgs) { }`

C. `static void doStuff(int doArgs...) { }`

D. `static void doStuff(int... doArgs, int y) { }`

E. `static void doStuff(int x, int... doArgs) { }`

Rules

- *Varargs* are straightforward to use. But there're a few rules we have to keep in mind:
  - Each method can only have one *varargs* parameter
  - The *varargs* argument must be the last parameter

Enum

- Java enums

Enum

- Representing a group of named constants in a programming language.
- Declaration of enum in java:
  - Enum declaration can be done outside a Class or inside a Class but not inside a Method.
  - It is recommended that we name constant with all capital letters

```
enum Color {
    RED, GREEN, BLUE;
}

// Driver method
public static void main(String[] args) {
    Color c1 = Color.RED;
    System.out.println(c1);
}
```

Enum (cont.)

- Every enum internally implemented by using Class.

```
/* internally above enum Color is converted to
class Color
{
     public static final Color RED = new Color();
     public static final Color BLUE = new Color();
     public static final Color GREEN = new Color();
}*/
```

Enum (cont.)

- Every enum constant represents an **object** of type enum.

- enum type can be passed as an argument to **switch** statement.

Enum (cont.)

- Enum in switch … case

```
// An Enum class
enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY;
}
public class Test {
    // Driver class that contains an object of "day" and main().
    Day day;
    // Constructor
    public Test(Day day) {
        this.day = day;
    }
    // Prints a line about Day using switch
    public void dayIsLike() {
        switch (day) {
        case MONDAY:
            System.out.println("Mondays are bad.");
            break;
        case FRIDAY:
            System.out.println("Fridays are better.");
            break;
        case SATURDAY:
        case SUNDAY:
            System.out.println("Weekends are best.");
            break;
        default:
            System.out.println("Midweek days are so-so.");
            break;
        }
    }
```

Enum (cont.)

- Every enum constant is always implicitly **public static final**.

  ➔ access it by using enum Name because of **static**.

  ➔ can't create child enums because of **final**.

- We can declare **main() method** inside enum.

```
enum Color {
    RED, GREEN, BLUE;

    // Driver method
    public static void main(String[] args) {
        Color c1 = Color.RED;
        System.out.println(c1);
    }

}
```

Enum and Inheritance

- All enums implicitly extend **java.lang.Enum class**.

  ➔an enum cannot extend anything else.

- **toString() method** is overridden in **java.lang.Enum class**,which returns enum constant name.

- enum can implement many interfaces.

Enum methods

- **values() method** can be used to return all values present inside enum.
- Order is important in enums. By using **ordinal() method**, each enum constant index can be found, just like array index.
- **valueOf() method** returns the enum constant of the specified string value, if exists.
- These methods are present inside **java.lang.Enum**.

Enum example

```
enum Color {
    RED, GREEN, BLUE;
}

public static void main(String[] args) {
    // Calling values()
    Color arr[] = Color.values();

    // enum with loop
    for (Color col : arr) {
        // Calling ordinal() to find index
        // of color.
        System.out.println(col + " at index " + col.ordinal());
    }

    // Using valueOf(). Returns an object of
    // Color with given constant.
    // Uncommenting second line causes exception
    // IllegalArgumentException
    System.out.println(Color.valueOf("RED"));
    // System.out.println(Color.valueOf("WHITE"));
}
```

Other problems with enum

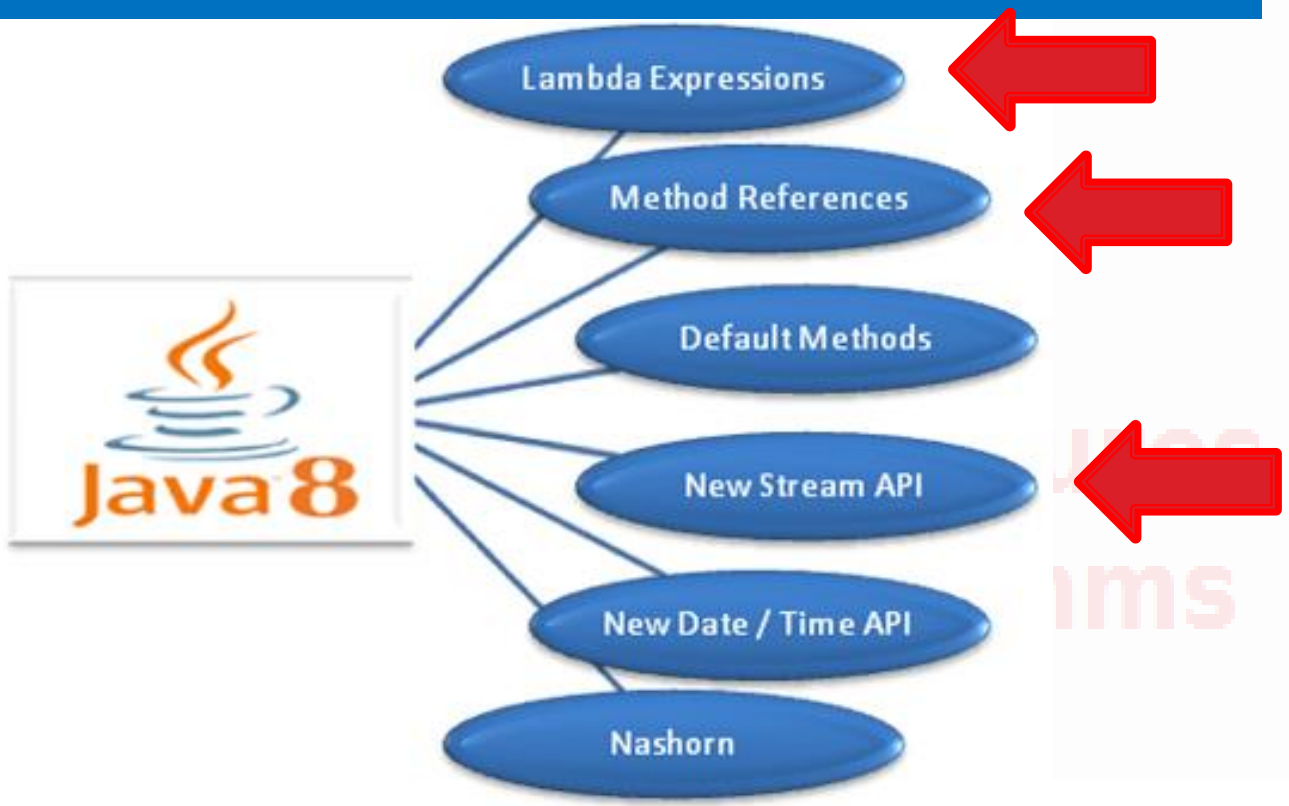
- **enum and constructor**:
  - enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
  - We can't create enum objects explicitly and hence we can't invoke enum constructor directly.

- **enum and methods**:
  - enum can contain **concrete** methods only i.e. no any **abstract** method.

Example

```
enum Color {
    RED, GREEN, BLUE;

    // enum constructor called separately for each
    // constant
    private Color() {
        System.out.println("Constructor called for : " + this.toString());
    }

    // Only concrete (not abstract) methods allowed
    public void colorInfo() {
        System.out.println("Universal Color");
    }
}

public class Test {
    // Driver method
    public static void main(String[] args) {
        Color c1 = Color.RED;
        System.out.println(c1);
        c1.colorInfo();
    }
}
```

News in Java 8



Functional style of programming

What is functional style of programming?

- Functional programming is a programming paradigm where programs are constructed by applying and composing functions.
- The functional style of programming was introduced in Java 8.
- The basic concepts are:
  - Functional interfaces
  - Method reference expressions
  - Lambda expressions
  - Streams
  - Collectors (not Collections)

Spirit of functional programming

- Functional programming could be considered as pure functions:
  - input-output only
  - the same input always generates the same output
  - no side-effects
    - a function relies on, or modifies, something outside its parameters to do something
  - no *explicit* changing of state

Java Functional Interfaces

What is functional interface?

- An Interface that contains exactly one abstract method.
  - It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

- Known as Single Abstract Method Interfaces (SAM Interfaces).
  - It is a new feature in Java, which helps to achieve functional programming approach.

Functional interface Example

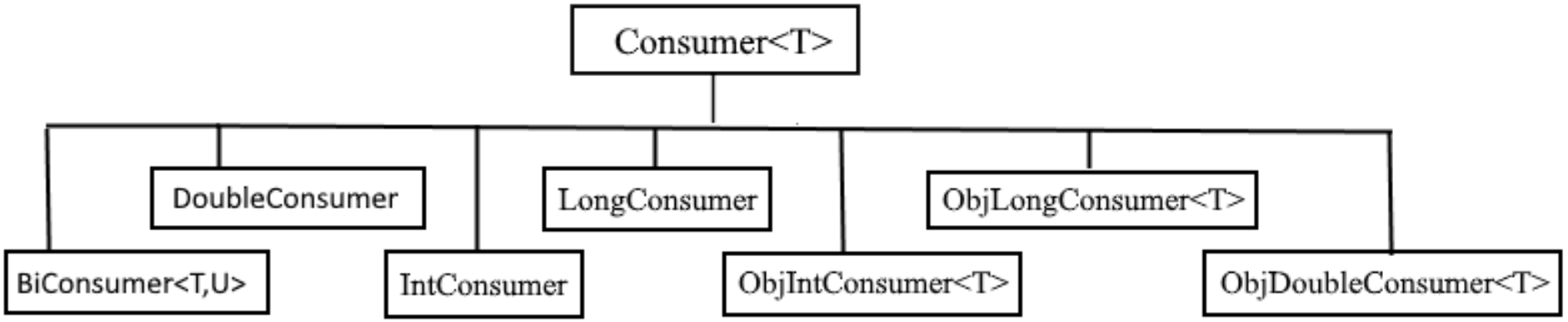
- Functional interface Sayable with single method named say(String msg)

```
@FunctionalInterface
interface sayable {
    void say(String msg);
}

public class FunctionalInterfaceExample implements sayable {
    public void say(String msg) {
        System.out.println(msg);
    }

    public static void main(String[] args) {
        FunctionalInterfaceExample fie = new FunctionalInterfaceExample();
        fie.say("Hello there");
    }
}
```

Consumer Functional Interface

- Consumer<T> is an inbuilt functional interface introduced in java 8 (java.util.Function)
- Can be used with a lambda expression and method reference

```
01. @FunctionalInterface
02. public interface Consumer<T> {
03.    void accept(T t);
04. }
```



Consumer Functional Interface (cont.)

- Consumer instance called multiplier of Integer type.
  - Multiplier operates on an Integer parameter.
- The accept method simply multiplies the input number by itself and prints the result

```
public class ConsumerDemo {
    public static void main(String[] args) {
        Consumer<Integer> multiplier = num -> System.out.println(num * num);
        multiplier.accept(10);
        multiplier.accept(4);
    }
}
```

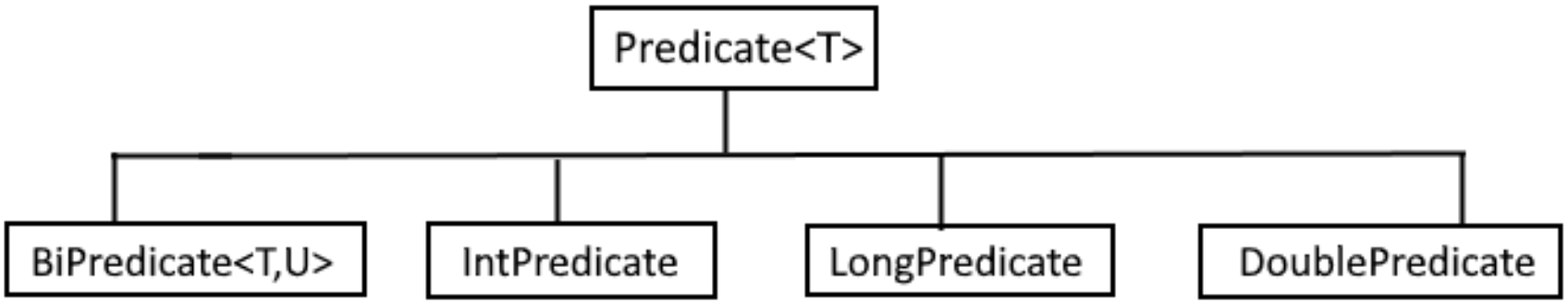
Output

```
100
16
```

Predicate Functional Interface (cont.)

- Predicate<T> is an inbuilt functional interface introduced in java 8 (java.util.Function)
- Can be used with a lambda expression and method reference

```
01. @FunctionalInterface
02. public interface Predicate < T > {
03.     boolean test(T t);
04. }
```



Predicate Functional Interface (cont.)

- Predicate instance called stringChecker of String type.
  - stringChecker accepts an argument of type String.
- The test method invokes the isEmpty method on the input String and a boolean value accordingly.

```
public class PredicateDemo {
    public static void main(String[] args) {
        Predicate<String> stringChecker = str -> str.isEmpty();
        String s = "Hello";
        boolean result = stringChecker.test(s);
        System.out.println(s + " is empty:" + result);
    }
}
```

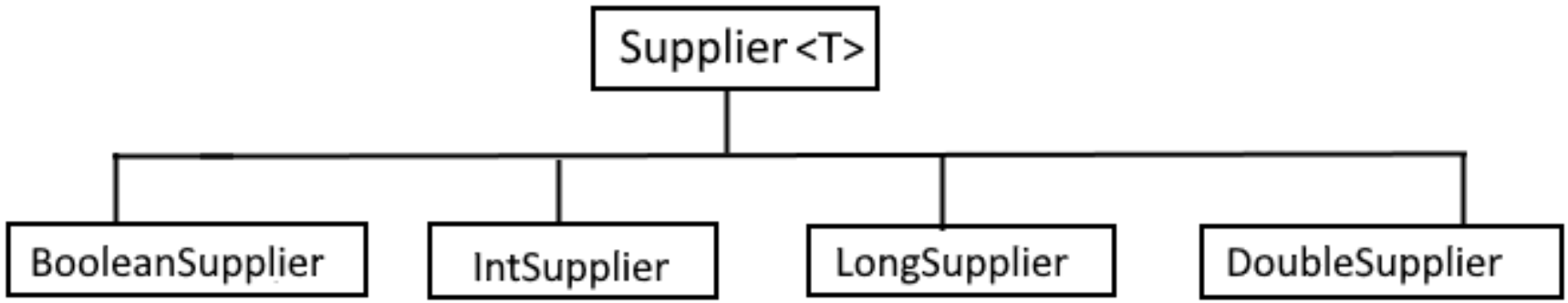
Output

Hello is empty:false

Supplier Functional Interface (cont.)

- Supplier<T> is an inbuilt functional interface introduced in java 8 (java.util.Function)
- Can be used with lambda expression and method reference

```
01. @FunctionalInterface
02. public interface Supplier < T > {
03.     T get();
04. }
```



Supplier Functional Interface (cont.)

- Supplier instance called randomNumberSupplier of Double type.
  - randomNumberSupplier returns a result of type Double
- The get method simply returns a new Random Double number.

```
public class SupplierDemo {
    public static void main(String[] args) {
        Supplier<Double> randomNumberSupplier = () -> new Random(10).nextDouble();
        System.out.println(randomNumberSupplier.get());
        System.out.println(randomNumberSupplier.get());
    }
}
```

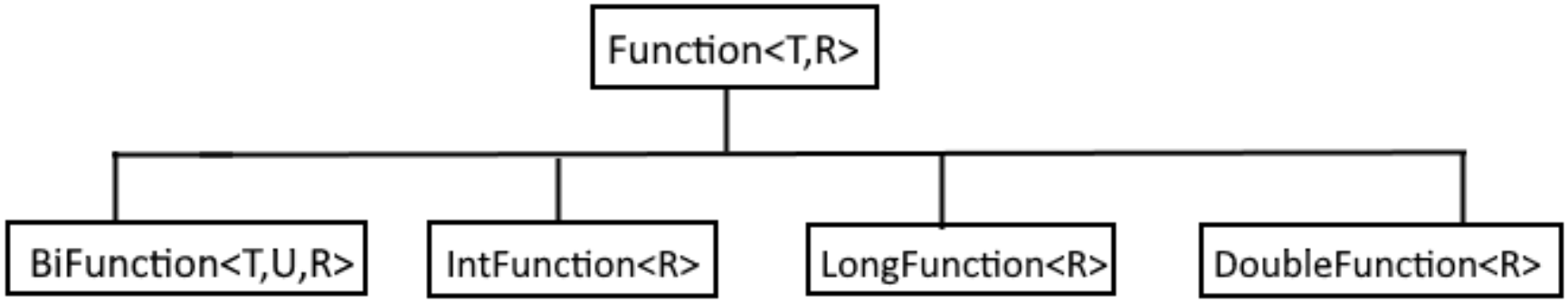
Output

0.7304302967434272
0.7304302967434272

Function Functional Interfaces

- Function<T,R> is an inbuilt functional interface introduced in java 8 (java.util.Function)
- Can be used with a lambda expression and method reference

```
01. @FunctionalInterface
02. public interface Function < T, R > {
03.     R apply(T t);
04. }
```



Function Functional Interfaces (cont.)

- Function instance called yearRetriever.
  - It accepts an argument of type LocalDate and returns a result of type Integer.
- The apply method accepts a LocalDate object and returns the year component corresponding to the LocalDate.

```
public class FunctionDemo {
    public static void main(String[] args) {
        Function<LocalDate, Integer> yearRetriever = date -> date.getYear();
        LocalDate today = LocalDate.now();
        System.out.println("Year corresponding to " + today +
                " is " + yearRetriever.apply(today));
    }
}
```

```
Output
Year corresponding to 2021-11-29 is 2021
```

Java Predefined-Functional Interfaces

Interface	Description
BiConsumer<T,U>	It represents an operation that accepts two input arguments and returns no result.
Consumer<T>	It represents an operation that accepts a single argument and returns no result.
Function<T,R>	It represents a function that accepts one argument and returns a result.
Predicate<T>	It represents a predicate (boolean-valued function) of one argument.
BiFunction<T,U,R>	It represents a function that accepts two arguments and returns a a result.
BinaryOperator<T>	It represents an operation upon two operands of the same data type. It returns a result of the same type as the operands.

Java Predefined-Functional Interfaces (cont.)

Interface	Description
BiPredicate<T,U>	It represents a predicate (boolean-valued function) of two arguments.
BooleanSupplier	It represents a supplier of boolean-valued results.
DoubleBinaryOperator	It represents an operation upon two double type operands and returns a double type value.
DoubleConsumer	It represents an operation that accepts a single double type argument and returns no result.
DoubleFunction<R>	It represents a function that accepts a double type argument and produces a result.
DoublePredicate	It represents a predicate (boolean-valued function) of one double type argument.
DoubleSupplier	It represents a supplier of double type results.
DoubleToIntFunction	It represents a function that accepts a double type argument and produces an int type result.
DoubleToLongFunction	It represents a function that accepts a double type argument and produces a long type result.

Java Predefined-Functional Interfaces (cont.)

Interface	Description
DoubleUnaryOperator	It represents an operation on a single double type operand that produces a double type result.
IntBinaryOperator	It represents an operation upon two int type operands and returns an int type result.
IntConsumer	It represents an operation that accepts a single integer argument and returns no result.
IntFunction<R>	It represents a function that accepts an integer argument and returns a result.
IntPredicate	It represents a predicate (boolean-valued function) of one integer argument.
IntSupplier	It represents a supplier of integer type.
IntToDoubleFunction	It represents a function that accepts an integer argument and returns a double.
IntToLongFunction	It represents a function that accepts an integer argument and returns a long.
IntUnaryOperator	It represents an operation on a single integer operand that produces an integer result.

Java Predefined-Functional Interfaces (cont.)

Interface	Description
LongBinaryOperator	It represents an operation upon two long type operands and returns a long type result.
LongConsumer	It represents an operation that accepts a single long type argument and returns no result.
LongFunction<R>	It represents a function that accepts a long type argument and returns a result.
LongPredicate	It represents a predicate (boolean-valued function) of one long type argument.
LongSupplier	It represents a supplier of long type results.
LongToDoubleFunction	It represents a function that accepts a long type argument and returns a result of double type.
LongToIntFunction	It represents a function that accepts a long type argument and returns an integer result.
LongUnaryOperator	It represents an operation on a single long type operand that returns a long type result.

Java Predefined-Functional Interfaces (cont.)

Interface	Description
ObjDoubleConsumer<T>	It represents an operation that accepts an object and a double argument, and returns no result.
ObjIntConsumer<T>	It represents an operation that accepts an object and an integer argument. It does not return result.
ObjLongConsumer<T>	It represents an operation that accepts an object and a long argument, it returns no result.
Supplier<T>	It represents a supplier of results.
ToDoubleBiFunction<T,U>	It represents a function that accepts two arguments and produces a double type result.
ToDoubleFunction<T>	It represents a function that returns a double type result.
ToIntBiFunction<T,U>	It represents a function that accepts two arguments and returns an integer.
ToIntFunction<T>	It represents a function that returns an integer.
ToLongBiFunction<T,U>	It represents a function that accepts two arguments and returns a result of long type.
ToLongFunction<T>	It represents a function that returns a result of long type.
UnaryOperator<T>	It represents an operation on a single operand that returnsa a result of the same type as its operand.

Lambda Expression

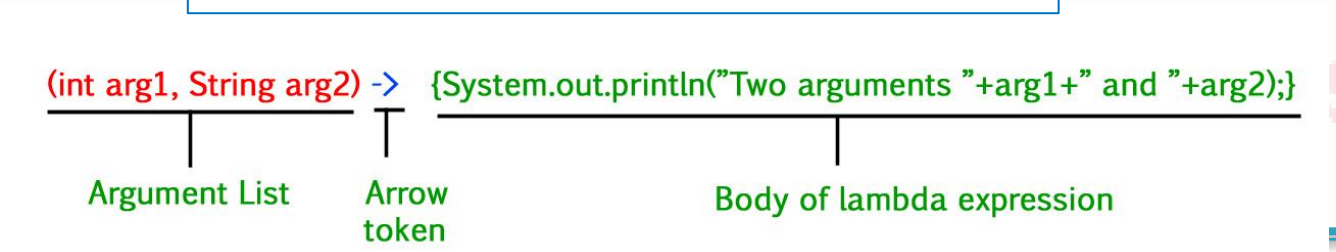
Java Lambda Expressions

- Lambda expressions are added in Java 8 and provide below functionalities.
  - Enable to treat functionality as a method argument, or code as data.
  - A function that can be created without belonging to any class.
  - A lambda expression can be passed around as if it was an object and executed on demand.

Lambda Syntax

- Java lambda expression is consisted of three components.
  - **1) Argument-list**: It can be empty or non-empty as well.
  - **2) Arrow-token**: It is used to link arguments-list and body of expression.
  - **3) Body**: It contains expressions and statements for lambda expression.

```
(argument-list) -> {body}
```



Lambda Syntax (cont.)

Syntax	Example
parameter -> expression	`x -> x * x`
parameter -> block	`s -> { System.out.println(s); }`
(parameters) -> expression	`(x, y) -> Math.sqrt(x*x + y*y)`
(parameters) -> block	`(s1, s2) ->` `{ System.out.println(s1 + "," + s2); }`
(parameter decls) -> expression	`(double x, double y) -> Math.sqrt(x*x + y*y)`
(parameters decls) -> block	`(List<?> list) ->` `{ Arrays.shuffle(list); Arrays.sort(list); }`

Without Lambda Expression

```
1  package lambda;
2
3  interface Drawable {
4      public void draw();
5  }
6
7  public class LambdaExpressionExample {
8      public static void main(String[] args) {
9          int width = 10;
10
11         // without lambda, Drawable implementation using anonymous class
12         Drawable d = new Drawable() {
13             public void draw() {
14                 System.out.println("Drawing " + width);
15             }
16         };
17         d.draw();
18     }
19 }
```

Java Lambda Expression Example

```
1  package lambda;
2
3  @FunctionalInterface // It is optional
4  interface Drawable {
5      public void draw();
6  }
7
8  public class LambdaExpressionExample2 {
9      public static void main(String[] args) {
10         int width = 10;
11
12         // with lambda
13         Drawable d2 = () -> {
14             System.out.println("Drawing " + width);
15         };
16         d2.draw();
17     }
18 }
```

Java Lambda Expression Example: No Parameter

```
1  package lambda;
2
3  interface Sayable {
4      public String say();
5  }
6
7  public class LambdaExpressionExample3 {
8      public static void main(String[] args) {
9          Sayable s = () -> {
10             return "I have nothing to say.";
11         };
12         System.out.println(s.say());
13     }
14 }
```

Java Lambda Expression Example: Single Parameter

```
1  package lambda;
2
3  interface Sayable {
4      public String say(String name);
5  }
6
7  public class LambdaExpressionExample4 {
8      public static void main(String[] args) {
9
10         // Lambda expression with single parameter.
11         Sayable s1 = (name) -> {
12             return "Hello, " + name;
13         };
14         System.out.println(s1.say("Sonoo"));
15
16         // You can omit function parentheses
17         Sayable s2 = name -> {
18             return "Hello, " + name;
19         };
20         System.out.println(s2.say("Sonoo"));
21     }
22 }
```

Java Lambda Expression Example: Multiple Parameters

```
1  package lambda;
2
3  interface Addable {
4      int add(int a, int b);
5  }
6
7  public class LambdaExpressionExample5 {
8      public static void main(String[] args) {
9
10         // Multiple parameters in lambda expression
11         Addable ad1 = (a, b) -> (a + b);
12         System.out.println(ad1.add(10, 20));
13
14         // Multiple parameters with data type in lambda expression
15         Addable ad2 = (int a, int b) -> (a + b);
16         System.out.println(ad2.add(100, 200));
17     }
18 }
```

Java Lambda Expression Example: with or without return keyword

- If there is only one statement ➔ may or may not use return keyword.

```
1  package lambda;
2
3  interface Addable {
4      int add(int a, int b);
5  }
6
7  public class LambdaExpressionExample6 {
8      public static void main(String[] args) {
9
10         // Lambda expression without return keyword.
11         Addable ad1 = (a, b) -> (a + b);
12         System.out.println(ad1.add(10, 20));
13
14         // Lambda expression with return keyword.
15         Addable ad2 = (int a, int b) -> {
16             return (a + b);
17         };
18         System.out.println(ad2.add(100, 200));
19     }
20 }
```

Java Lambda Expression Example: Multiple Statements

- Multiple statements ➔ must use return keyword.

```
 1 package lambda;
 2
 3 @FunctionalInterface
 4 interface Sayable {
 5     String say(String message);
 6 }
 7
 8 public class LambdaExpressionExample8 {
 9     public static void main(String[] args) {
10
11         // You can pass multiple statements in lambda expression
12         Sayable person = (message) -> {
13             String str1 = "I would like to say, ";
14             String str2 = str1 + message;
15             return str2;
16         };
17         System.out.println(person.say("time is precious."));
18     }
19 }
```

Java Lambda Expression Example: Foreach Loop

```
 1 package lambda;
 2
 3 import java.util.*;
 4
 5 public class LambdaExpressionExample7 {
 6     public static void main(String[] args) {
 7
 8         List<String> list = new ArrayList<String>();
 9         list.add("ankit");
10         list.add("mayank");
11         list.add("irfan");
12         list.add("jai");
13
14         list.forEach((n) -> System.out.println(n));
15     }
16 }
```

Java Lambda Expression Example: Comparator

```
class Product {
    int id;
    String name;
    float price;

    public Product(int id, String name, float price) {
        this.id = id;
        this.name = name;
        this.price = price;
    }
}
```

```
public class LambdaExpressionExample10 {
    public static void main(String[] args) {
        List<Product> list = new ArrayList<Product>();

        // Adding Products
        list.add(new Product(1, "HP Laptop", 25000f));
        list.add(new Product(3, "Keyboard", 300f));
        list.add(new Product(2, "Dell Mouse", 150f));

        System.out.println("Sorting on the basis of name...");

        // implementing lambda expression
        Collections.sort(list, (p1, p2) -> {
            return p1.name.compareTo(p2.name);
        });
        for (Product p : list) {
            System.out.println(p.id + " " + p.name + " " + p.price);
        }
    }
}
```

Java Lambda Expression Example: Creating Thread

```
1  package lambda;
2
3  public class LambdaExpressionExample9 {
4      public static void main(String[] args) {
5
6          // Thread Example without lambda
7          Runnable r1 = new Runnable() {
8              public void run() {
9                  System.out.println("Thread1 is running...");
10             }
11         };
12         Thread t1 = new Thread(r1);
13         t1.start();
14         // Thread Example with lambda
15         Runnable r2 = () -> {
16             System.out.println("Thread2 is running...");
17         };
18         Thread t2 = new Thread(r2);
19         t2.start();
20     }
21 }
```

Java Lambda Expression Example: Event Listener

```
public class LambdaEventListenerExample {
    public static void main(String[] args) {
        JTextField tf = new JTextField();
        tf.setBounds(50, 50, 150, 20);
        JButton b = new JButton("click");
        b.setBounds(80, 100, 70, 30);

        // lambda expression implementing here.
        b.addActionListener(e -> {
            tf.setText("hello swing");
        });

        JFrame f = new JFrame();
        f.add(tf);
        f.add(b);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setLayout(null);
        f.setSize(300, 200);
        f.setVisible(true);
    }
}
```

Java Lambda Expression Example: Filter Collection Data

```
public class LambdaExpressionExample11 {
    public static void main(String[] args) {
        List<Product> list = new ArrayList<Product>();
        list.add(new Product(1, "Samsung A5", 17000f));
        list.add(new Product(3, "Iphone 6S", 65000f));
        list.add(new Product(2, "Sony Xperia", 25000f));
        list.add(new Product(4, "Nokia Lumia", 15000f));
        list.add(new Product(5, "Redmi4 ", 26000f));
        list.add(new Product(6, "Lenevo Vibe", 19000f));

        // using lambda to filter data
        Stream<Product> filtered_data = list.stream().filter(p -> p.price > 20000);

        // using lambda to iterate through collection
        filtered_data.forEach(product -> System.out.println(product.name + ": " + product.price));
    }
}
```

Method references

Method references

- A concise notation for certain lambdas
  - lambda expression:

```
accounts.forEach(a -> a.addInterest());
```

  - method reference:

```
accounts.forEach(Account::addInterest);
```

- Advantage (over lambdas)
  - reuse existing method
- Needs type inference context for target type
  - similar to lambda expressions

Method references

- 4 types of method references:
  - Static method reference
  - Instance Method (Bound receiver)
  - Instance Method (UnBound receiver)
  - Constructor reference

Method references (cont.)

- Various forms of method references ...
  - static method: Type::MethodName
    - e.g. System::currentTimeMillis
  - constructor: Type::new
    - e.g. String::new

Method references (cont.)

- Various forms of method references ...
  - non-static method w/ unbound receiver: Type::MethodName
    - e.g. String::length
  - non-static method w/ bound receiver: Expr::Method
    - e.g. System.out::println

    (System.out is an instance of PrintStream)

Reference to instance method

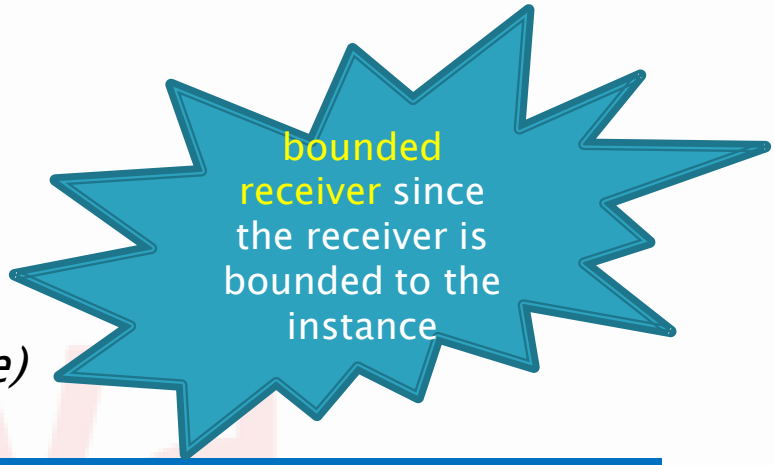
- Situation:
  - instance method needs an instance on which it can be invoked
    - called: receiver
- Two possibilities:
  - receiver is explicitly provided in an expression
    - called: bound receiver
  - receiver is implicitly provided from the context
    - called: unbound receiver

Bound receiver

- Calling a method in a lambda to an external object that already exists
- Syntax:

```
instance"::"methodName
```

(instance: represents any object instance)



- Example:

```
List<String> stringList = ... ;
stringList.forEach(System.out::print);
```

receiver

  - With lambda

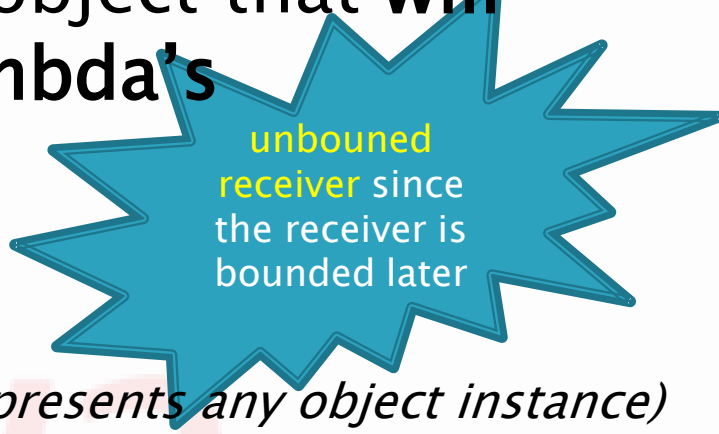
```
stringList.forEach((String s) -> System.out.print(s));
```

Unbound receiver

- Referring to a method of an object that **will be supplied as one of the lambda's parameters**
- Syntax:

  Type "::"MethodName

  (Type: represents any object instance)



- Example:

```
Stream<String> stringStream = ... ;
stringStream.sorted(String::compareToIgnoreCase);
```

  - With lambda:

```
stringStream.sorted(
(String s1, String s2) -> s1.compareToIgnoreCase(s2));
```

Compare these situations

- Example 1:

```
Stream<Person> psp = ... ;
psp.sorted(Person::compareByName);
```

```
class Person {
        public static int compareByName(Person a, Person b) { … }
}
```

- Example 2:

```
Stream<String> stringStream = ... ;
stringStream.sorted(String::compareToIgnoreCase);
```

```
class String {
        public int compareToIgnoreClase(String str) { … }
}
```

Notes

- Situations for three different ways of method reference:
  - (args) -> ClassName.staticMethod(args) can be ClassName::staticMethod
    - This is static (you can think as unBound also)
  - (arg0, rest) -> arg0.instanceMethod(rest) can be ClassName::instanceMethod (arg0 is of type ClassName)
    - This is unBound
  - (args) -> instance.instanceMethod(args) can be instance::instanceMethod
    - This is Bound

Notes (cont.)

- Method references do not specify argument type(s)
- Compiler infers from context
  - which overloaded version fits

```
List<String> stringList = ... ;
stringList.forEach(System.out::print);
```

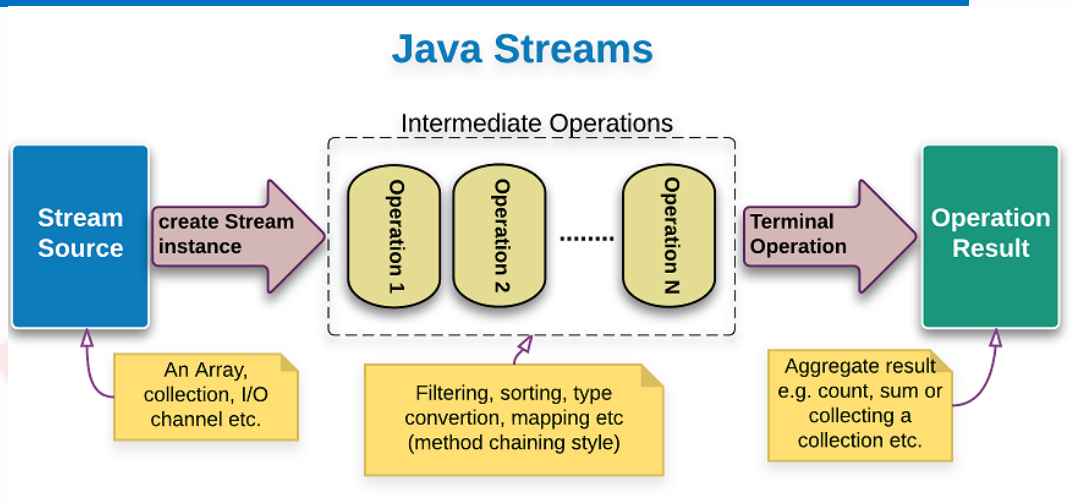
➔ **void print(String s)**

- Resort to lambda expressions
  - if compiler fails or a different version should be used

Lambda expression vs Method references

Lambda Expressions	Equivalent Method References
(String s) -> Integer.parseInt(s)	Integer::parseInt
(String s) -> s.toLowerCase()	String::toLowerCase
(int i) -> System.out.println(i)	System.out::println
(Student s) -> s.getName()	Student::getName
() -> s.getName()	s::getName where *'s'* refers to *Student* object which already exist.
() -> new Student()	Student::new

Streams



What is a stream?

- A bunch of data objects, typically from a collection, array, or input device, for bulk data processing
- Processed by a pipeline
  - A single stream generator (data source)
  - Zero or more intermediate stream operations
  - A single terminal stream operation
- Supports mostly-functional data processing
- Enables painless parallelism
  - Simply replace `stream` with `parallelStream`
  - We may or may not see a performance improvement

What is a stream?

- interface java.util.stream.Stream<T>
  - Consists of classes, interfaces and enum to allows functional-style operations on the elements
  - Supports forEach, filter, map, reduce, and more
- Two new methods in java.util.Collection<T>
  - Stream<T> stream(), sequential functionality
  - Stream<T> parallelStream(), parallel functionality

```
List<Account>accountCol = ... ;
Stream<Account>accounts = accountCol.stream();
Stream<Account>millionaires =
accounts.filter(a -> a.balance() > 1000000);
```

Streams' features

- Stream does not store elements.
  - It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
- Stream is functional in nature.
  - Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- Stream is lazy and evaluates code only when required.
- The elements of a stream are only visited once during the life of a stream.
  - Like an Iterator, a new stream must be generated to revisit the same elements of the source.

Stream Example

```
public class StreamExample {

    List<Integer> numbers = Arrays.asList(7, 2, 5, 4, 2, 1);

    public void withoutStream() {
        long count = 0;
        for (Integer number : numbers) {
            if (number % 2 == 0) {
                count++;
            }
        }
        System.out.printf("There are %d elements that are even", count);
    }

    public void withStream() {
        long count = numbers.stream().filter(num -> num % 2 == 0).count();
        System.out.printf("There are %d elements that are even", count);
    }
}
```

Streams and their operations

- Streams do not store their elements
  - not a collection, but created from a collection, array, ...
  - view/adaptor of a data source (collection, array, ...)
- Streams provide functional operations forEach, filter, map, reduce, ...
  - applied to elements of underlying data source

Streams and their operations (cont.)

- Actually applied functionality is two-folded
  - user-defined: functionality passed as parameter
  - framework method: stream operations

- Separation between "*what to do*" & "*how to do*"
  - user => what functionality to apply
  - library => how to apply functionality (parallel/sequential, lazy/eager, out-of-order)

```
accounts.filter(a -> a.balance() > 1000000);
accounts.forEach(a -> a.addInterest());
```

Parameters of stream operations ...

- ... can be
  - lambda expressions
  - method references
  - (inner classes)
- Example: forEach

```
void forEach(Consumer<? super T> consumer);
public interface Consumer<T> {
        public void accept(T t);
}
accounts.forEach((Account a) -> { a.addInterest(); });
accounts.forEach(a -> a.addInterest());
accounts.forEach(Account::addInterest);
```

Advantages of using streams

- Efficient and shortcode
- A very easy way to do parallel computation without having to worry about the multi-threading implementations.
- Providing a large set of operations that can be utilized in many scenarios.
- Providing a more memory efficient way as the stream is closed ➔ no extra objects and variables created.
- A wide range of functionalities can be implemented (using lambda expressions)

Parallel stream

- Streams provide support for parallel computation to exploit multiple cores on a processing unit.
  - by creating a stream().parallel() or any Collection.parallelStream()
- Example: Find fruits, whose names end with "e" using a parallel stream.

```
List newList = fruits.stream().parallel().filter(
x-x.endsWith("e")).collect(Collectors.toList());

Or

Stream fruitStream = fruits.parallelStream();
```

Map

- Build a new sequence, where each element is the result of a mapping from an element of the original sequence
  - An intermediate operation that consumes a stream and produces a stream

- Example: Convert each element of the list to uppercase

```
List newList = fruits.stream().map(
x->x.toUpperCase()).collect(Collectors.toList());
```

OR using method reference

```
List newList = fruits.stream().map(
String::toUpperCase).collect(Collectors.toList());
```

Filter

- Build a new sequence that is the result of a filter applied to each element in the original collection
  - A intermediate operation that consumes a stream and produces a stream.

- Example: filter all elements that start with “A”

```
List newList = fruits.stream().map(
x->x.toUpperCase()).filter(x->x.startsWith("A")).collect(Coll
ectors.toList());
```

Reduce

- Produce a single result from all elements of the sequence
  - A terminal operation that consumes a stream and produces a single result and not a stream.
- Example: Concatenate all fruits that start with "A".

```
List newList = fruits.stream().map(
x->x.toUpperCase()).filter(x->x.startsWith("A")).reduce("",
(x,y)-> x+y);
```

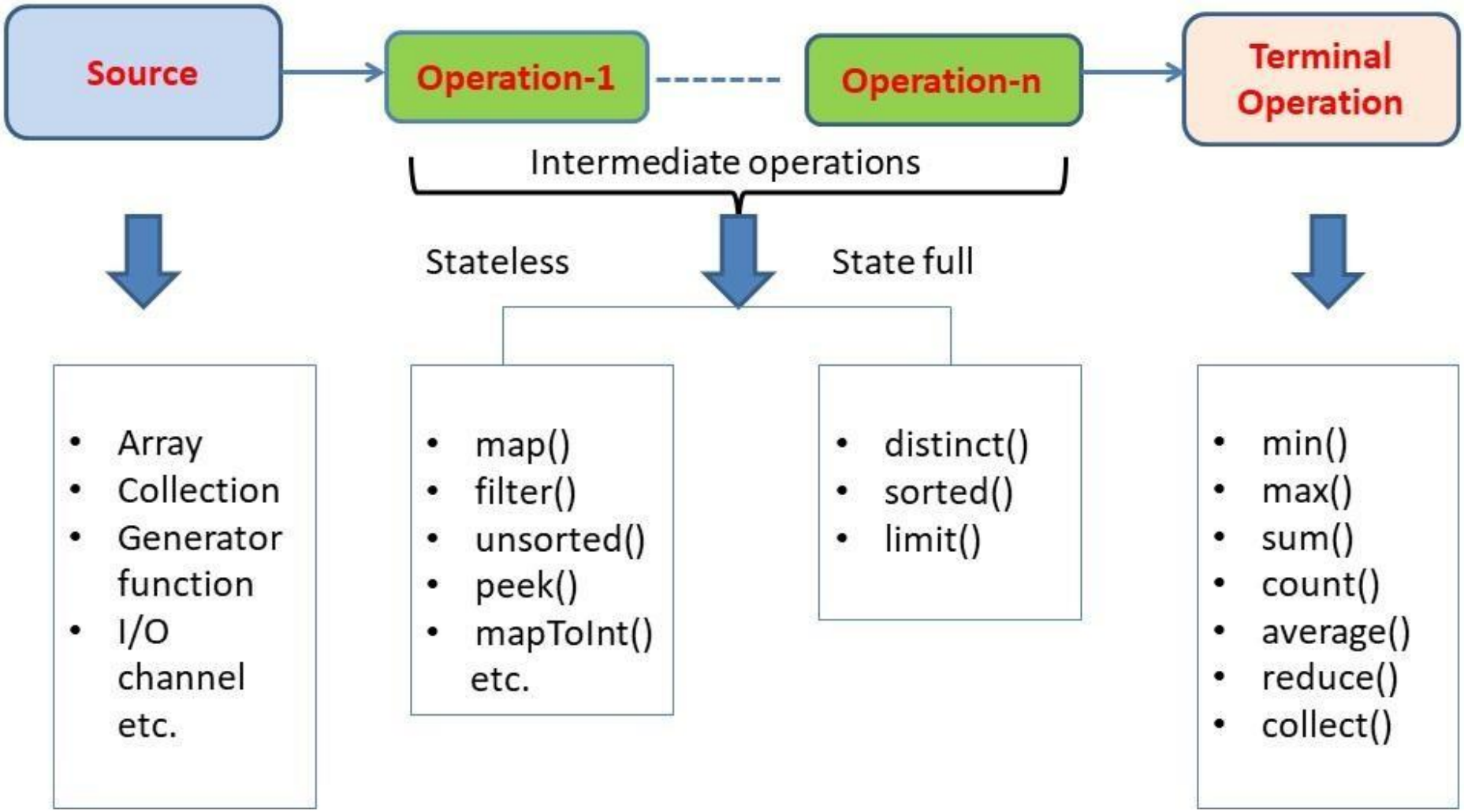
Collectors

- Java.util.stream.Collectors: is a final utility class that contains various methods which are used with terminal operation Stream.collect().
  - Collectors also provide some very useful utility methods: Collectors.groupingBy(), Collectors.Counting(), …
- **Example:** Let's count no fruits of each type

```
Map<String, Long> map =
fruits.stream().map(x->x.toUpperCase()).collect(Collectors.gr
oupingBy(Function.identity(),Collectors.counting()));
```

Stream pipeline

- Components of a stream pipeline



Primitive streams

- Streams for elements with primitive type:
  - IntStream, LongStream, DoubleStream
- Reason: performance
  - code optimization; no buffering of intermediate stream results; easier to handle parallel streams
- No stream types for char and float
  - use stream type of respective 'bigger' primitive type
  - IntStream for char, and DoubleStream for float
    - e.g. interface CharSequence contains:
      - IntStream chars();

How to obtain a stream?

- java.util.Collection<T>
  - Stream<T> stream(), sequential functionality
  - Stream<T> parallelStream(), parallel functionality
- java.util.Arrays
  - static <T> Stream<T> stream(T[] array)
  - plus overloaded versions (primitive types, ...)
- many more ...
- Collections allow to obtain a parallel stream directly
  - in all other cases use stream's method: parallel()

Streams vs Collections

Collections	Streams
Collections are mainly used to store and group the data.	Streams are mainly used to perform operations on data.
You can add or remove elements from collections.	You can't add or remove elements from streams.
Collections have to be iterated externally.	Streams are internally iterated.
Collections can be traversed multiple times.	Streams are traversable only once.
Collections are eagerly constructed.	Streams are lazily constructed.

1. Conceptual Difference

- Collections: used to store and group the data in a particular data structure like *List*, *Set* or *Map*.
- Streams: used to perform complex data processing operations like *filtering*, *matching*, *mapping* etc on stored data such as arrays, collections or I/O resources.

```
// Usage of collections
// Collections are mainly used to store the data
// Here, names are stored as List
List<String> names = new ArrayList<>();
names.add("Charlie");
names.add("Douglas");
names.add("Sundaraman");
names.add("Charlie");
names.add("Yuki");
// Usage of streams
// Streams are mainly used to perform operations on data
// like selecting only unique names
names.stream().distinct().forEach(System.out::println);
```

Output :

Charlie
Douglas
Sundaraman
Yuki

2. Data Modification

- We can add to or remove elements from collections.
- But, we can't add to or remove elements from streams.
  - Stream consumes a source, performs operations on it and returns a result.

```
List<String> names = Arrays.asList("Charlie", "Douglas", "Jacob");
// Adding elements to names
names.add("Sundaraman");
names.add("Yuki");
// Removing elements from names
names.remove(2);

// getting stream of unique names
Stream<String> uniqueNames = names.stream().distinct();
// You can't add or remove elements from stream
// There are no such methods in Stream
```

3. External Iteration Vs Internal Iteration

- Streams perform iteration internally (collections are externally iterated)

```
List<String> names = new ArrayList<>();
names.add("Charlie");
names.add("Douglas");
names.add("Sundaraman");
names.add("Charlie");
names.add("Yuki");
// External iteration of collections
for (String name : names) {
    System.out.println(name);
}
// Internal iteration of streams. No for loops
names.stream().map(String::toUpperCase).forEach(System.out::println);
```

4. Traversal

- Streams are traversable only once.
  - To traverse it again, you have to get new stream from the source again.
- But, collections can be traversed multiple times.

```
// Internal iteration of streams. No for loops
names.stream().map(String::toUpperCase).forEach(System.out::println);
List<Integer> numbers = Arrays.asList(4, 2, 8, 9, 5, 6, 7);
Stream<Integer> numbersGreaterThan5 = numbers.stream().
        filter(i -> i > 5);
// Traversing numbersGreaterThan5 stream first time
numbersGreaterThan5.forEach(System.out::println);
// Second time traversal will throw error
// Error : stream has already been operated upon or closed
numbersGreaterThan5.forEach(System.out::println);
```

5. Eager Construction Vs Lazy Construction

- Collections are eagerly constructed
  - i.e all the elements are computed at the beginning itself.
- But, streams are lazily constructed
  - i.e intermediate operations are not evaluated until terminal operation is invoked

```
List<Integer> numbers1 = Arrays.asList(4, 2, 8, 9, 5, 6, 7);

numbers1.stream().filter(i -> i >= 5).limit(3).
                                    forEach(System.out::println);

//Here, not all numbers are evaluated.
//numbers are evaluated until 3 numbers >= 5 are found.
```

Collectors

Java 8

Collectors Class

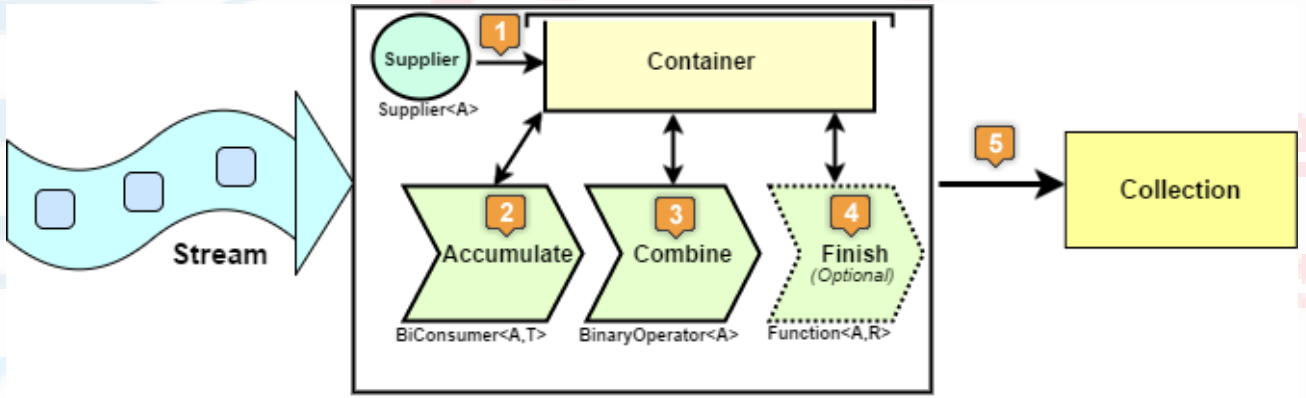
What is Collectors?

- *Stream.collect():* one of the Java 8's *Stream API*'s terminal methods.
  - It allows us to perform mutable fold operations (repackaging elements to some data structures and applying some additional logic, concatenating them, etc.) on data elements held in a *Stream* instance.
- The strategy for this operation is provided via the *Collector* interface implementation.

What is Collectors?

- Collectors is a final class that extends Object class.

- It provides reduction operations:
  - accumulating elements into collections,
  - summarizing elements according to various criteria, etc.



A Collector in Action

Collectors example

- Group a list of transactions by currency to obtain the sum of the values of all transactions with that currency
  - returning a Map<Currency, Integer>
- Partition a list of transactions into two groups: expensive and not expensive
  - returning a Map<Boolean, List<Transaction>>
- Create multilevel groupings such as grouping transactions by cities and then further categorizing by whether they're expensive or not
  - returning a Map<String, Map<Boolean, List<Transaction>>>

Built-in methods

- Collectors.toList()
- Collectors.toSet()
- Collectors.toCollection()
- Collectors.toMap()
- Collectors.collectingAndThen()
- Collectors.joining()
- Collectors.counting()
- Collectors.summarizingDouble/Long/Int()
- Collectors.averagingDouble/Long/Int()
- Collectors.summingDouble/Long/Int()
- Collectors.maxBy()
- Collectors.minBy()
- Collectors.groupingBy()
- Collectors.partitioningBy()
- Collectors.reducing()
- …

Collectors.toList()

- Used for collecting all *Stream* elements into a *List* instance

```
public class Product {
    private String name;
    private int price;

    public Product(String name, int price) {
        this.name = name;
        this.price = price;
    }

    public static void main(String[] args) {
        List<Product> ps = new ArrayList<>();
        ps.add(new Product("Pepsi", 10000));
        ps.add(new Product("C2", 8000));
        ps.add(new Product("Sting", 12000));

        List<Integer> prices = ps.stream().map(x -> x.price).collect(Collectors.toList());
        System.out.println(prices);
    }
}
```

Collectors.toCollection()

- Collectors.toList():
  - return an ArrayList or a LinkedList or any other implementation of the List interface
- Collectors.toList():
  - return an HashSet or a LinkedHashSet or any other implementation of the Set interface
- How to specific a concrete implementation of Set or List?

Collectors.toCollection()

- Collectors.toCollection():
  - can accept desired constructor method reference like ArrayList::new, LinkedList::new, HashSet::new, …

```
public static void main(String[] args) {
    List<Product> ps = new ArrayList<>();
    ps.add(new Product("Pepsi", 10000, "SoftDrink"));
    ps.add(new Product("C2", 8000, "SoftDrink"));
    ps.add(new Product("Snack", 12000, "Chips"));

    Set<Product> re = ps.stream().collect(Collectors.toCollection(HashSet::new));
    System.out.println(re);
}
```

Collectors.toCollection()

- Also use the toCollection() method to add elements of a stream to an existing list (or a set)

```
public static void main(String[] args) {
    List<Product> ps = new ArrayList<>();
    ps.add(new Product("Pepsi", 10000, "SoftDrink"));
    ps.add(new Product("C2", 8000, "SoftDrink"));
    ps.add(new Product("Snack", 12000, "Chips"));

    Set<Product> re = new HashSet<>();

    ps.stream().collect(Collectors.toCollection(() -> re));
    System.out.println(re);
}
```

Collectors.toMap()

- *Map* collector can be used to collect *Stream* elements into a *Map* instance. Two functions:
  - keyMapper: extracte a *Map* key from a *Stream* element
  - valueMapper: extracte a value associated with a given key

```
public static void main(String[] args) {
    List<Product> ps = new ArrayList<>();
    ps.add(new Product("Pepsi", 10000, "SoftDrink"));
    ps.add(new Product("C2", 8000, "SoftDrink"));
    ps.add(new Product("Snack", 12000, "Chips"));

    Map<String, Integer> map = ps.stream().collect(
            Collectors.toMap(Product::getName, Product::getPrice));
    System.out.println(map);
```

Collectors.toMap()

- How about this code fragment?

```
public static void main(String[] args) {
    List<Product> ps = new ArrayList<>();
    ps.add(new Product("Pepsi", 10000, "SoftDrink"));
    ps.add(new Product("C2", 8000, "SoftDrink"));
    ps.add(new Product("Snack", 12000, "Chips"));

    Map<String, Integer> map = ps.stream().collect(
            Collectors.toMap(Product::getType, Product::getPrice));
    System.out.println(map);
```

```
Exception in thread "main" java.lang.IllegalStateException: Duplicate key SoftDrink (attempted merging values 10000 and 8
        at java.base/java.util.stream.Collectors.duplicateKeyException(Collectors.java:135)
        at java.base/java.util.stream.Collectors.lambda$uniqKeysMapAccumulator$1(Collectors.java:182)
        at java.base/java.util.stream.ReduceOps$3ReducingSink.accept(ReduceOps.java:169)
        at java.base/java.util.ArrayList$ArrayListSpliterator.forEachRemaining(ArrayList.java:1625)
        at java.base/java.util.stream.AbstractPipeline.copyInto(AbstractPipeline.java:484)
        at java.base/java.util.stream.AbstractPipeline.wrapAndCopyInto(AbstractPipeline.java:474)
        at java.base/java.util.stream.ReduceOps$ReduceOp.evaluateSequential(ReduceOps.java:913)
        at java.base/java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:234)
        at java.base/java.util.stream.ReferencePipeline.collect(ReferencePipeline.java:682)
        at java8.collectors.Product.main(Product.java:37)
```

Collectors.toMap()

- Solution to the code fragment:

```
public static void main(String[] args) {
    List<Product> ps = new ArrayList<>();
    ps.add(new Product("Pepsi", 10000, "SoftDrink"));
    ps.add(new Product("C2", 8000, "SoftDrink"));
    ps.add(new Product("Snack", 12000, "Chips"));

    Map<String, Integer> map = ps.stream().collect(
            Collectors.toMap(Product::getType, Product::getPrice,
                    (item, identicalItem) -> item+identicalItem));
    System.out.println(map);
}
```

Collectors.counting()

- *Counting* is a simple collector that allows simply counting of all *Stream* elements.

```
public static void main(String[] args) {
    List<Product> ps = new ArrayList<>();
    ps.add(new Product("Pepsi", 10000, "SoftDrink"));
    ps.add(new Product("C2", 8000, "SoftDrink"));
    ps.add(new Product("Snack", 12000, "Chips"));

    Map<String, Long> result = ps.stream()
            .collect(Collectors.groupingBy(Product::getType, Collectors.counting()));
    System.out.println(result);
}
```

Collectors.groupingBy()

- *GroupingBy* collector is used for grouping objects by some property and storing results in a *Map* instance.

```
public static void main(String[] args) {
    List<Product> ps = new ArrayList<>();
    ps.add(new Product("Pepsi", 10000, "SoftDrink"));
    ps.add(new Product("C2", 8000, "SoftDrink"));
    ps.add(new Product("Snack", 12000, "Chips"));

    Map<String, Set<Product>> result = ps.stream()
            .collect(Collectors.groupingBy(Product::getType, Collectors.toSet()));
    System.out.println(result);
}
```

Collectors.partitioningBy()

- *PartitioningBy* is a specialized case of *groupingBy*
  - accepts a *Predicate* instance
  - and collects *Stream* elements into a *Map* instance
  - that stores *Boolean* values as keys and collections as values.

```
public static void main(String[] args) {
    List<Product> ps = new ArrayList<>();
    ps.add(new Product("Pepsi", 10000, "SoftDrink"));
    ps.add(new Product("C2", 8000, "SoftDrink"));
    ps.add(new Product("Snack", 12000, "Chips"));

    Map<Boolean, List<Product>> result = ps.stream()
            .collect(Collectors.partitioningBy(p -> p.getPrice()>=10000));
    System.out.println(result);
}
```

Collectors.minBy()/maxBy()

- These methods return a collector that outputs the minimum/maximum element according to the provided comparator.

```
public static void main(String[] args) {
    List<Product> ps = new ArrayList<>();
    ps.add(new Product("Pepsi", 10000, "SoftDrink"));
    ps.add(new Product("C2", 8000, "SoftDrink"));
    ps.add(new Product("Snack", 12000, "Chips"));

    Comparator<Product> comp = Comparator.comparing(Product::getPrice);
    Product minProduct = ps.stream().collect(Collectors.minBy(comp)).get();
    Product maxProduct = ps.stream().collect(Collectors.maxBy(comp)).get();

    System.out.println(minProduct + "\n" + maxProduct);
}
```

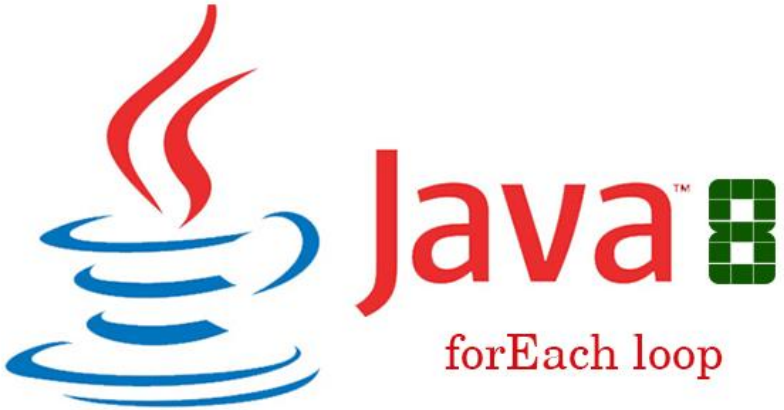
Collectors.joining()

- used to group all elements to a string.
  - returns one collector that joins all elements to a string.

```
public static void main(String[] args) {
    Stream<String> strStream = Stream.of("one","two","three","four","five");
    System.out.println(strStream.collect(Collectors.joining("-")));
}
```

```
one-two-three-four-five
```

forEach



What is forEach?

- A new, concise and interesting way to iterate over a collection.
  - can be used to loop or iterate a Map, List, Set, or Stream.
- Defined in Iterable and Stream interface.
- In Java 8, we can loop a List with forEach + lambda expression or method reference.

forEachWithList

```
public class ForEachWithList {
    public static void main(String[] args) {
        List<Person> items = new ArrayList<>();
        items.add(new Person(100, "Ramesh"));
        items.add(new Person(100, "A"));
        items.add(new Person(100, "B"));
        items.add(new Person(100, "C"));
        items.add(new Person(100, "D"));

        // lambda
        items.forEach(item -> System.out.println(item.getName()));

        // Output : C
        items.forEach(item -> {
            if ("C".equals(item)) {
                System.out.println(item);
            }
        });

        // method reference
        // Output : A,B,C,D,E
        items.forEach(System.out::println);

        // Stream and filter
        // Output : B
        items.stream().filter(s -> s.getName().equals("Ramesh")).forEach(System.out::println);
    }
}
```

forEachWithMap

```
public class ForEachWithMap {
    public static void main(String[] args) {
        // Before Java 8, how to loop map
        Map<Integer, Person> map = new HashMap<>();
        map.put(1, new Person(100, "Ramesh"));
        map.put(2, new Person(100, "Ram"));
        map.put(3, new Person(100, "Prakash"));
        map.put(4, new Person(100, "Amir"));
        map.put(5, new Person(100, "Sharuk"));

        // In Java 8, you can loop a Map with forEach + lambda expression.
        map.forEach((k, p) -> {
            System.out.println(k);
            System.out.println(p.getName());
        });
    }
}
```

forEach() method with Set

```
public class ForEachWithSet {
    public static void main(String[] args) {
        Set<String> items = new HashSet<>();
        items.add("A"); items.add("B");
        items.add("C"); items.add("D");
        items.add("E");
        // before java 8
        for (final String item : items) {
            System.out.println(item);
        }
        // java 8 with lambda expression
        // Output : A,B,C,D,E
        items.forEach(item -> System.out.println(item));

        // Output : C
        items.forEach(item -> {
            if ("C".equals(item)) {
                System.out.println(item);
            }
        });
        // method reference
        items.forEach(System.out::println);
        // Stream and filter
        items.stream().filter(s -> s.contains("B")).forEach(System.out::println);
    }
}
```

forEachOrdered in Stream

- For sequential streams, the order of elements (during iteration) is same as the order in the stream source.
- While using *parallel streams:*
  - forEach() method does not gaurantee the element ordering to provide the advantages of parallelism.
  - use forEachOrdered() if order of the elements matter during the iteration

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

        Consumer<Integer> action = System.out::println;

        numberList.stream().filter(n -> n % 2 == 0).parallel().forEachOrdered(action);
    }
}
```

Interface vs Abstract class (from Java 8)

Interface vs Abstract class

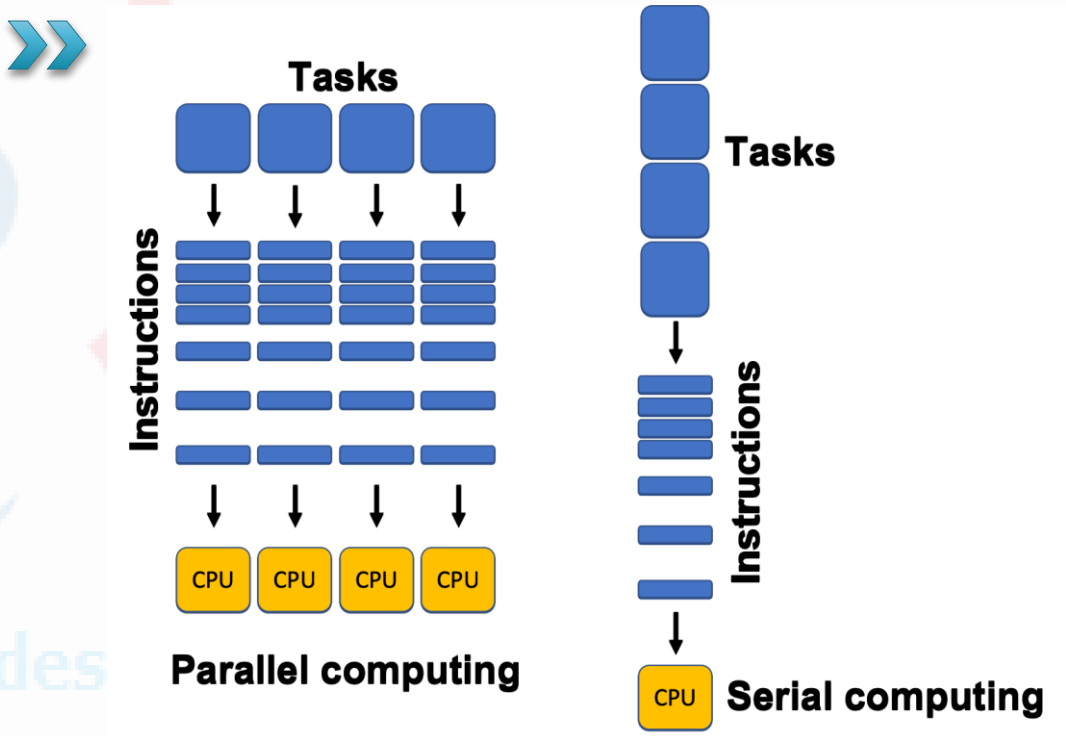
	Interface	Abstract Class
Constructors	✗	✓
Static Fields	✓	✓
Non-static Fields	✗	✓
Final Fields	✓	✓
Non-final Fields	✗	✓
Private Fields & Methods	✗	✓
Protected Fields & Methods	✗	✓
Public Fields & Methods	✓	✓
Abstract methods	✓	✓
Static Methods	✓	✓
Final Methods	✗	✓
Non-final Methods	✓	✓
Default Methods	✓	✗

Default method

- Java 8 introduces a new concept of default method implementation in interfaces.
  - This capability is added for backward compatibility so that old interfaces can be used to leverage the lambda expression capability of Java 8.
  - i.e., List/Collection do not have 'forEach' method declaration

```
interface Sayable {
    // Default method
    default void say() {
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
}
public class DefaultMethods implements Sayable {
    public void sayMore(String msg) { // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say(); // calling default method
        dm.sayMore("Work is worship"); // calling abstract method
    }
}
```

Parallel Array Sorting



Java Parallel Array Sorting

- Java provides a new additional feature in Array class which is used to sort array elements parallel.

- New methods has added to java.util.Arrays package that use the JSR 166 Fork/Join parallelism common pool to provide sorting of arrays in parallel.

- The methods are called parallelSort() and are overloaded for all the primitive data types and Comparable objects.

Java Parallel Array Sorting (cont.)

- Some selected methods:

Methods	Description
public static void parallelSort(byte[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(byte[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(char[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(char[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.
public static void parallelSort(double[] a)	It sorts the specified array into ascending numerical order.
public static void parallelSort(double[] a, int fromIndex, int toIndex)	It sorts the specified range of the array into ascending numerical order. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.

Java Parallel Array Sorting (cont.)

- Example:

```
public class ParallelArraySorting {
    public static void main(String[] args) {
        // Creating an integer array
        int[] arr = { 5, 8, 1, 0, 6, 9 };
        // Iterating array elements
        for (int i : arr) {
            System.out.print(i + " ");
        }
        // Sorting array elements parallel
        Arrays.parallelSort(arr);
        System.out.println("\nArray elements after sorting");
        // Iterating array elements
        for (int i : arr) {
            System.out.print(i + " ");
        }
    }
}
```

FACULTY OF INFORMATION TECHNOLOGY

Java
Data Structures