Expresiones Lambda

2025

Functional Interfaces

Functional Interface is an interface that defines a single abstract operation (function).

Problem:

- It is not possible to partially implement an interface. The concrete class must implement all abstract methods that it has inherited.
- An interface with many abstract methods is not convenient to use. A class may have to provide implementations for methods that it does not actually need.

Solution:

- A class may implement as many interfaces as needed.
- An interface can define just one abstract method; no extra operations that could be "unwanted baggage" have to be implemented.
- It is a flexible and recommended approach to designing interfaces.

```
public interface X {
 void a();
 void b();
 void c();
}
```

```
public class Y
       implements X {
 public void a() {}
 public void b() {}
 public void c() {}
}
```

```
public interface A {
 void a();
}
```

```
public interface B {
 void b();
}
```

```
public class Y
       implements A, B {
 public void a() {}
 public void b() {}
}
```

Understand Lambda Expressions

Lambda expression is an inline implementation of a functional interface.

- Lambda expression infers its properties from the context:
  - Context of invocation infers which functional interface is implemented.
  - Functional interface's only abstract method infers the method that has to be overridden.
  - Generics infer which parameters the method should have.
  - Return type infers a return statement.
- Lambda expression may just contain parameter names and the desired algorithm.
- Lambda token `->` separates parameter list from the method body.

Invocation context

```
List<Product> products = ...
Collections.sort(products,<Comparator>);
```

Anonymous inner class

```
Collections.sort(products, new Comparator<Product>(){
  public int compare(Product p1, Product p2) {
    return p1.getPrice().compareTo(p2.getPrice());
  }
});
```

Lambda expression

```
Collections.sort(products, (p1, p2) -> p1.getPrice().compareTo(p2.getPrice()) );
```

Define Lambda Expression Parameters and Body

Parameter definitions of Lambda expressions:

- To apply modifiers (annotations or keywords) to parameters, define them using:
  - Specific types
  - Locally inferred types
- When no modifier is required, you may just infer types from the context.
- Formal body `{ }` and `return` statements are optional when using a simple expression.
- Round brackets for parameters are also optional.
- Expressions can be predefined and reused.

```
List<String> list = new ArrayList<>();
Comparator<String> sortText = (s1,s2) -> s1.compareTo(s2);
list.removeIf( (final String s) -> s.equals("remove me") );
list.removeIf( (final var s) -> s.equals("remove me") );
list.sort( (s1,s2) -> { return s1.compareTo(s2); } );
Collections.sort(list, sortText);
```

✣ **Note:** Annotations or modifiers can be used to impose restrictions upon parameters.
For example, making parameters final would prevent their reassignment within the expression.

Use Method References

Lambda expressions may use method referencing.

- Reference method is semantically identical to the method that lambda expression is implementing.
- `<Class>::<staticMethod>` : Reference a static method
- `<object>::<instanceMethod>` : Reference an instance method of a particular object
- `<Class>::<instanceMethod>` : Reference an instance method of an arbitrary object of a particular type
- `<Class>::new` - reference a constructor (see notes)

```
public class TextFilter {
  public static boolean removeA(String s) {
    return s.equals("remove A");
  }
  public int sortText(String s1, String s2) {
    return s1.compareTo(s2);
  }
}
```

- **Note:** Class `String` has an instance method `compareToIgnoreCase` that implements non-case-sensitive text sorting.

```
TextFilter filter = new TextFilter();
List<String> list = new ArrayList<>();
list.removeIf(s -> TextFilter.removeA(s));
list.removeIf(TextFilter::removeA);                    // same as the line above
Collections.sort(list, (s1,s2) -> filter.sortText(s1,s2));
Collections.sort(list, filter::sortText);              // same as the line above
Collections.sort(list, (s1,s2) -> s1.compareToIgnoreCase(s2));
Collections.sort(list, String::compareToIgnoreCase); // same as the line above
```

Default and Static Methods in Functional Interfaces

Interfaces may provide additional nonabstract methods.

- Lambda expression implements the only abstract method provided by the functional interface.
- default and static methods may be defined by the interface to provide additional features.
- private methods could be present, but they are not visible outside of the interface.
- There is no requirement to override default methods unless there is a conflict between different interfaces that a given class implements.
- Lambda expressions cannot cause such conflicts, because each one is an inline implementation of exactly one interface.

```
public interface SomeInterface {
  public static final int SOME_VALUE = 123;
  void someAbstractMethod();
  public default void someDefaultMethod() { }
  private void somePrivateMethod() { }
  public static void someStaticMethod() { }
}
```

Use Default and Static Methods of the `Comparator` Interface

Examples of default methods provided by the `java.util.Comparator` interface:

- `thenComparing` adds additional comparators.
- `reversed` reverses sorting order.

Examples of static methods provided by the `Comparator` interface:

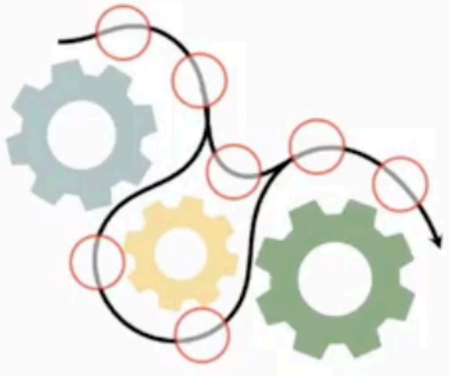
- `nullsFirst` and `nullsLast` return comparators that enable sorting collections with null values.

```
List<Product> menu = new ArrayList<>();
menu.add(new Food("Cake", BigDecimal.valueOf(1.99)));
menu.add(new Food("Cookie", BigDecimal.valueOf(2.99)));
menu.add(new Drink("Tea", BigDecimal.valueOf(2.99)));
menu.add(new Drink("Coffee", BigDecimal.valueOf(2.99)));
Comparator<Product> sortNames = (p1, p2) -> p1.getName().compareTo(p2.getName());
Comparator<Product> sortPrices = (p1, p2) -> p1.getPrice().compareTo(p2.getPrice());
Collections.sort(menu, sortNames.thenComparing(sortPrices).reversed());
menu.add(null);
Collections.sort(menu, Comparator.nullsFirst(sortNames));
```

Note: Additional primitive variants of these methods are provided to avoid excessive boxing and unboxing.

Streams API

Characteristics of Streams



Stream is an immutable flow of elements.

- Stream processing can be sequential (default) or parallel.
- Once an element is processed, it is no longer available from the stream.
- Stream pipeline traversal uses method chaining - intermediate operations return streams.
- Pipeline traversal is lazy.
  - Intermediate actions are deferred until stream is traversed by the terminal operation.
  - The chain of activities could be fused into a single pass on data.
  - Stream processing ends as soon as the result is determined; remaining stream data can be ignored.
- Stream operations use functional interfaces and can be implemented as lambda expressions.
- Stream may represent both finite and infinite flows of elements.

```
List<Product> list = new ArrayList<>();
```

Stream:

```
list.stream().parallel()
             .filter(p->p.getPrice()>10)
             .forEach(p->p.setDiscount(0.2));
```

Loop:

```
for (Product p: list) {
  if (p.getPrice()>10)) {
    p.setDiscount(0.2));
  }
}
```

✣ Examples assume that stream source is a collection of products.

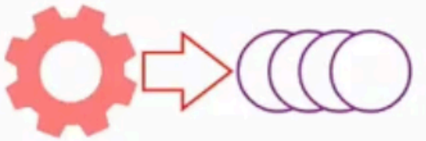
Create Streams Using Stream API

Streams handling is described by the following interfaces:

- `BaseStream` defines core stream behaviors,
  such as managing the stream in a parallel or sequential mode.
- `Stream`, `DoubleStream`, `IntStream`, `LongStream` interfaces
  extend the `BaseStream` and provide stream processing operations.
- Streams use generics.
- To avoid excessive boxing and unboxing, primitive stream variants are also provided.
- Stream can be obtained from any collection and array or by using static methods of the `Stream` class.
- Many other Java APIs can create and use streams (see notes).

```
BaseStream
└→ Stream
└→ DoubleStream
└→ IntStream
└→ LongStream
```

```
IntStream.generate(()->(int)(Math.random()*10)).takeWhile(n->n!=3).sum();

Stream.of(new Food(),new Drink()).forEach(p->p.setPrice(1));

List<Product> list = new ArrayList();
Product[] array = {new Drink(), new Food()};

list.stream().parallel().mapToDouble(p->p.getPrice()).sum();
Arrays.stream(array).filter(p->p.getPrice()>2).forEach(p->p.setDiscount(0.1));
```



Stream Pipeline Processing Operations

Stream handling operation categories:

- Intermediate: Perform action and produce another stream
- Terminal: Traverse stream pipeline and end the stream processing
- Short-circuit: Produce finite result, even if presented with infinite input
- Use functional interfaces from the `java.util.function` package
- Can be implemented using lambda expressions
- Basic function purposes:
  - `Predicate` performs tests.
  - `Function` converts types.
  - `UnaryOperator` (a variant of function) converts values.
  - `Consumer` processes elements.
  - `Supplier` produces elements.

```
Stream.generate(<Supplier>)
      .filter(<Predicate>)
      .peek(<Consumer>)
      .map(<Function>/<UnaryOperator>)
      .forEach(<Consumer>);
```

Intermediate	Terminal
filter	forEach
map	forEachOrdered
flatMap	count
peek	min
distinct	max
sorted	sum
dropWhile	average
skip	collect
limit	reduce
takeWhile	allMatch
	anyMatch
	noneMatch
	findAny
	findFirst

Using Functional Interfaces

Stream operations use functional interfaces:

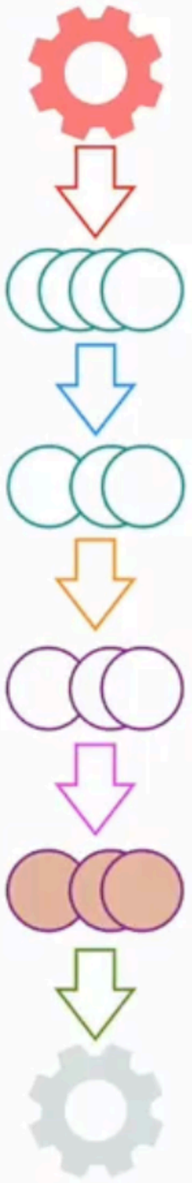
- Located in `java.util.function` package
- Can be implemented using lambda expressions
- Basic function shapes are:
  - `Predicate<T>` defines method `boolean test(T t)` to apply conditions to filter elements.
  - `Function<T,R>` defines method `R apply(T t)` to convert types of elements.
  - `UnaryOperator<T>` (variant of Function) defines method `T apply(T t)` to convert values.
  - `Consumer<T>` defines method `void accept(T t)` to process elements.
  - `Supplier<T>` defines method `T get()` to produce elements.

✣ See notes for `Supplier` and `UnaryOperator` examples.

Predicate	Function	UnaryOperator	Consumer	Supplier
`Predicate<T>`	`Function<T,R>`	`UnaryOperator<T>`	`Consumer<T>`	`Supplier<T>`

```
List<Product> list = new ArrayList<>();
list.stream()
    .filter(p -> p.getDiscount() == 0)
    .peek(p -> p.applyDiscount(0.1))
    .map(p -> p.getBestBefore())
    .forEach(d -> d.plusDays(1));
```

✣ Produce stream of products from the list
✣ Retain only products with no discount
✣ Apply discount of 10 percent
✣ Produce a stream of LocalDate objects
✣ Calculate the next day



Primitive Variants of Functional Interfaces

Improve stream pipeline handling performance by avoiding excessive auto-boxing-unboxing

	Predicate	Function	UnaryOperator	Consumer	Supplier
primitive output		**ToIntFunction<T>** ToLongFunction<T> ToDoubleFunction<T>			IntSupplier LongSupplier DoubleSupplier BooleanSupplier
primitive input	**IntPredicate** LongPredicate DoublePredicate	IntFunction<R> LongFunction<R> DoubleFunction<R>		**IntConsumer** LongConsumer DoubleConsumer	
primitive input-output		IntToLongFunction IntToDoubleFunction LongToIntFunction LongToDoubleFunction DoubleToIntFunction DoubleToLongFunction	IntUnaryOperator LongUnaryOperator DoubleUnaryOperator		

```
Stream.of("ONE","TWO","THREE","FOUR")
        .mapToInt(s->s.length())
        .peek(i->System.out.println(i))
        .filter(i->i>3)
        .sum();
```

- The example maps strings to int values to filter and compute the result.
- See notes for more examples.

Bi-argument Variants of Functional Interfaces

Process more than one value at a time:
(Extra parameter is provided compared to basic function interfaces)

- `BiPredicate<T,U>` defines method `boolean test(T t, U u)` to apply conditions.
- `BiFunction<T,U,R>` defines method `R apply(T t, U u)` to convert two types into a single result.
- `BinaryOperator<T>` (variant of `BiFunction`) defines method `T apply(T t1, T t2)` to convert two values.
- `BiConsumer<T,U>` defines method `void accept(T t, U u)` to process a pair of elements.

	Predicate	Function	UnaryOperator	Consumer
	`BiPredicate<T,U>`	`BiFunction<T,U,R>`	`BinaryOperator<T>`	`BiConsumer<T,U>`
primitive variants		`ToIntBiFunction<T,U>`	`IntBinaryOperator`	`ObjIntConsumer<T>`
		`ToLongBiFunction<T,U>`	`LongBinaryOperator`	`ObjLongConsumer<T>`
		`ToDoubleBiFunction<T,U>`		`ObjDoubleConsumer<T>`

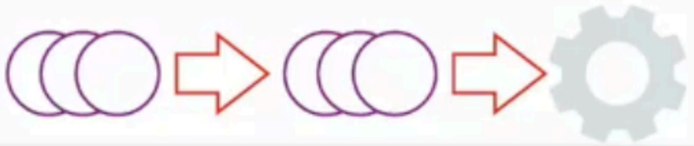
```
Product p1 = new Food("Cake", BigDecimal.valueOf(3.10));
Product p2 = new Drink("Tea", BigDecimal.valueOf(1.20));
Map<Product, Integer> items = new HashMap();
items.put(p1, Integer.valueOf(1));
items.put(p2, Integer.valueOf(3));
items.forEach((p, q) -> p.getPrice().multiply(BigDecimal.valueOf(q.intValue())));
```

- The example is processing product prices and quantities to produce totals.

Perform Actions with Stream Pipeline Elements

Intermediate or terminal actions are handled by `peek` and `forEach` operations.

- Operations `peek`, `forEach`, and `forEachOrdered` accept `Consumer<T>` interface.
- Lambda expression must implement the abstract `void accept(T t)` method.
- The default `andThen` method provided by the `Consumer` interface combines consumers together.
- Unlike the `forEachOrdered`, the `forEach` operation does not guarantee respecting the order of elements, which is actually beneficial for parallel stream processing.



```
Consumer<Product> expireProduct = (p) -> p.setBestBefore(LocalDate.now());
Consumer<Product> discountProduct = (p) -> p.setDiscount(BigDecimal.valueOf(0.1));

list.stream().forEach(expireProduct.andThen(discountProduct));

list.stream().peek(expireProduct)
             .filter(p.getPrice().compareTo(BigDecimal.valueOf(10)) > 0)
             .forEach(discountProduct);
```

- Lambda expressions can be implemented inline.
- Consumer actions can be combined or applied at different points within the pipeline.
- Operation `filter` accepts a `Predicate` (details are covered next).

Perform Filtering of Stream Pipeline Elements

Filtering of the pipeline content is performed by the `filter` operation.

- Method `filter` accepts `Predicate<T>` interface and returns a stream comprising only elements that satisfy the filter criteria.
- Lambda expression must implement abstract `boolean test(T t)` method.
- Default methods provided by the `Predicate`:
  - `and` combines predicates like the `&&` operator.
  - `or` combines predicates like the `||` operator.
  - `negate` returns a predicate that represents the logical negation of this predicate.
- Static methods provided by the `Predicate` interface:
  - `not` returns a predicate that is the negation of the supplied predicate.
  - `isEqual` returns a predicate that compares the supplied object with the contents of the collection.

```
Predicate<Product> foodFilter = p -> p instanceof Food;
Predicate<Product> priceFilter = p -> p.getPrice().compareTo(BigDecimal.valueOf(2)) < 0;
list.stream().filter(foodFilter.negate().or(priceFilter))
             .forEach(p -> p.setDiscount(BigDecimal.valueOf(0.1));
list.stream().filter(Predicate.isEqual(new Food("Cake", BigDecimal.valueOf(1.99))))
             .forEach(p -> p.setDiscount(BigDecimal.valueOf(0.1));
```

Perform Mapping of Stream Pipeline Elements

Map stream elements to a new stream of different content type using `map` operation.

- Method `map` accepts a `Function<T, R>` interface and returns a new stream comprising elements produced by this function based on the original stream content.
- Lambda expression must implement abstract `R apply(T t)` method.
- Default methods provided by the `Function`:
  - `andThen` and `compose` combine functions together.
- Static methods provided by the `Function` interface:
  - `identity` returns a function that always returns its input argument (equivalent of `t->t` function).
- Primitive variants of `map` are `mapToInt`, `mapToLong`, and `mapToDouble`.
- Interface `UnaryOperator<T>` is a variant of a `Function` that maps values without changing the type.

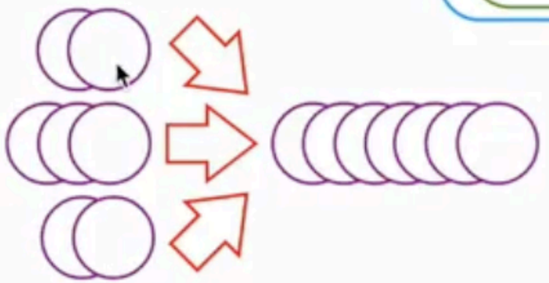
```
Function<Product, String> nameMapper = p -> p.getName();
UnaryOperator<String> trimMapper = n -> n.trim();
ToIntFunction<String> lengthMapper = n -> n.length();
list.stream().map(nameMapper.andThen(trimMapper)).mapToInt(lengthMapper).sum();
```

- Difference between `andThen` and `compose` is the order in which functions are combined.
- In the example above, `nameMapper` is applied before `trimMapper`.

Join Streams Using `flatMap` Operation

Flatten a number of streams into a single stream:

- Operation `Stream<R> flatMap(Function<T, Stream<R>> f)` merges streams.
- Primitive variants are `flatMapToInt`, `flatMapToLong`, and `flatMapToDouble` (see notes).




```
public class Order {
  private List<Product> items;
  public Stream<Product> items() {
    return items.stream();
  }
}
```

```
List<Order> orders = new ArrayList();
double x = orders.stream()
                 .flatMap(order->order.items())
                 .filter(item->item.getName().equals("Tea"))
                 .mapToDouble(item->item.getPrice().doubleValue())
                 .sum();
```

- The example processes a stream of Order objects, each containing a stream of Product objects, flattened into a single stream of Products. All products except Tea are filtered out, and a total sum of all Tea products sold in any of the orders is calculated.

Other Intermediate Stream Operations

Intermediate operations rearranging or reducing stream content:

- Operation `distinct()` returns a stream with no duplicates.
- Operations `sorted()` and `sorted(Comparator<T> t)` rearrange the order of elements.
- Operation `skip(long l)` skips a number of elements in the stream.
- Operation `takeWhile(Predicate p)` takes elements from the stream while they match the predicate.
- Operation `dropWhile(Predicate p)` removes elements from the stream while they match the predicate.
- Operation `limit(long l)` returns a stream of elements limited to a given size.


```
Stream.of("A","C","B","D","B","D")
      .distinct()
      .sorted()
      .skip(2)
      .forEach(s -> s.toLowerCase());
```

```
Stream.of("B","C","A","E","D","F")
      .takeWhile(s->!s.equals("D"))
      .dropWhile(s->!s.equals("C"))
      .limit(2)
      .forEach(s -> s.toLowerCase());
```

- Operations `takeWhile` and `dropWhile` may produce different results if the stream is ordered.
- Their cost can be significantly increased for ordered and parallel streams.

Short-Circuit Terminal Operations

Short-circuit terminal operations produce finite result even if presented with infinite input.

- All short-circuit operations terminate stream pipeline processing as soon as result is computed.
- Operation `allMatch(Predicate p)` returns true if all elements in the stream match the predicate.
- Operation `anyMatch(Predicate p)` returns true if any elements in the stream match the predicate.
- Operation `noneMatch(Predicate p)` returns true if no elements in the stream match the predicate.
- Operation `findAny()` returns an element from the stream wrapped in the `Optional` object.
- Operation `findFirst()` returns the first element from the stream wrapped in the `Optional` object.

```
String[] values = {"RED","GREEN","BLUE"};
boolean allGreen = Arrays.stream(values).allMatch(s->s.equals("GREEN"));
boolean anyGreen = Arrays.stream(values).anyMatch(s->s.equals("GREEN"));
boolean noneGreen = Arrays.stream(values).noneMatch(s->s.equals("GREEN"));
Optional<String> anyColour = Arrays.stream(values).findAny();
Optional<String> firstColour = Arrays.stream(values).findFirst();
```

Process Stream Using `count`, `min`, `max`, `sum`, `average` Operations

Terminal operations calculate values from stream content.

- Method `count` returns a number of elements in the stream.
- Methods `sum` and `average` are available only for primitive stream variants (int, long, and double).
- Method `average` returns an `OptionalDouble` object (primitive variant for `Optional` class).
- Methods `min` and `max` return an `Optional` object wrapper for the minimum or maximum value from the stream, according to the `Comparator` supplied.

```
String[] values = {"RED","GREEN","BLUE"};
long v1 = Arrays.stream(values).filter(s->s.indexOf('R') != -1).count();        // 2
int v2 = Arrays.stream(values).mapToInt(v->v.length()).sum();                   // 12
OptionalDouble v3 = Arrays.stream(values).mapToInt(v->v.length()).average();
double avgValue = v3.isPresent() ? v3.getAsDouble() : 0;                        // 4
Optional<String> v4 = Arrays.stream(values).max((s1,s2) -> s1.compareTo(s2));
Optional<String> v5 = Arrays.stream(values).min((s1,s2) -> s1.compareTo(s2));
String maxValue = (v4.isPresent()) ? v4.get() : "no data";                      // RED
String minValue = (v5.isPresent()) ? v5.get() : "no data";                      // BLUE
```

Notes on the use of `Optional` object:

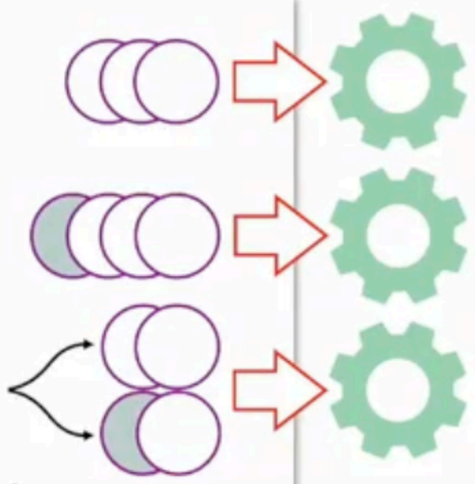
- Method `isPresent()` returns true if the value is present inside the `Optional` object.
- Method `get()` retrieves the value.

Aggregate Stream Data using `reduce` Operation

Produce a single result from the stream of values using the `reduce` operation:

- `Optional<T> reduce(BinaryOperator<T> accumulator)` performs accumulation of elements.
- `T reduce(T identity, BinaryOperator<T> accumulator)` identity acts as the initial (default) value.
- `<U> U reduce(U identity, BiFunction<U,T,U> accumulator, BinaryOperator<U> combiner)`
  BiFunction performs both value mapping and accumulation of values.
  BinaryOperator combines results produced by the BiFunction in parallel stream handling mode.

```
Optional<String> x1 = list.stream()
                          .map(p->p.getName())
                          .reduce((s1,s2)->s1+" "+s2);
                                 /*simple reduction*/
String x2 = list.stream()
                .map(p->p.getName())
                .reduce("",(s1,s2)->s1+" "+s2);
                /*reduction with initial(default) value*/
String x3 = list.stream()
                .parallel()
                .reduce("",(s,p)->p.getName()+" "+s,(s1,s2)->s1+s2);
          /*reduction with initial(default) value and a parallel combiner*/
```



- All examples perform stream reduction by concatenating product names into a single string.

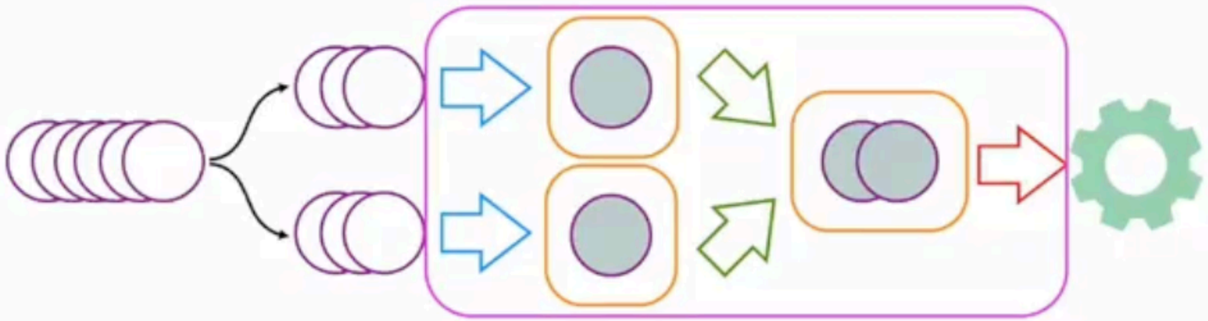
General Logic of the `collect` Operation

Perform a mutable reduction operation on the elements of the stream.

- Method `collect` accepts `Collector` interface implementation, which:
  - Produces new result containers using `Supplier`
  - Accumulates data elements into these result containers using `BiConsumer`
  - Combines result containers using `BinaryOperator`
  - Optionally performs a final transform of the processing result using the `Function`
- Class `Collectors` presents a number of predefined implementations of the `Collector` interface.

```
stream.collect(<supplier>, <accumulator>, <combiner>)
stream.collect(<collector>)
stream.collect(Collectors.collectingAndThen(<collector>,<finisher>))
```

- Operations `collect` and `reduce` both perform reduction. However, `collect` operation accumulates results within intermediate containers, which may improve performance.

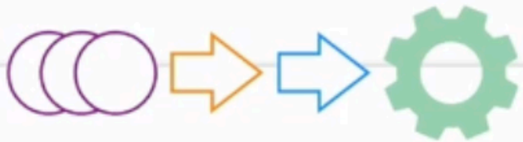


Using Basic Collectors

Predefined implementations of the `Collector` interface supplied by `Collectors` class:

- Calculating summary values such as average, min, max, count, sum
- Mapping and joining stream elements
- Gathering stream elements into a collection such as list, set, or map

```
DoubleSummaryStatistics stats =
       list.stream()
           .collect(Collectors.summarizingDouble(p->p.getPrice().doubleValue()));
```



```
String s1 =
       list.stream()
           .collect(Collectors.mapping(p->p.getName(),Collectors.joining(",")));
```

```
List<Product> drinks =
        list.stream().filter(p->p instanceof Drink).collect(Collectors.toList());
```

Process Stream Using `count`, `min`, `max`, `sum`, `average` Operations

Terminal operations calculate values from stream content.

- Method `count` returns a number of elements in the stream.
- Methods `sum` and `average` are available only for primitive stream variants (int, long, and double).
- Method `average` returns an `OptionalDouble` object (primitive variant for `Optional` class).
- Methods `min` and `max` return an `Optional` object wrapper for the minimum or maximum value from the stream, according to the `Comparator` supplied.

```
String[] values = {"RED","GREEN","BLUE"};
long v1 = Arrays.stream(values).filter(s->s.indexOf('R') != -1).count();        // 2
int v2 = Arrays.stream(values).mapToInt(v->v.length()).sum();                   // 12
OptionalDouble v3 = Arrays.stream(values).mapToInt(v->v.length()).average();
double avgValue = v3.isPresent() ? v3.getAsDouble() : 0;                        // 4
Optional<String> v4 = Arrays.stream(values).max((s1,s2) -> s1.compareTo(s2));
Optional<String> v5 = Arrays.stream(values).min((s1,s2) -> s1.compareTo(s2));
String maxValue = (v4.isPresent()) ? v4.get() : "no data";                      // RED
String minValue = (v5.isPresent()) ? v5.get() : "no data";                      // BLUE
```

Notes on the use of `Optional` object:

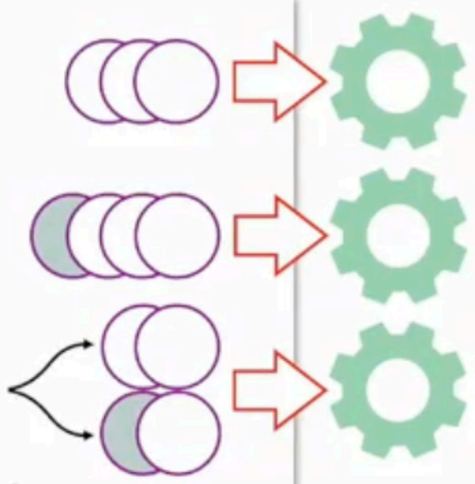
- Method `isPresent()` returns true if the value is present inside the `Optional` object.
- Method `get()` retrieves the value.

Aggregate Stream Data using `reduce` Operation

Produce a single result from the stream of values using the `reduce` operation:

- `Optional<T> reduce(BinaryOperator<T> accumulator)` performs accumulation of elements.
- `T reduce(T identity, BinaryOperator<T> accumulator)` identity acts as the initial (default) value.
- `<U> U reduce(U identity, BiFunction<U,T,U> accumulator, BinaryOperator<U> combiner)`
  BiFunction performs both value mapping and accumulation of values.
  BinaryOperator combines results produced by the BiFunction in parallel stream handling mode.

```
Optional<String> x1 = list.stream()
                          .map(p->p.getName())
                          .reduce((s1,s2)->s1+" "+s2);
                                 /*simple reduction*/
String x2 = list.stream()
                .map(p->p.getName())
                .reduce("",(s1,s2)->s1+" "+s2);
                /*reduction with initial(default) value*/
String x3 = list.stream()
                .parallel()
                .reduce("",(s,p)->p.getName()+" "+s,(s1,s2)->s1+s2);
          /*reduction with initial(default) value and a parallel combiner*/
```



- All examples perform stream reduction by concatenating product names into a single string.

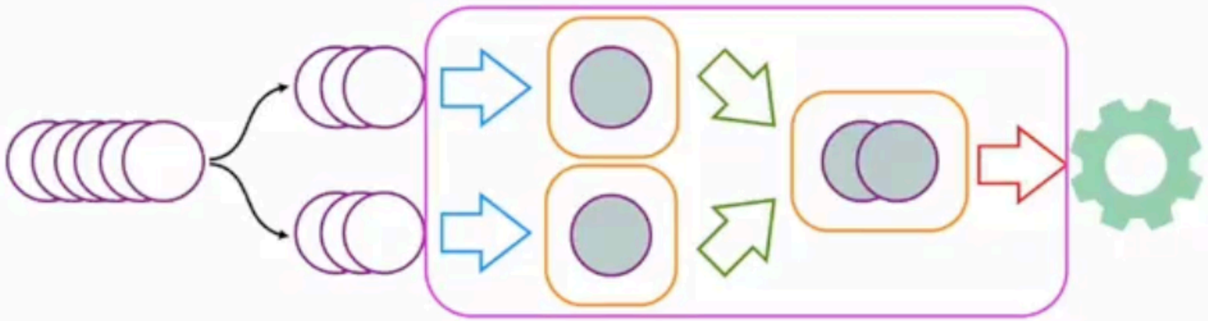
General Logic of the `collect` Operation

Perform a mutable reduction operation on the elements of the stream.

- Method `collect` accepts `Collector` interface implementation, which:
  - Produces new result containers using `Supplier`
  - Accumulates data elements into these result containers using `BiConsumer`
  - Combines result containers using `BinaryOperator`
  - Optionally performs a final transform of the processing result using the `Function`
- Class `Collectors` presents a number of predefined implementations of the `Collector` interface.

```
stream.collect(<supplier>, <accumulator>, <combiner>)
stream.collect(<collector>)
stream.collect(Collectors.collectingAndThen(<collector>,<finisher>))
```

- Operations `collect` and `reduce` both perform reduction. However, `collect` operation accumulates results within intermediate containers, which may improve performance.

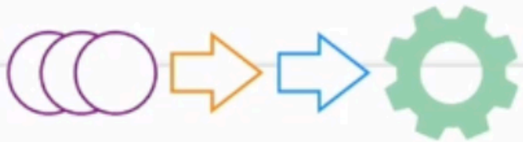


Using Basic Collectors

Predefined implementations of the `Collector` interface supplied by `Collectors` class:

- Calculating summary values such as average, min, max, count, sum
- Mapping and joining stream elements
- Gathering stream elements into a collection such as list, set, or map

```
DoubleSummaryStatistics stats =
       list.stream()
           .collect(Collectors.summarizingDouble(p->p.getPrice().doubleValue()));
```



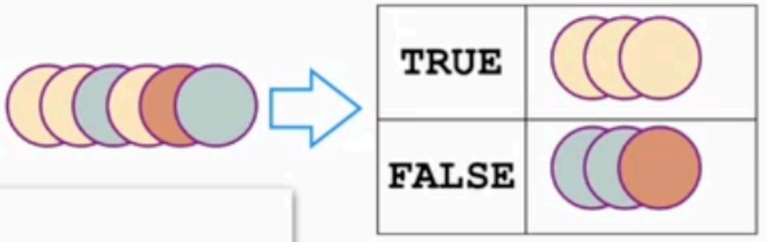
```
String s1 =
       list.stream()
           .collect(Collectors.mapping(p->p.getName(),Collectors.joining(",")));
```

```
List<Product> drinks =
        list.stream().filter(p->p instanceof Drink).collect(Collectors.toList());
```

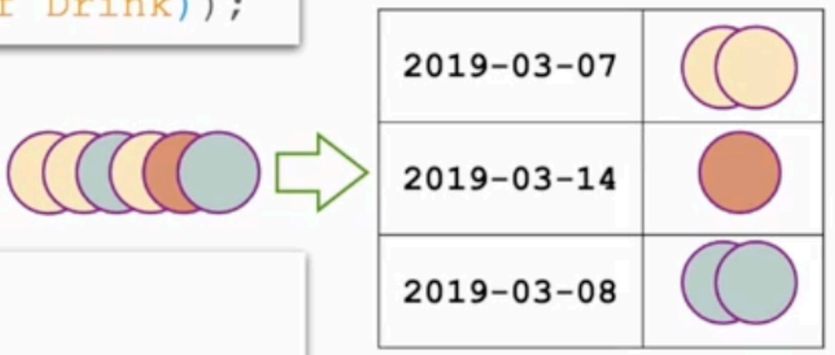
Perform Grouping or Partitioning of the Stream Content

Class Collectors provides Collector objects to subdivide stream elements into partitions or groups:

- Partitioning divides content into a map with two key values (boolean true/false) using `Predicate`.
- Grouping divides content into a map of multiple key values using `Function`.




```
Map<Boolean, List<Product>> productTypes =
  list.stream()
      .collect(Collectors.partitioningBy(p->p instanceof Drink));
```




```
Map<LocalDate, List<Product>> productGroups =
  list.stream()
      .collect(Collectors.groupingBy(p->p.getBestBefore()));
```

Mapping and Filtering with Respect to Groups or Partitions

Mapping and filtering in a multilevel reduction, using the downstream grouping or partitioning:

- `flatMapping` collector is applied to each input element in the stream before accumulation.
- `filtering` collector eliminates content from the stream without removing an entire group, if the group turns out to be empty.

```
ORDER
Customer customer;
LocalDate date;
List<Product> items;
```

Joe
2018-11-21
Tea 1.99
Cake 2.99

Bob
2018-11-21
Coffee 1.99

Joe
2018-11-22
Coffee 1.99
Cake 2.99

Stream of orders

```
Map<Customer, Set<Product>> customerProducts =
orders.collect(Collectors.groupingBy(o -> o.getCustomer(),
               Collectors.flatMapping(o -> o.getItems().stream(),
                                      Collectors.toSet())));
/*{Joe=[Tea, Coffee, Cake], Bob=[Coffee]}*/
```

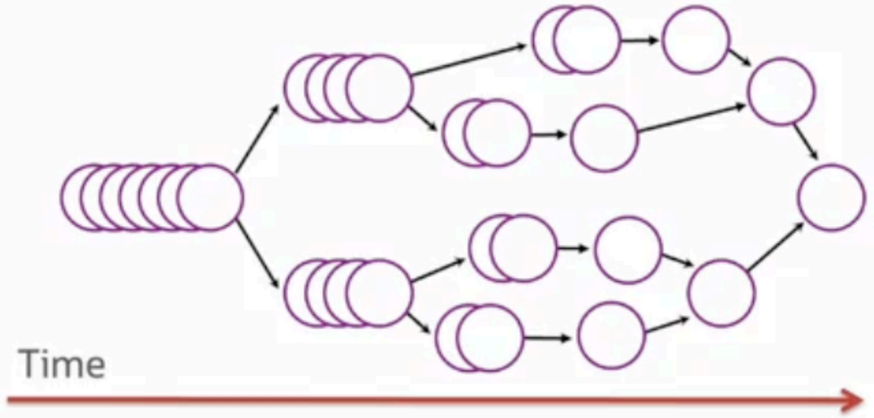
```
Map<Customer, Set<Order>> customerOrdersOnDate =
orders.collect(Collectors.groupingBy(o -> o.getCustomer(),
               Collectors.filtering(o -> o.getDate().equals(LocalDate.of(2018,11,22)),
                                    Collectors.toSet())));
/*{Joe=[Order[date=2018-11-22, customer Joe, products=[Coffee, Cake]]],Bob=[]}*/
```

- See notes for comparing with `flatMap` and `filter` methods.

Parallel Stream Processing

Parallel stream processing logic:

- Elements of the stream are subdivided into subsets.
- Subsets are processed in parallel.
- Subsets may be subdivided into further subsets.
- Processing order is stochastic (indeterminate).
- Subsets are then combined.



- Turn parallelism on or off using the `parallel` or `sequential` (default) methods.
- Entire stream processing will turn sequential or parallel depending on which method was invoked last.

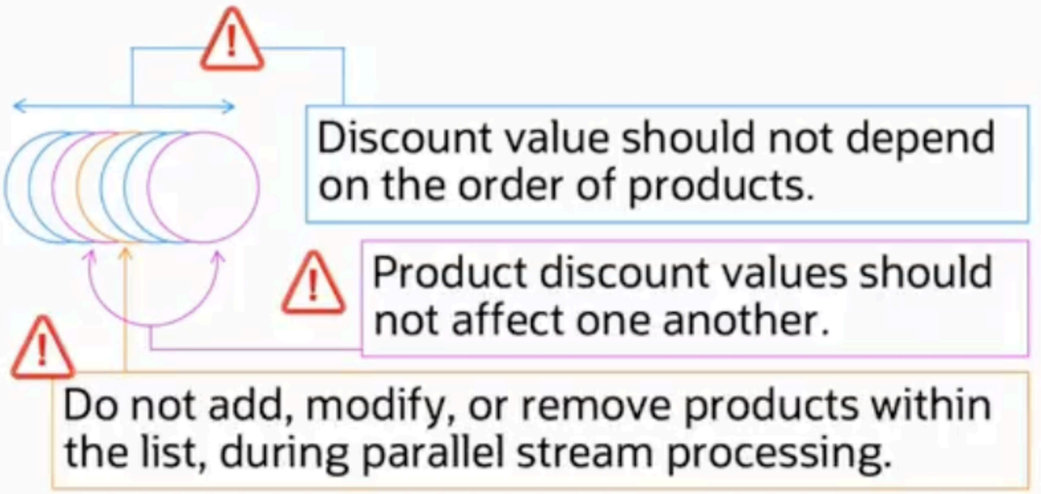
```
list.stream().parallel().mapToDouble(p->p.getPrice().doubleValue()).sum();
```

Parallel Stream Processing Guidelines

Parallel stream processing should observe the following guidelines:

- Stateless (state of one element must not affect another element)
- Noninterfering (data source must not be affected)
- Associative (result must not be affected by the order of operands)

```
List<Product> list = new ArrayList();
// populate list before processing starts
Double discount = list.stream().parallel()
        .mapToDouble(p -> p.getDiscount())
        .sum().get();
```



- The reason for these guidelines is the stochastic nature of stream processing:
  - It is not possible to predict the order in which different CPU cores complete their processing.
  - Stream workload can be dynamically reassigned to different CPU cores to achieve better performance.

Restrictions on Parallel Stream Processing

Incorrect handling of parallel streams can corrupt memory and slow down processing.

- Do not perform operations that require sequential access to a shared resource.
- Do not perform operations that modify shared resource.
- Use appropriate Collectors, such as:
  - `toMap` in sequential mode
  - `toConcurrentMap` in parallel mode

Parallel processing of the stream can only be beneficial if:

- Stream contains large number of elements
- Multiple CPU cores are available to physically parallelize computations
- Processing of stream elements requires significant CPU resources

```
List<BigDecimal> prices = new ArrayList<>();
list.stream()
    .parallel()
    .peek(p->System.out.println(p))
    .map(p->p.getPrice())
    .forEach(p->prices.add(p));
```

```
List<BigDecimal> prices = list.stream()
    .parallel()
    .map(p->p.getPrice())
    .collect(Collectors.toList());
```

```
Map<String, BigDecimal> namesAndPrices =
  list.stream()
      .parallel()
      .collect(Collectors.toConcurrentMap(p->p.getName(),
                                          p->p.getPrice()));
```

Spliterator

`Spliterator` is analogue of `Iterator`, with parallel processing capability.

Spliterator is used to process content of streams and collections.

- Process next element if it exists
- Process all remaining elements
- Once element is processed it is no longer available
- Attempt to split Spliterator object in half

```
Spliterator<Integer> s1 = new Random().ints(10,0,10).spliterator();
s1.tryAdvance(v->System.out.print(v));
Spliterator s2 = s1.trySplit();
if (s2 == null) {
  System.out.println("Did not split");
} else {
  s1.forEachRemaining(v->System.out.print(v));
  s2.forEachRemaining(v->System.out.print(v));
}
```

Notes:

- Method `tryAdvance()` is an alternative to the combination of `hasNext()` and `next()` methods of the `Iterator`.
- Method `forEachRemaining()` is an alternative to the entire `Iterator` loop

Spliterator Characteristics

Analyze Spliterator characteristics

- A Boolean test to determine if spliterator possesses a given characteristic
- Attempt to determine a number of available elements
- Characteristics change upon a split

```
Spliterator<Integer> s1 = new Random().ints(10,0,10).spliterator();
String characteristics =
  "Concurrent "+s.hasCharacteristics(Spliterator.CONCURRENT)+"\n"+
  "Distinct "+s.hasCharacteristics(Spliterator.DISTINCT)+"\n"+
  "Immutable "+s.hasCharacteristics(Spliterator.IMMUTABLE)+"\n"+
  "NonNull "+s.hasCharacteristics(Spliterator.NONNULL)+"\n"+
  "Ordered "+s.hasCharacteristics(Spliterator.ORDERED)+"\n"+
  "Sized "+s.hasCharacteristics(Spliterator.SIZED)+"\n"+
  "Sorted "+s.hasCharacteristics(Spliterator.SORTED)+"\n"+
  "Subsized "+s.hasCharacteristics(Spliterator.SUBSIZED);
System.out.println(characteristics);
System.out.println("Size "+s.getExactSizeIfKnown());
System.out.println("Estimate Size "+s.estimateSize());
```

Logging

Using Java Logging API

Use Logger class provided by Java Logging API to write logs.

- Each Logger is identified by a name.
- It is common practice to use class name as a logger name.
- There are seven levels of logging that allow you to log different levels of severity.

module-info.java

```
module demos {
  requires java.logging;
}
```

```
package demos;
import java.util.logging.*;
public class Test {
  private static Logger logger =
       Logger.getLogger(demos.Test.class.getName());
  public static void main(String[] args) {
    try {
      /* actions that can throw exceptions */
    }catch(Exception e) {
      logger.log(Level.ERROR,"Your error message",e);
    }
    logger.log(Level.INFO, "Your message");
    logger.info("Your message");
  }
}
```

✤ **Note:** Use of the `module-info` class is covered in the lesson titled “Modules and Deployment.”

Logging Method Categories

The logging methods are grouped into five main categories:

Method	Purpose
`log`	This is the most commonly used logging method. These overloaded methods have parameters for a level, a message, and optional additional parameters.
`logp`	Similar to the `log` methods, the "log precise" methods take additional parameters that specify a class and method name.
`logrb`	Similar to the `logp` methods, the "log with resource bundle" methods take a resource bundle name that is used to localize the message.

`entering`
`exiting`
`throwing`
	These are convenience methods used to log method entry, method exit, and exception throwing at the FINER level.

`severe`
`warning`
`config`
`info`
`fine`
`finer`
`finest`
	These are convenience methods used in simple cases when a message should be logged at the level indicated by the method name.

Guarded Logging

Use guarded logging to avoid processing messages that are due to be discarded.

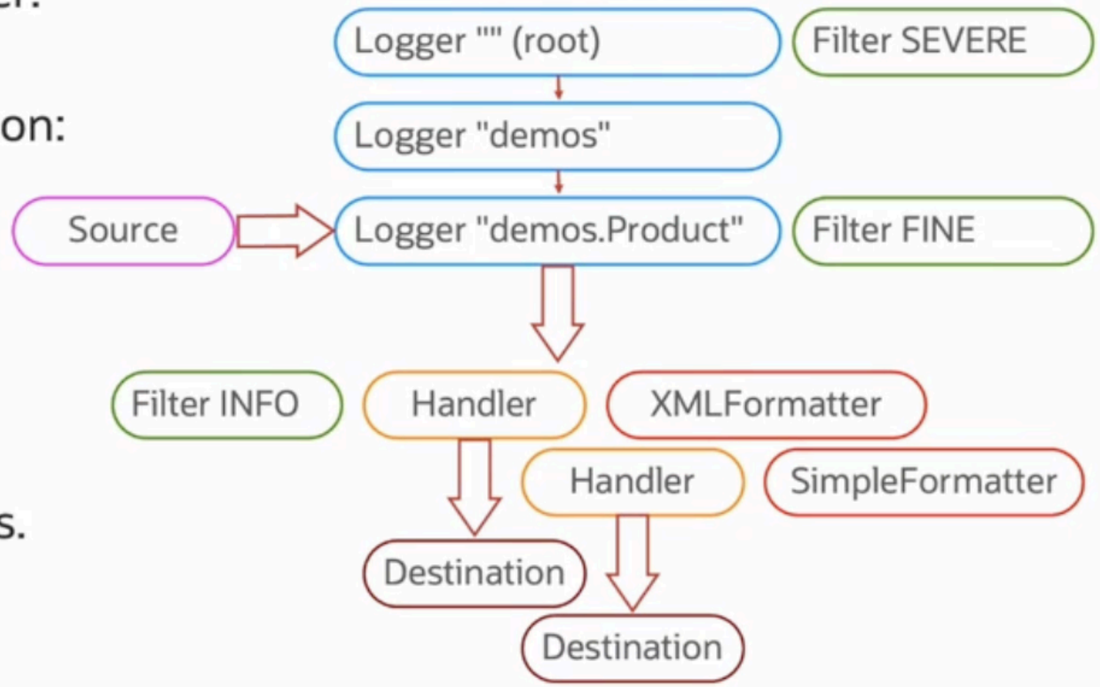
1. Logging level can be set programmatically or via the configuration.
2. Message is concatenated, but is not recorded because it is below the logging-level threshold.
3. Message is not processed if it is below the logging-level threshold.
4. Object parameters can be used to avoid concatenating messages unnecessarily.

```
1  logger.setLevel(Level.INFO);

2  logger.log(Level.FINE, "Product "+id+" has been selected");

3  if(logger.isLoggable(Level.FINER)) {
       logger.log(Level.FINE, "Product "+id+" has been selected");
   }

4  logger.log(Level.FINE, "Product {0} has been selected", id);
```

Log Writing Handling

Log records can be discarded by one or more filters that may be attached to a logger of a log handler.

- Logger writes log messages with different log levels.
- Loggers form a hierarchy.
  - Child logger inherits log level from parent logger.
  - Child logger can override the log level.
- Log handler writes log messages to a log destination:
  - Console
  - File
  - Memory
  - Socket
  - Stream
- Filters can be set for both loggers and log handlers.
- Handlers use formatters to format the record.
  - SimpleFormatter
  - XMLFormatter



Logging Configuration

Logging can be configured using `logging.properties`:

```
handlers=java.util.logging.ConsoleHandler
demos.handlers=java.util.logging.FileHandler

.level=INFO
demos.level=FINE

java.util.logging.ConsoleHandler.formatter=java.util.logging.SimpleFormatter

java.util.logging.FileHandler.pattern=%h/java%u.log
java.util.logging.FileHandler.limit=50000
java.util.logging.FileHandler.count=1
java.util.logging.FileHandler.formatter=java.util.logging.XMLFormatter
```

Manejo de Excepciones

Describe Java Exceptions

Exception is an unexpected event that occurs within a program.

Exceptions interrupt normal execution flow.

All exceptions descend from the class `Throwable`.

Types of Java Exceptions:

- Checked Exceptions
  - Must be caught or
  - Must be explicitly propagated
- Unchecked (Runtime) Exceptions
  - May be caught and
  - Do not have to be explicitly propagated

Notes:

- Unchecked exceptions are often an evidence of a bug in your code, which should be fixed rather than caught.
- Checked exceptions usually represent genuine problems that a normal functioning program may encounter and thus must be caught.

Exception type examples:

```
java.lang.Throwable
  java.lang.Error
    java.lang.AssertionError
    java.lang.VirtualMachineError
      java.lang.OutOfMemoryError
  java.lang.Exception
    java.sql.SQLException
    java.io.IOException
      java.io.FileNotFoundException
      java.nio.file.FileSystemException
        java.nio.file.NoSuchFileException
    java.lang.RuntimeException
      java.lang.NullPointerException
      java.lang.ArithmeticException
      java.lang.IndexOutOfBoundsException
        java.lang.StringIndexOutOfBoundsException
        java.lang.ArrayIndexOutOfBoundsException
      java.lang.IllegalArgumentException
        java.lang.NumberFormatException
        java.util.IllegalFormatException
```

Create Custom Exceptions

Custom exception class characteristics:

- Must extend class `Exception` or another more specific descendant of `Throwable`
- May provide constructors that utilize superclass constructor abilities such as:
  - Provide an error message
  - Wrap another exception indicating a cause of this exception

```
public class ProductException extends Exception {
  public ProductException() {
    super();
  }
  public ProductException(String message) {
    super(message);
  }
  public ProductException(String message, Throwable cause) {
    super(message, cause);
  }
}
```

- **Note:** Ability to wrap one exception inside another is often used when you want to catch one exception and throw another, but not lose information about the original cause of the error.

Throwing Exceptions

To produce an exception:

- Instantiate exception of the required type using any of the available constructors
- Use throw operator to interrupt the flow and trigger the exception propagation process

When an exception is raised:

- Normal program flow is terminated
- Control is passed to the nearest available exception handler (covered next)

If exception handler is not available within this method:

- Unchecked exceptions are automatically propagated to the invoker
- Checked exceptions must be explicitly listed within the `throws` clause

```
public void doThings() throws IOException, CustomException {
  /* actions that may produce one of the following exceptions:
  throw new IOException();
  throw new CustomException();
  throw new NullPointerException();
  or any other runtime exceptions that do not require
  to be explicitly declared by the throws clause */
}
```

- **Note:** It is technically sufficient to declare that a method throws just a generic exception in order to propagate any checked exceptions. However, this is not a good practice, because it obscures which specific exceptions this method can produce.

Catching Exceptions

To catch an exception:

- Surround code that can produce exceptions with the `try` block
- Place one or more `catch` blocks or `finally` block or both after that `try` block
- Specific exception handlers (catching exception subtypes) must be placed before generic handlers
- Unchecked (runtime exception) handlers are optional
- When exception occurs within the try block, program flow is interrupted, and the control is passed to the nearest catch that matches the exception type

```
try {
  doThings();
}catch(NullPointerException |
       ArithmeticException e){
  /* exception handler actions */
}catch(NoSuchFileException e){
  /* exception handler actions */
}catch(IOException e){
  /* exception handler actions */
}catch(Exception e){
  /* exception handler actions */
}finally{
  /* actions that will be executed
     regardless if exceptions occur or not */
}
```

Notes:

- Unrelated exceptions can be handled by the same catch block.
- It is technically sufficient to provide a single generic exception handler. However, this is not a good practice, because you would struggle to distinguish specific reasons for why the program failed.

Exceptions and the Execution Flow

When a normal execution path encounters a runtime exception:

- Starting from the current statement the program flow is interrupted
- Control is passed to the closest matching exception handler, or if none is available, the program will exit

```
1  public class Test {
2    public static void main(String[] args) {
3      int x = 5;
4      int y = 0;
5      int z = divide(x,y);
6      System.out.println(z);
7    }
8    public static int divide(int x, int y) {
9      int z = x/y;
10     return z;
11   }
12 }
```

In the example:

- Exception has not been intercepted
- Method that caused the exception was interrupted, so was the invoking method
- Program has exited
- The information about the exception and the stack trace is printed to the console
- Exception stack trace showed the path of the exception propagation

```
java Test
Exception in thread "main" java.lang.ArithmeticException: / by zero
     at Test.divide(Test.java:9)
     at Test.main(Test.java:5)
```

Helpful `NullPointerExceptions`

- Since version 14, Java `NullPointerException` produces extra error message information that helps to troubleshoot the issue.
- Option is on by default, but can be turned off for security reasons.

```
1 public class Test {
2  private static String value;
3  public static void main(String[] args) {
4    System.out.println(value.indexOf("a"));
5  }
6 }
```

```
java Test
Exception in thread "main" java.lang.NullPointerException: Cannot
invoke "String.indexOf(String)" because "Test.value" is null
     at Test.main(Test.java:4)
```

NEW

```
java Test
Exception in thread "main" java.lang.NullPointerException
     at Test.main(Test.java:4)
```

OLD

Example Throwing an Unchecked Exception

To produce the exception:

- Create an instance of the relevant type of the exception
- Use `throw` operator to interrupt the flow and trigger the exception propagation process

```
public class Test {
  public static void main(String[] args) {
    int x = 5;
    int y = 0;
    int z = divide(x,y);
    System.out.println(z);
  }
  public static int divide(int x, int y) {
    if (y == 0) {
      throw new ArithmeticException("Error: "+x+"/"+y);
    }
    int z = x/y;
    return z;
  }
}
```

```
java Test
Exception in thread "main" java.lang.ArithmeticException: Error: 5/0
     at Test.divide(Test.java:10)
     at Test.main(Test.java:5)
```

Notes:

- You may (but don't have to) provide a handler for catching unchecked exceptions.
- Unchecked exceptions often indicate a bug in a code that should be fixed, not caught.

Example Throwing a Checked Exception

Checked exceptions must be explicitly caught or propagated.

- To catch an exception:
  - Surround code that can produce exceptions with the `try` block
  - Place one or more `catch` blocks or `finally` block or both after that `try` block
- To propagate checked exceptions to the invoker, add the `throws` clause to the method definition listing exception that this method may potentially produce.

```
public static void main(String[] args) {
  try {
    openFile(null);
  }catch(IOException e) {
    e.printStackTrace();
  }
}
public static void openFile(String fileName) throws IOException {
  if (name == null) {
    throw new NoSuchFileException("Filename must be set");
  }
}
```

Notes:

- You may catch unchecked exceptions.
- You must catch checked exceptions

(unless you listed them in a `throws` clause to be propagated further).

Handling Exceptions

Exception-handling structures:

- `try` block contains logic that may throw exceptions.
  - Exceptions within the `try` block interrupt the rest of the block.
  - Transfer control to the nearest matching `catch` block.
- `catch` blocks contain exception-handling logic:
  - Writing logs
  - Throwing other exceptions
  - Terminating the rest of the method
  - Any other corrective actions
- `finally` block
  - Is executed no matter if errors occur or not within the `try` block
  - Performs resource cleanup and closure
- Further, `try` blocks may be embedded inside `catch` or `finally` blocks.

```
BufferedReader in = null;
try {
  in = new BufferedReader(new FileReader("some.txt"));
  String text = in.readLine();
} catch (FileNotFoundException ex) {
  logger.log(Level.SEVERE, "Error opening file", ex);
  return;
} catch (IOException ex) {
  logger.log(Level.SEVERE, "Error reading file", ex);
  throw new CustomException("Failed to read text",ex);
} finally {
  try {
    in.close();
  } catch (IOException ex) {
    logger.log(Level.SEVERE, "Error closing file", ex);
  }
}
```

Resource Auto-Closure

Try-with-parameters syntax provides auto-closure of resource.

- Classes that implement the `AutoCloseable` interface can be instantiated using the try-with-parameters syntax.
- Multiple resources may be initialized inside the try-with-parameters construct.
- Automatic closure of such resources is provided by an implicitly formed `finally` block.

```
try (/*initialise autocloseable resources*/) {
  /* use resources */
} catch (Exception e) {
  /* handle exceptions */
}
/* finally block invoking close method on every
autocloseable resource is formed implicitly */
```

```
try (BufferedReader in = new BufferedReader(new FileReader("some.txt"));
     PrintWriter out = new PrintWriter(new FileWriter("other.txt"))) {
  String text = in.readLine();
  out.println(text);
} catch (FileNotFoundException ex) {
  logger.log(Level.SEVERE, "Opening file error", ex);
} catch (IOException ex) {
  logger.log(Level.SEVERE, "Read-write error", ex);
}
```

Debugging

Java Debugger

- Debugger is used to find and fix bugs in Java platform programs.
- Command-line debugger tool `jdb` is provided with JDK.
- Java IDEs provide visual debug capabilities:
  - Toggle breakpoints in your source code

```
30  public class Shop {
31
32      public static void main(String[] args) {
33          ProductManager pm = new ProductManager(Locale.UK);
34          pm.createProduct( id: 101,  name: "Tea", BigDecimal.valueOf(1.99),
35                  Rating.NOT_RATED);
36          pm.printProductReport( id: 101);
37          pm.reviewProduct( id: 101, Rating.FOUR_STAR,  comments: "Nice hot cup of tea");
38          pm.reviewProduct( id: 101, Rating.TWO_STAR,  comments: "Rather weak tea");
39          pm.reviewProduct( id: 101, Rating.FOUR_STAR,  comments: "Fine tea");
40          pm.reviewProduct( id: 101, Rating.FOUR_STAR,  comments: "Good tea");
```

  - Launch program in debug mode (from the main toolbar, or right click on a Java class)

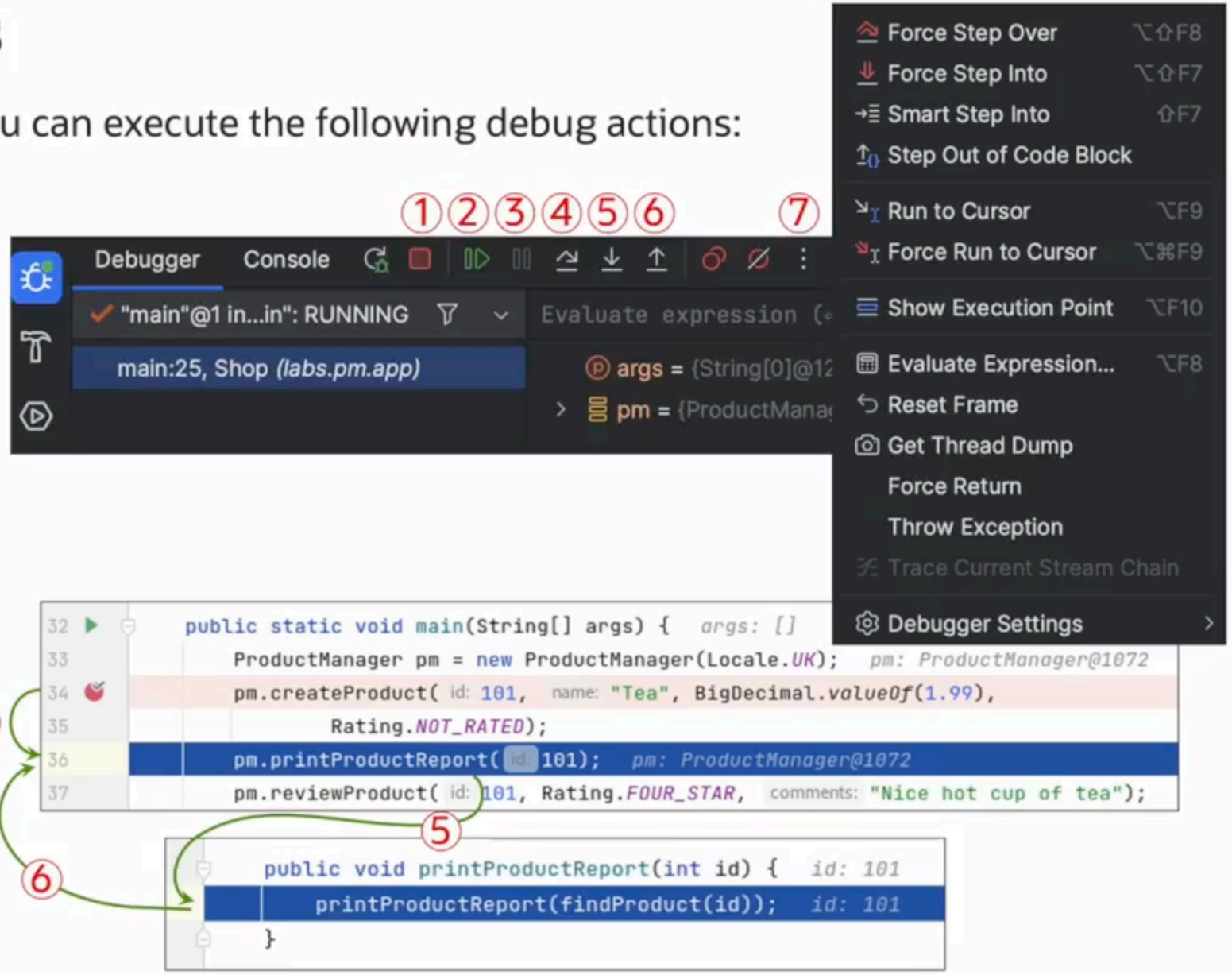
Notes:

- Debugger actions are covered next.
- JDB tool documentation is available at `https://docs.oracle.com/en/java/javase/21/docs/specs/man/jdb.html`

Debugger Actions

Once debugger is running, you can execute the following debug actions:

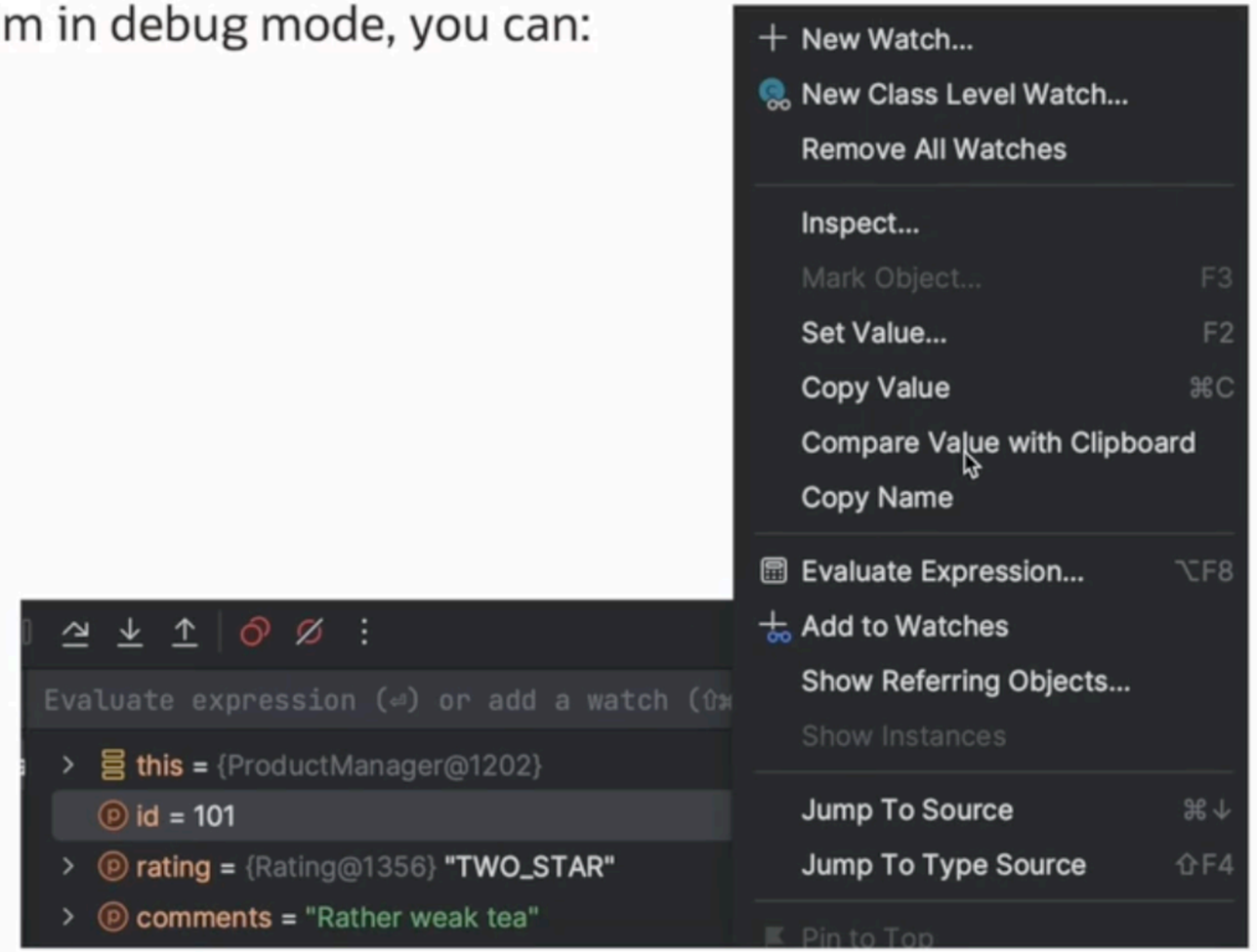
1. Stop debugger session
2. Resume
3. Pause
4. Step over
5. Step into
6. Step out
7. More actions



Manipulate Program Data in Debug Mode

When your IDE is executing program in debug mode, you can:

- Study all the available variables
- Evaluate expressions
- Modify the values
- Watch the specific variables



Validate Program Logic Using Assertions

Assertions can be used to verify the application is executing as expected.

- Assertions test for failure of various conditions.

```
assert <boolean expression>;
assert <boolean expression>: <error text expression>;
```

- When assertion expression is `false`, application terminates and assertion error information is printed.

```
Set<String> values = new HashSet();
String value = "acme";
boolean existingValue = values.add(value);
assert existingValue: "Value "+value+" already exists in the set";
```

- Assertions are disabled by default:
  - Never assume they'll be executed (not used in production code).
  - Do not use assertion to validate parameters or user input.
  - Do not create assertions that cause any state changes or other side effects in the program flow.
- To enable assertions, use `-ea` or `-enableassertions` command-line option:

```
java -ea <package>.<MainClass>
```

Normal Program Flow with No Exceptions

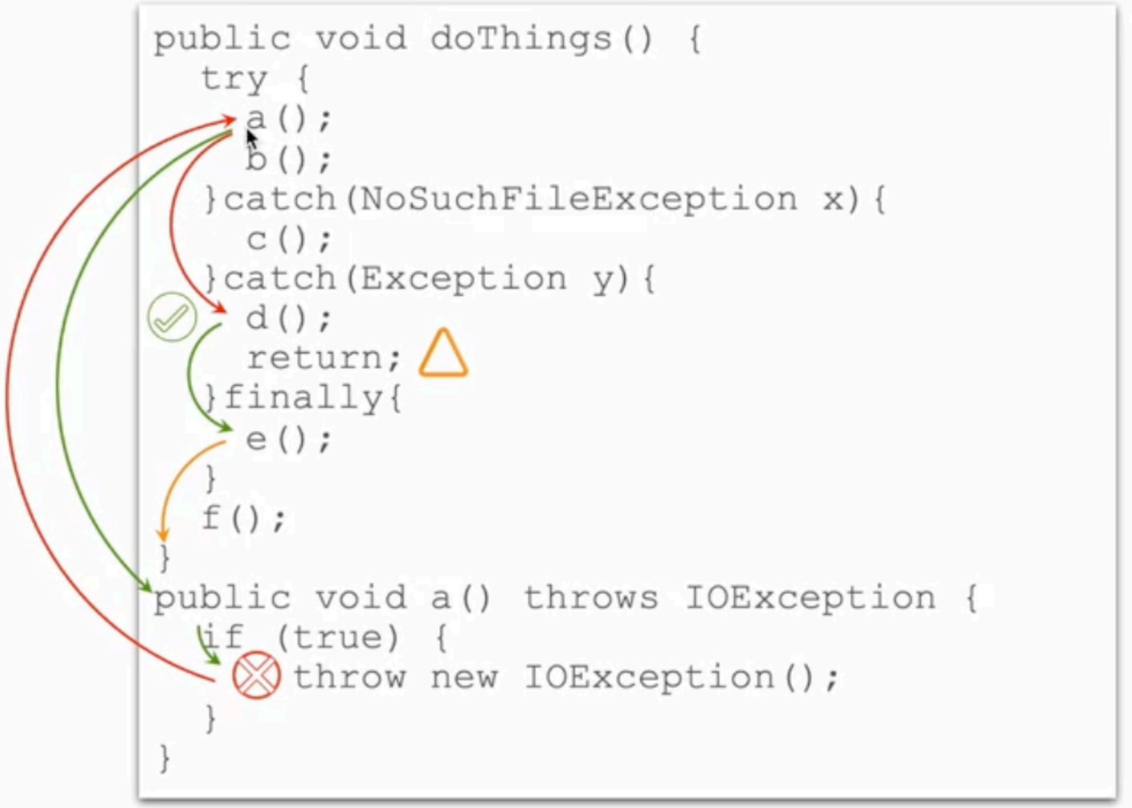
Which operations are going to be invoked in this scenario?

```
public void doThings() {
  try {
    a();
    b();
  }catch(NoSuchFileException x){
    c();
  }catch(IOException y){
    d();
  }finally{
    e();
  }
  f();
}
public void a() throws IOException {
  if (false) {
      throw new IOException();
  }
}
```

Program Flow Catching Any Exceptions

When `IOException` (or any other) is produced:

- Execution path invokes methods `a`, `d`, and `e`
- `Exception` catch block is triggered
- `Finally` block is executed
- Although exception was intercepted, the normal program flow does not resume, because the exception handler has terminated the method



- A generic exception handler would catch any checked or unchecked exceptions.
- Exception handler may terminate the rest of the method using `return` statement or even terminate Java run time using `System.exit(0)`.

Summary

In this lesson, you should have learned how to:

- Use Java Logging API
- Describe exception and error types
- Create custom exceptions
- Introduce `try/catch/finally` syntax
- Throw exception and pass exceptions to invokers
- Introduce `try` with parameters
- Use debugger
- Test code with assertions