




Optional Types

▼ Type	 Lecture
📅 Date	@February 21, 2022
☰ Lecture #	1
🔗 Lecture URL	https://youtu.be/3Q7Rp6sxE_M
🔗 Notion URL	https://21f1003586.notion.site/Optional-Types-45386c4ede174512955ea513e3c21c70
# Week #	9

Dealing with empty streams

- Largest and smallest values seen
 - `max()` and `min()`
 - Requires a comparison function
 - What happens if the stream is empty?

```
Optional<Double> maxrand =  
    Stream.generate(Math::random)
```

```
.limit(100)
.filter(n -> n < 0.001)
.max(Double::compareTo);
```

- `max()` of empty stream is undefined
 - Return value could be `Double` or `null`
- `Optional<T>` object
 - Wrapper
 - May contain an object of type `T`
 - Value is present
 - Or no object

Handling missing optional values

- Or `orElse()` to pass a default value (if the desired value is not present)

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
    .limit(100)
    .filter(n -> n < 0.001)
    .max(Double::compareTo);

Double fixrand = maxrand.orElse(-1.0);
```

- Use `orElseGet()` to call a function which generates replacement for a missing value

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
    .limit(100)
    .filter(n -> n < 0.001)
    .max(Double::compareTo);

Double fixrand = maxrand.orElseGet(
  () -> SomeFunctionToGenerateDouble
);
```

- Use `orElseThrow()` to generate an exception when a missing value is encountered

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
    .limit(100)
    .filter(n -> n < 0.001)
    .max(Double::compareTo);

Double fixrand = maxrand.orElseThrow(
  IllegalStateException::new
);
```

Ignoring missing values

- Use `ifPresent()` to test if a value is present, and process it
 - Missing value is ignored

```
optionalValue.ifPresent(v -> Process v);
```

- For instance, add `maxrand` to a collection `results`, if it is present

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
    .limit(100)
    .filter(n -> n < 0.001)
    .max(Double::compareTo);

var results = new ArrayList<Double>();
maxrand.ifPresent(v -> results.add(v));
```

- Or, we can pass the function in different forms as well

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
    .limit(100)
    .filter(n -> n < 0.001)
    .max(Double::compareTo);

var results = new ArrayList<Double>();
maxrand.ifPresent(results::add);
```

- Specify an alternative action if the value is not present

```
Optional<Double> maxrand =
  Stream.generate(Math::random)
    .limit(100)
```

```

        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.isPresentOrElse(
    v -> results.add(v),
    () -> System.out.println("No max")
);

```

Creating an optional value

- Create an optional value
 - `Optional.of(v)` creates value `v`
 - `Optional.empty` creates empty optional

```

public static Optional<Double> inverse(Double x) {
    if(x == 0) {
        return Optional.empty();
    } else {
        return Optional.of(1 / x);
    }
}

```

- Use `ofNullable()` to transform `null` automatically into an empty optional
 - Useful when working with functions that return object of type `T` or `null`, rather than `Optional<T>`
 - Example, the above code will produce a result if `x` is not `0` otherwise, it will return empty

```

public static Optional<Double> inverse(Double x) {
    return Optional.ofNullable(1 / x);
}

```

Passing on optional values

- Can produce an output `Optional` value from an input `Optional`
- `map` applies function to value, if present
 - If input is empty, so is output

```
Optional<Double> maxrand =
    Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Optional<Double> maxrandasqr =
    maxrand.map(v -> v * v);
```

- Another example

```
Optional<Double> maxrand =
    Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

var results = new ArrayList<Double>();

maxrand.map(results::add);
```

- Supply an alternative for a missing value
 - If value is present, it is passed as is
 - If value is empty, value generated by `or()` is passed

```
Optional<Double> maxrand =
    Stream.generate(Math::random)
        .limit(100)
        .filter(n -> n < 0.001)
        .max(Double::compareTo);

Optional<Double> fixrand =
    maxrand.or(() -> Optional.of(-1.0));
```

Composing optional values of different types

- Suppose that
 - `f()` returns `Optional<T>`
 - Class `T` defines `g()`, returning `Optional<U>`
- Cannot compose `s.f().g()`
 - `s.f()` has type `Optional<T>`, not `T`

- Instead, use `flatMap`
 - `s.f().flatMap(T::g)`
 - If `s.f()` is present, apply `g()`
 - Otherwise return empty `Optional<U>`

```
Optional<U> result = s.f().flatMap(T::g);
```

- For example, pass output of earlier safe `inverse()` to safe `SquareRoot()`

```
public static Optional<Double> inverse(Double x) {
    if(x == 0) {
        return Optional.empty();
    } else {
        return Optional.of(1/x);
    }
}

public static Optional<Double> squareRoot(Double x) {
    if(x < 0) {
        return Optional.empty();
    } else {
        return Optional.of(Math.sqrt(x));
    }
}

Optional<Double> result = inverse(x).flatMap(MyClass::squareRoot);
```

Turning an optional into a stream

- Suppose `lookup(u)` returns a `User` if `u` is a valid username

```
Optional<User> lookup(String id) { ... }
```

- We want to convert a stream of userids into a stream of users
 - Input is `Stream<String>`
 - Output is `Stream<User>`
 - But `lookup` returns `Optional<User>`
- Pass through a `flatMap`

```
Stream<String> ids = ...;
Stream<User> users =
  ids.map(Users::lookup)
    .flatMap(Optional::stream);
```

- What if `lookup` was implemented without using `Optional`?
 - `oldLookup` returns `User` or `null`
 - Use `ofNullable` to regenerate `Optional<User>`


```
Stream<String> ids = ...;
Stream<User> users = ids.flatMap(
  id -> Stream.ofNullable(
    Users.oldLookup(id)
  )
);
```

Summary

- `Optional<T>` is a clean way to encapsulate a value that may be absent
- Different ways to process values of type `Optional<T>`
 - Replace the missing value by a default
 - Ignore missing values
- Can create values of type `Optional<T>` where outcome may be undefined
- Can write functions that transform optional values to optional values
- `flatMap` allows us to cascade functions with optional types
 - Use `flatMap` to regenerate a stream from optional values



Collecting results from Streams

▼ Type	 Lecture
📅 Date	@February 21, 2022
☰ Lecture #	2
🔗 Lecture URL	https://youtu.be/v0fNOGui3Wg
🔗 Notion URL	https://21f1003586.notion.site/Collecting-results-from-Streams-65c4b17d4cf44d7b83c48c486776e3a0
# Week #	9

Collecting values from a stream

- Convert collections into sequences of values — streams
- Process a stream as a collection?
- `Stream` defines a standard iterator, use to loop through values in a stream
- Alternatively, use `forEach` with a suitable function
- Can convert a stream into an array using `toArray()`

- Creates an array of `Object` by default
- Pass array constructor to get a more specific array type

```
mystream.forEach(System.out::println);

Object[] result = mystream.toArray();

String[] result = mystream.toArray(String[]::new);
// mystream.toArray() has the type Object[]
```

Storing a stream as a collection

- What if we want to convert the stream back into a collection?
- Use `collect()`
 - Pass appropriate *factory method* from `Collectors`
 - Static method that directly calls a constructor
 - Implicitly creates an object
- Create a list from a stream

```
List<String> result = mystream.collect(Collectors.toList());
```

- ... or a set

```
Set<String> result = mystream.collect(Collectors.toSet());
```

- To create a concrete collection, provide a constructor

```
TreeSet<String> result = stream.collect(
    Collectors.toCollection(TreeSet::new)
);
```

Stream summaries

- We saw how to reduce a stream to a single result value — `count()`, `max()`, ...
 - In general, need a stream of numbers
- `Collectors` has methods to aggregate summaries in a single object

- `summarizingInt` works for a stream of integers
- Pass function to convert given stream to numbers — here `String::length`
- Returns `IntSummaryStatistics` that stores count, max, min, sum, average

```
IntSummaryStatistics summary =
    mystream.collect(
        Collectors.summarizingInt(String::length);
    );

double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();
```

- Methods to access relevant statistics

- `getCount()`
- `getMax()`
- `getMin()`
- `getSum()`
- `getAverage()`

- Similarly, `summarizingLong()` and `summarizingDouble()` return `LongSummaryStatistics` and `DoubleSummaryStatistics`

Converting a stream to a map

- Convert a stream of `Person` to a map
 - For `Person p`, `p.getID()` is the key and `p.getName()` is value

```
Stream<Person> people = ...;
Map<Integer, String> idToName =
    people.collect(
        Collectors.toMap(
            Person::getId,
            Person::getName
        )
    );
```

- To store entire object as value, use `Function.identity()`

```
Stream<Person> people = ...;
Map<Integer, Person> idToPerson =
```

```
people.collect(
    Collectors.toMap(
        Person::getId,
        Function.identity()
    )
);
```

- What happens if we use name for key and id for value?
 - Likely to have duplicate keys — `IllegalStateException`

```
Stream<Person> people = ...;
Map<String, Integer> nameToID =
    people.collect(
        Collectors.toMap(
            Person::getName,
            Person::getId
        )
    );
```

- Provide a function to fix such problems

```
Stream<Person> people = ...;
Map<String, Integer> nameToID =
    people.collect(
        Collectors.toMap(
            Person::getName,
            Person::getId,
            (existingValue, newValue) -> existingValue
        )
    );
```

Grouping and partitioning values

- Instead of discarding values with duplicate keys, group them
- Collect all ids with the same name in a list
- Instead, may want to partition the stream using a predicate

```
Stream<Person> people = ...;
Map<String, List<Person>> nameToPersons =
    people.collect(
        Collectors.groupingBy(
            Person::getName
        )
    );
```

- Partition names into those that start with `A` and the rest
 - Key values of resulting map are `true` and `false`

```
Stream<Person> people = ...;
Map<String, List<Person>> aAndOtherPersons =
    people.collect(
        Collectors.partitioningBy(
            p -> p.getName().substr(0,1).equals("A")
        )
    );


List<Person> startingLetterA = aAndOtherPersons.get(true);
```

Summary

- We converted collections into sequences and processed them as streams
- After transformations, we may want to process a stream as a collection
- Use iterators, `forEach()` to process a stream element by element
- Use `toArray()` to convert to an array
- Factory methods in `Collector` allows us to convert a stream back into a collection of our choice
- Can convert an arbitrary stream into a stream of numbers and collect summary statistics
- Can convert a stream into a map
- Can group values by a key, or partition by a predicate



Input/Output Streams

▼ Type	 Lecture
📅 Date	@February 21, 2022
☰ Lecture #	3
🔗 Lecture URL	https://youtu.be/FFdDai4qf8w
🔗 Notion URL	https://21f1003586.notion.site/Input-Output-Streams-c35b40f4d8ce4385abf3ec9b190a4e5e
# Week #	9

Input and Output streams

- Input → read a sequence of bytes from some source
 - A file, an internet connection, memory, ...
- Output → write a sequence of bytes to some source
 - A file, an internet connection, memory, ...
- Java refers to these as input and output streams
 - Not the same as stream objects in class `Stream`

- Input and Output values could be of different types
 - Ultimately, input and output are raw uninterpreted bytes of data
 - Interpret as text — different Unicode encodings
 - Or as binary data — integers, floats, doubles, ...
- Use a pipeline of input/output stream transformers
 - Read raw bytes from a file, pass to a stream that reads text
 - Generate binary data, pass to a stream that writes raw bytes to a file

Reading and Writing raw bytes

- Classes `InputStream` and `OutputStream`
- Read one or more bytes — abstract methods are implemented by subclasses of `InputStream`
- Check availability before reading

```
abstract int read();
int read(byte[] b);
byte[] readAllBytes();
// ... and more

InputStream in = ...
int bytesAvailable = in.available();
if(bytesAvailable > 0) {
    var data = new byte[bytesAvailable];
    in.read(data);
}
```

- Write bytes to output
- Close a stream when done — release resources
- Flush an output stream — output is buffered

```
abstract void write(int b);
void write(byte[] b);
// ... and more

OutputStream out = ...
byte[] values = ...;
out.write(values);

in.close();
```

```
out.flush();
```

Connecting a stream to an external source

- Input and output streams ultimately connect to external resources
 - A file, an internet connection memory ...
 - We limit ourselves to files
- Create an input stream attached to a file
- Create an output stream attached to a file
- Overwrite or append?
 - Pass a boolean second argument to the constructor

```
var in = new FileInputStream("input.class");

var out = new FileOutputStream("output.bin");

// Overwrite
var out = new FileOutputStream("newoutput.bin", false);

// Append
var out = new FileOutputStream("sameoutput.bin", true);
```

Reading and Writing text

- Recall `Scanner` class
 - Can apply to any input stream
- Many read methods

```
var fin = new FileInputStream("input.txt");
var scin = new Scanner(fin);

String s = scin.nextLine(); // One line
String w = scin.next();     // One word
int i = scin.nextInt();     // Read an int
boolean b = scin.hasNext(); // Any more words?
```

- To write text, use `PrintWriter` class
 - Apply to any output stream

```
var fout = new FileOutputStream("output.txt");
var pout = new PrintWriter(fout);
```

- Use `println()`, `print()` to write txt

```
String msg = "Hello, World!";
pout.println(msg);
```

- Example: Copy input text file to output text file

```
var in = new Scanner(...);
var out = new PrintWriter(...);

while(in.hasNext()) {
    String line = in.nextLine();
    out.println(line);
}
```

- Beware: input/output methods generate many different kinds of exceptions
 - Need to wrap code with `try` blocks

Reading and Writing binary data

- To read binary data, use `DataInputStream` class
 - Can apply to any input stream
- Many read methods

```
var fin = new FileInputStream("input.class");
var din = new DataInputStream(fin);

readInt, readShort, readLong
readFloat, readDouble
readChar, readUTF
readBoolean
```

- To write binary data, use `DataOutputStream` class
 - Apply to any output stream
- Many write methods

```

var fout = new FileOutputStream("output.bin");
var dout = new DataOutputStream(fout);

writeInt, writeShort, writeLong
writeFloat, writeDouble
writeChar, writeUTF
writeBoolean
writeChars
writeByte

```

- Example: Copy input binary file to output binary file
 - Catch exceptions

```

var in = new DataInputStream(...);
var out = new DataOutputStream(...);

int bytesAvailable = in.available();
while(bytesAvailable > 0) {
    var data = new byte[bytesAvailable];
    in.read(data);
    out.write(data);
    bytesAvailable = in.available();
}

```

Other features

- Buffering an input stream
 - Reads blocks of data
 - More efficient

```

var din = new DataInputStream(
    new BufferedInputStream(
        new FileInputStream("grades.dat")
    )
);

```

- Speculative reads
 - Examine the first element
 - Return to stream if necessary

```

var pbin = new PushbackInputStream(
    new BufferedInputStream(

```

```

        new FileInputStream("grades.dat")
    )
};

int b = pbin.read();
if(b != '<') {
    pbin.unread(b);
}

```

- Streams are specialized
 - `PushBackStream` can only `read()` and `unread()`
 - Feed to a `DataInputStream` to read meaningful data

```

var pbin = new PushbackInputStream(
    new BufferedInputStream(
        new FileInputStream("grades.dat")
    )
);

var din = new DataInputStream(pbin);

```

- Java has a whole zoo of streams for different tasks
 - Random access files, zipped data, ...
- Chain together streams in a pipeline
 - Read binary data from a zipped file

`FileInputStream`

`ZipInputStream`

`DataInputStream`


Summary

- Java's approach to input/output is to separate out concerns
- Chain together different types of input/output streams
 - Connect an external source as input or output
 - Read and Write raw bytes
 - Interpret raw bytes as text
 - Interpret raw bytes as data
 - Buffering, speculative read, random access files, zipped data, ...

- Chaining together streams appears tedious, but adds flexibility



Serialization

▼ Type	 Lecture
📅 Date	@February 22, 2022
☰ Lecture #	4
🔗 Lecture URL	https://youtu.be/oweQzNSHA5I
🔗 Notion URL	https://21f1003586.notion.site/Serialization-ca1d3636d5584ebca8f284baa577fb12
# Week #	9

Reading and Writing Objects

- We can read and write binary data
 - `DataInputStream` , `DataOutputStream`
- Read and write low level units
 - Bytes, integers, floats, characters, ...
- Can we export and import objects directly?
- Why would we want to do this?

- Backup objects onto disk, with state
- Restore objects from the disk
- Send objects across a network
- Serialization and Deserialization

Reading and writing objects ...

- To write objects, Java has another output stream type, `ObjectOutputStream`

```
var out = new ObjectOutputStream(  
    new FileOutputStream("employee.dat")  
);
```

- Use `writeObject()` to write out an object

```
var emp = new Employee(...);  
var boss = new Manager(...);  
out.writeObject(emp);  
out.writeObject(boss);
```

- To read back objects, use `ObjectInputStream`

```
var in = new ObjectInputStream(  
    new FileInputStream("employee.dat")  
);
```

- Retrieve objects in the same order they were written using `readObject()`

```
var e1 = (Employee) in.readObject();  
var e2 = (Employee) in.readObject();
```

- Class has to allow serialization — implement marker interface `Serializable`

```
public class Employee implements Serializable { ... }
```

How serialization works

- `ObjectOutputStream` examines all the fields and saves their contents

- `ObjectInputStream` "reconstructs" the object, effectively calls a constructor
- What happens when many objects share the same object as an instance variable?

```
class Manager extends Employee {
    private Employee secretary;
    ...
}
```

- Two managers have the same secretary
- Each object is assigned a serial number — hence, serialization
 - When first encountered, save the data to output stream
 - If saved previously, record the serial number
- Reverse the process when reading

Customizing serialization

- Some objects should not be serialized — value of file handles, ...
- Mark such fields as `transient`

```
public class LabeledPoint implements Serializable {
    private String label;
    private transient Point2D.Double point;
    ...
}
```

- Can override `writeObject()`
 - `defaultWriteObject()` writes out the object with all non-transient fields
 - Then explicitly write relevant details of transient fields

```
private void writeObject(ObjectOutputStream out)
    throws IOException {
    out.defaultWriteObject();
    out.writeDouble(point.getX());
    out.writeDouble(point.getY());
}
```

- ... and `readObject()`

- `defaultReadObject()` reconstructs object with all non-transient fields
- Then explicitly reconstruct transient fields

```
private void readObject(ObjectInputStream in)
    throws IOException {
    in.defaultReadObject();
    double x = in.readDouble();
    double y = in.readDouble();
    point = new Point2D.Double(x, y);
}
```

Handle with care

- Serialization is a good option to share data within an application
- Over time, older serialized objects may be incompatible with newer versions
 - Some mechanisms for version control, but still some pitfalls possible
- Deserialization implicitly invokes a constructor
 - Running code from an external source
 - Always a security risk

Summary

- Serialization allows us to export and import objects, with state
 - Backup objects onto disk, with state
 - Restore objects from disk
 - Send objects across a network
- Use `ObjectOutputStream` and `ObjectInputStream` to write and read objects
- Serial numbers are used to ensure only a single copy of each shared object is archived
- Mark fields that should not be serialized as `transient`
 - Customize `writeObject()` and `readObject()`
- Serialization carries risks of ...
 - Version control of objects
 - Running unknown code