
MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Paul Corazza
1

Lecture 7: Interfaces in Java 8 and the Object Superclass

Wholeness Statement

Java supports inheritance between classes in support of the OO concepts of inherited types and polymorphism. Interfaces support encapsulation, play a role similar to abstract classes, and provide a safe alternative to multiple inheritance. Likewise, relationships of any kind that are grounded on the deeper values at the source of the individuals involved result in fuller creativity of expression with fewer mistakes.

Outline

- Java 8 interfaces: Introduction
- Java 8 interfaces: Two Applications of Default Method
- Java 8 interfaces and the Diamond Problem
- FPP Review: Overriding Methods in the Object Class

Java 8 Features of Interfaces

- Before Java 8, none of the methods in an interface had a method body; all were unimplemented.

- In Java 8, two kinds of implemented methods are now allowed: *default methods* and *static methods*. Both can be added to legacy interfaces without breaking code.

- A default method is a fully implemented method within an interface, whose declaration begins with the keyword `default`
- A static method in an interface is a fully implemented static method having the same characteristics as any static method in a class.

See Demos in package lesson7.lecture.defaultmethods and lesson7.lecture.interfacestatic

New Programming Style

Default Methods in an interface eliminate the need to create special classes that represent a default implementation of the interface.

- Examples from pre-Java 8 of default implementations of interfaces:
  WindowListener / WindowAdapter (in the AWT),
  List / AbstractList. [See JavaLibrary project in workspace]
- Now, in developing new code, it is possible in many cases to place these default implementations in the interface directly.

Static Methods in an interface eliminate the need to create special utility classes that naturally belong with the interface.

- Examples from pre-Java 8 of how interfaces sometimes have companion utility classes (consisting of static methods):
  Collection / Collections [See JavaLibrary project]
  Path / Paths.
- For new code, it is now possible to place this static companion code directly in the interface.

Solution to Evolving API Problem

When you need to add new methods to an existing interface, provide them with default implementations using the new Java 8 default feature. Then

- legacy code will not be required to implement the new methods, so existing code will not be broken
- new functionality will be available for new client projects.

Exercise 7.1 – Rewrite List Interface

Explore the package exercise7_1 in the InClass Exercises project. You will see a class MyStringList along with an interface StringList that it implements. StringList contains several common list operations:

```
String[] strArray();  int size();  void setSize(int s);
        void add(String s);  String get(int i);
```

Show how to use Java 8 `default` methods to provide implementations of `add` and `get`. This considerably reduces the effort to implement StringList in MyStringList since most of the implementation work has been moved into the interface.

NOTE: Something like this could have been done in Java's List interface (moving most of the implementations from `AbstractList` into default methods of `List`), except that the `List` interface was created long before default interface methods had been introduced.

Outline

- ✓ Java 8 interfaces: Introduction
- ❑ Java 8 interfaces: Two Applications of Default Methods
- ❑ Java 8 interfaces and the Diamond Problem
- ❑ Review: Overriding Methods in the Object Class

Two Applications of Default Methods

First Set of Examples:

`enum`s can now "inherit" from another type

Second Set of Examples:

`forEach` – default method in `Iterable`

First Set of Examples: Review of Enums

- An *enumerated type* is a Java class all of whose possible instances are explicitly enumerated during initialization.
- Example:

```
public enum Size { SMALL, MEDIUM, LARGE};
//usage:
if(requestedSize==Size.LARGE)
        applyDiscount();
```

- The `enum Size` (which is a special kind of Java class) has been declared to have just three instances, named `SMALL, MEDIUM, LARGE`.

Review of Enums (cont)

Two important applications for enums:

1. Using enums as *constants* in an application
   - *Weak Programming Practice*: Create a class (or interface) containing constants, stored as public static final values – most often arising when constants are ints or Strings
   - *Problem.* No compiler control over usage of these constants when they occur as input arguments to methods (example on next slide)
   - *Better Approach* Represent constants as instances of an enumerated type.
2. Optimal, threadsafe implementation of the Singleton Pattern

Example of Handling Constants in Java

In the java.awt package there is a class Label, used to represent a label in a UI (built using the old AWT). It makes use of constants to designate alignment properties: LEFT, CENTER, RIGHT. This use of constants is flawed, but it is a commonly used style

```java
public class AlignmentConstants {
    /**
     * Indicates that the label should be left justified.
     */
    public static final int LEFT        = 0;

    /**
     * Indicates that the label should be centered.
     */
    public static final int CENTER      = 1;

    /**
     * Indicates that the label should be right justified.
     * @since   JDK1.0t.
     */
    public static final int RIGHT       = 2;
}
```

```java
//extracted from java.awt.Label
//Java library does it the bad way
public class Label {
    private String text;
    private int alignment;
    public Label(String text, int alignment) {
        this.text = text;
        setAlignment(alignment);
    }
    public synchronized void setAlignment(int alignment) {
        switch (alignment) {
        case AlignmentConstants.LEFT:
        case AlignmentConstants.CENTER:
        case AlignmentConstants.RIGHT:
            this.alignment = alignment;
            return;
        }
        throw new IllegalArgumentException(
          "improper alignment: " + alignment);
    }
    public String getText() {
        return text;
    }
    public int getAlignment() {
        return alignment;
    }
}
```

Problem: No compiler control over use of these constants. Could make the following call:

```
Label label = new Label("Hello", 23);
```

You won't know till you run the code that "23" is meaningless. The compiler sees that a value of the correct type has been used, but at runtime, 23 will be recognized as an illegal value.

It is better to control the values passed in with the help of the compiler. This is accomplished using an enum to store constants, rather than collecting together a bunch of public static final integers.

Improved Label Using enums

```
public enum Alignment {
    /**
     * Indicates that the label should be left justified.
     */
    LEFT,

    /**
     * Indicates that the label should be centered.
     */
    CENTER,

    /**
     * Indicates that the label should be right justified.
     * @since   JDK1.0t.
     */
    RIGHT;
}
```

```
//Better way, not currently implemented
//in Java libraries
public class Label {
    private String text;
    private Alignment alignment;
    public Label(String text, Alignment alignment) {
        this.text = text;
        setAlignment(alignment);
    }
    public synchronized void setAlignment(Alignment alignment)
        this.alignment = alignment;
    }
    public String getText() {
        return text;
    }
    public Alignment getAlignment() {
        return alignment;
    }
}
```

See the demo: lesson7.lecture.enums.*

Review of Best Practice for Using enums

From Bloch, *Effective Java (2nd edition):*

> *Use enums (in place of public static final variables) whenever you need a fixed set of constants all of whose values you know at compile time.*

Best Practices, continued

- Question: What if you have constants that must be of specific types, like int or String (or another type)?

```
class DimConstants {
    public static final double LENGTH = 1.0;
    public static final double WIDTH = 2.0;
}
class Test {
    public static void main(String[] args) {
        System.out.println(DimConstants.LENGTH);
    }
}
```

- Solution: Use an enum constructor.

```
class Test {
    public static void main(String[] args) {
        System.out.println(Dim.LENGTH.val());
    }
}
```

```
public enum Dim {
    LENGTH(1.0),
    WIDTH(2.0);
    double val;
    Dim(double x) {
        val = x;
    }
    public double val() {
        return val;
    }
}
```

Exercise 7.2

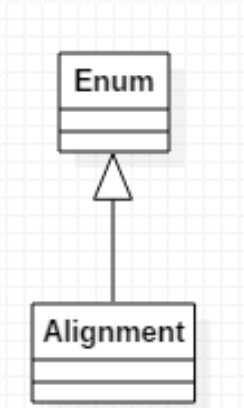
- Below is a `Constants` class consisting of `public final static` variables to provide constants for the rest of the application. Replace with an `enum Const` that provides the same functionality. Refactor the `main` method so that it uses the new `Const` type.

- You can find `Constants` and a test class in the `InClassExercises` project.

```
public class Constants {
    public static final String COMPANY = "Microsoft";
    public static final int SALES_TARGET = 20000000;
}
```

Review of enum Implementation in Java

- In the Label example (earlier slide), each of the instances declared within the `Alignment` enum has type `Alignment`, which is a subclass of `Enum`. Therefore
  - `Alignment` is itself a *class*
  - `Alignment` is not allowed to inherit from any other class (multiple inheritance not allowed).



Using enums to Create Singletons

- A *singleton* class is a class that can have at most one instance
- Easy implementation using an `enum`:

```
enum MySingleton {
      INSTANCE;
      public void behavior() {}
}
//access it like this:
MySingleton.INSTANCE.behavior();
```

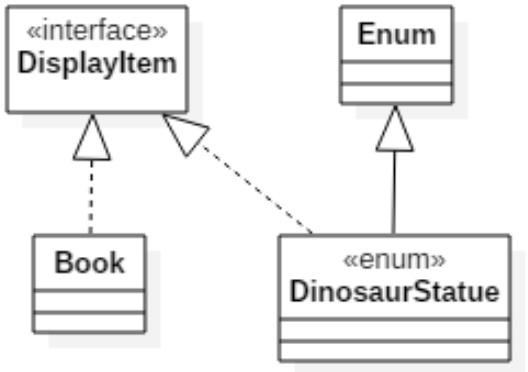
Demo: `lesson7.lecture.singletons`

In Java 8, Enums Can "inherit"

//from lesson7.lecture.enums3.java8

```
interface DisplayItem {
  default String displayInfo() {
                . . .
  }
}
class Book implements DisplayItem {
}

enum DinosaurStatue implements DisplayItem {
  INSTANCE;
}
```



See lesson7.lecture.enums3.java7 and lesson7.lecture.enums3.java8

Second Set of Examples: forEach

- The `Iterable` interface is part of the Collections API that is implemented by all collection classes, and supports iteration through a collection
- The only method in `Iterable` is `iterator()`, which returns an `Iterator`
- `Iterator` has two methods:
  - `hasNext()`
  - `next()`
- When a class (even user-defined) implements the `Iterable` interface, the "for each" construct can be used (and of course, an instance of `Iterator` is available).

See Demo: `lesson7.lecture.iterator`

New (Java 8) in the Iterable interface is a default method:

`forEach`

Sample usage:

```
Consumer<String> consumer = new Consumer<String>() {
    @Override
    public void accept(String s) {
        System.out.println(s);
    }
};
System.out.println("------using new forEach method------");
l.forEach(consumer);
```

Output:

```
------using new forEach method------
Bob
Steve
Susan
Mark
Dave
```

See Demos:
lesson7.lecture.iterator

1. The `forEach` method applies the `Consumer` method `accept` to each element of the list.

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

2. In this example, the `accept` method just prints the value to the console.

3. `Consumer` is an interface introduced in Java 8, with just one abstract method `accept`, which accepts a single argument and produces no return value.

```
interface Consumer<T> {
   void accept(T input);
}
```

Exercise 7.3

- You have a Java `ArrayList` containing multiple elements and an empty MyStringList (from In-Class Exercise 7.1). Use the new Java 8 `forEach` method on the Java list to copy all its elements into the instance of MyStringList.

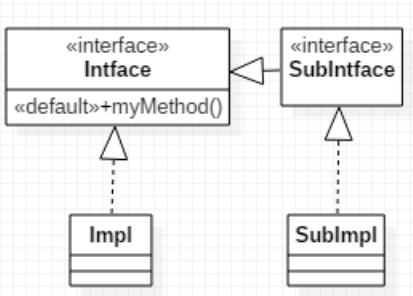
- Startup code is in the exercise_3 package in the InClassExercises project. Test your work by using the `main` method in the ListInfo class.

Outline

- ✓ Java 8 interfaces: Introduction
- ✓ Java 8 interfaces: Two Applications of Default Methods
- ❑ Java 8 interfaces and the Diamond Problem
- ❑ FPP FPP Review: Overriding Methods in the Object Class

Rules for Default Methods in an Interface

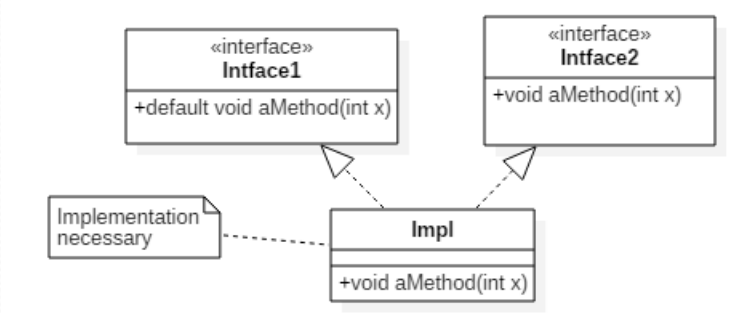
- If a class implements an interface with a default method, that class inherits the default method (or can override it).



- Potential clash if
  - two interfaces have the same method, or
  - one interface and a superclass have the same method

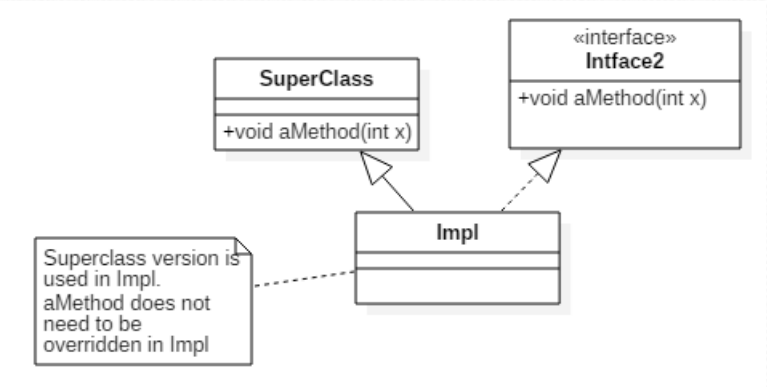
Interface vs Interface – clash! When two interfaces each have a method with the same signature:

- If one of these is a default method, any implementer of both interfaces *must* override the method (or declare it as an abstract method) – can't simply do nothing.
- If one of these is a default method, any *subinterface* of both interfaces must provide a default method (i.e. an implementation) of this method, or declare the method (even if unimplemented).
- Note: Even in Java 7, it is not possible to implement two interfaces each of which has a method with the same signature but different return types.



Superclass vs Interface – superclass wins! When a class extends a superclass and also implements an interface, and both super class and interface have a method with the same signature, the superclass implementation wins – this is the version that is inherited by the class. The subclass/ implementer is not required to override the shared method.

(See Demos in lesson7.lecture.defaultmethodrules)



Static Methods Do Not Clash

- Static methods defined in an interface are *not* inherited by implementers (this differs from the behavior for subclasses of a class)
- Therefore, if two interfaces implement static methods with the same signature, there is no clash to address when a class implements these interfaces.
- Static methods can always be accessed in a static way in such cases, but it is not related to inheritance.

  See demo lesson7.lecture.interfacestatic_clash

Exercise 7.4

Look at the code snippets on the PDF file in lesson7.exercise_4 package of the InClassExercises project. Try to determine, without using a compiler, what happens when the code is compiled/run.

Main Point 1

Interfaces are used in Java to specify publicly available services in the form of method declarations. A class that implements such an interface must make each of the methods operational. Interfaces may be used polymorphically, in the same way as a superclass in an inheritance hierarchy. Because many interfaces can be implemented by the same class, interfaces provide a safe alternative to multiple inheritance. Java8 now supports static and default methods in an interface, which make interfaces even more flexible: For instance, enums can now "inherit" from other types and new public operations can be added to legacy interfaces without breaking code (as was done with the `forEach` method in the `Iterable` interface).

The concept of an interface is analogous to the creation itself – the creation may be viewed as an "interface" to the undifferentiated field of pure consciousness; each object and avenue of activity in the creation serves as a reminder and embodiment of the ultimate reality.

Outline

- ✓ Java 8 interfaces: Introduction
- ✓ Java 8 interfaces: Two Applications of Default Methods
- ✓ Java 8 interfaces and the Diamond Problem
- ❑ FPP Review: Overriding Methods in the Object Class

FPP Review: Overriding Methods in the Object Class

The `Object` class is the superclass of all Java classes, and contains several useful methods

-- in most cases, they are useful *only if* they are overridden.

- `toString`
- `equals`
- `hashCode`

Overriding `toString()`

The purpose of `toString()` is to provide a (readable) `String` representation (which can be logged or printed to the console) of the state of an object.

Example from FPP:

```
// toString for an Account object
public String toString(){
     String ret =
         "Account type: " + acctType +
                "\nCurrent bal:   " + balance;
     return ret;
}
```

Best Practice. For every significant class you create, override the `toString` method.

Overriding `equals()`

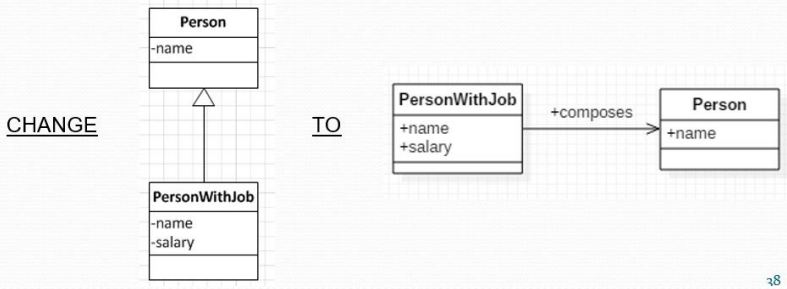
Care is needed in overriding equals when one class inherits from another.

Best Pracices Suppose B is a subclass of A.

1. If it is acceptable for B to use the same equals method as used in A, then the best strategy is the *instanceof strategy* and make equals final. See lesson7.lecture.overrideequals.instanceofstrategy3.

2. If *two different equals methods* are required, two strategies are possible
   A. Use the same classes strategy, but declare subclass B to be final
      See lesson7.lecture.overrideequals.sameclassesstrategy

   B. Use composition instead of inheritance – this will always work as long as the inheritance relationship between B and A is not needed (e.g. for polymorphism). See lesson7.lecture.overrideequals.composition



Overriding `hashCode()`

There are two general rules for creating hash codes:

I. (Primary Hashing Rule) Equal keys must be given the same hash code (otherwise, the same key will occupy different slots in the table)

If `k1.equals(k2) then k1.hashCode() == k2.hashCode()`

II. (Secondary Hashing Guideline) Different keys should be given different hash codes (if not, in the worst case, if every key is given the same hash code, then all keys are sent to the same slot in the table; in this case, hashtable performance degrades dramatically).

Best Practice: The hash codes should be distributed as evenly as possible (this means that one integer occurs as a hash code approximately just as frequently as any other)

Overriding `hashCode()`

Best Practices:

- Whenever equals is overridden, hashCode should also be overridden
- The hashCode method should take into account the same fields as the equals method
- the class on which the object is based should be *immutable*

To define your own hashCode method, use the Objects.hash(…) method.

Example

To override hashCode, we make use of the Java library method `Objects.hash`, which takes any number of arguments; the method creates a hashcode based on the hashcodes of the instance variables of Person

```
public class Person {
    private LocalDate hireDate;
    private String name;
    private int age;
    @Override
    public boolean equals(Object ob) {
        if(ob==null) return false;
        if(!(ob instanceof Person)) return false;
        Person p = (Person)ob;
        return hireDate.equals(p.hireDate)
            && name.equals(p.name)
            && age == p.age;
    }
    @Override
    public int hashCode() {
        return Objects.hash(hireDate, name, age);
    }
}
```

Review: Making Your Classes Immutable

1. A class is immutable if the data it stores cannot be modified once it is initialized. Java's String and number classes (such as Integer, Double, BigInteger) are immutable. Immutable classes provide good building blocks for creating more complex objects. Java 8: LocalDate, as we saw earlier, is also immutable.

2. Immutable classes tend to be smaller and focused (building blocks for more complex behavior). If many instances are needed, a "mutable companion" should also be created (for example, the mutable companion for String is StringBuilder) to handle the multiplicity without hindering performance.

3. Guidelines for creating an immutable class (from *Effective Java*, 2nd ed.)

   - **All fields should be *private* and *final*.** This keeps internals private and prevents data from changing once the object is created.
   - **Provide *getters* but no *setters* for all fields**. Not providing setters is essential for making the class immutable.
   - **Make the class *final*.** (This prevents users of the class from accessing the internals of the class in another way – to be discussed in Lesson 6.)
   - **Make sure that getters do not return mutable objects**.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

1. Inheritance in Java makes it possible for a subclass to enjoy (and re-use) the features of a superclass.

2. All classes in Java – even user defined classes – automatically inherit from the class Object

3. ***Transcendental Consciousness*** is the field of pure awareness, beyond the active thinking level, that is the birthright and essential nature of everyone. Everyone "inherits" from pure consciousness

4. ***Wholeness moving within itself****:* In Unity Consciousness, there is an even deeper realization: The only data and behavior that exist in the universe is that which is "inherited from" pure consciousness – everything in that state is seen as the play of one's own consciousness.


MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Paul Corazza
1

Lecture 9: The Stream API

Solving Problems by Engaging Deeper Values of Intelligence

Wholeness Statement

The stream API is an abstraction of collections that supports aggregate operations like filter and map. These operations make it possible to process collections in a declarative style that supports parallelization, compact and readable code, and processing without side effects. Deeper laws of nature are ultimately responsible for how things appear in the world. Efforts to modify the world from the surface level lead to struggle and only partial success. Affecting the world by accessing the deep underlying laws that structure everything can produce enormous impact with little effort. The key to accessing and winning support from deeper laws is going beyond the surface of awareness to the depths within.

Outline

1. **Introduction to Java 8 Streams**
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. Terminal Operations
7. Working with the Optional class
8. Reduce
9. Collecting Results
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

What Are Streams and Why Are They Used?

1. *What They Are.* A stream is a way of representing data in a collection (and in a few other data structures) which supports functional-style operations to manipulate the data.

   From the API docs: A stream is "a sequence of elements supporting sequential and parallel aggregate operations."

   Streams provide new ways of accessing and extracting data from Collections.

2. *Why They Are Used*. To understand why they are used, consider the following problem:

> *Problem.* Given a list of words (say from a book), count how many of the words have length > 12.

Imperative-style solution

```
int count = 0;
for(String word : list) {
     if(word.length() > 12)
         count++;
}
```

Issues:

i. Relies upon shared variable `count`, so is not threadsafe

ii. Commits to a particular sequence of steps for iteration

iii. Emphasis is on *how* to obtain the result, not *what* is needed

Functional-style solution

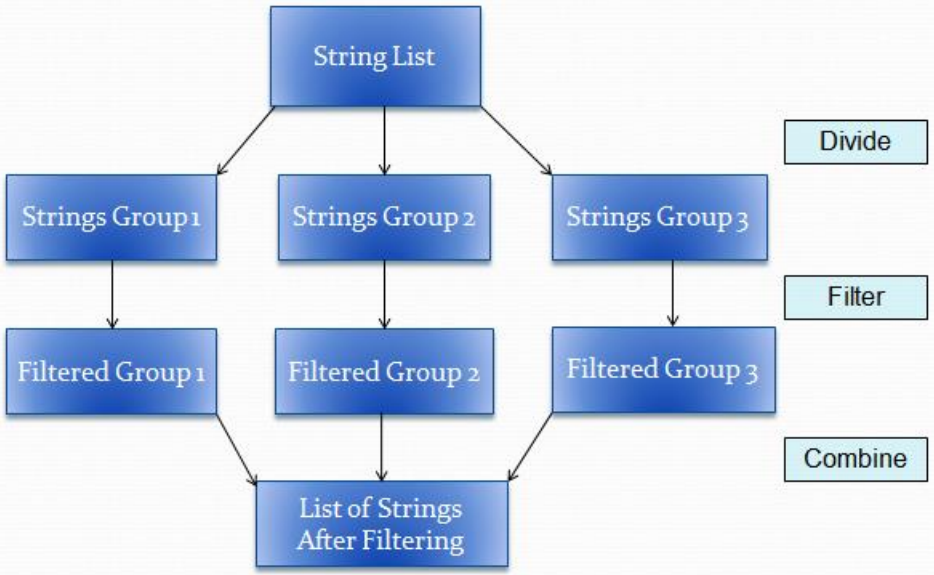
```
final long count
   = words.stream()
          .filter(w -> w.length() > 12)
          .count();
```

Advantages:

i. Purely functional, so threadsafe
ii. Makes no commitment to an iteration path, so more parallelizable
iii. Declarative style – "what, not how"
iv. With Java 8 or later it is easy to transform into a parallel processing solution

Parallel-processing solution

```
final long count
        = words.parallelStream()
                .filter(w -> w.length() > 12)
                .count();
```



Outline

1. Introduction to Java 8 Streams
2. **Streams: Basic Facts and a 3-Step Template**
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. Terminal Operations
7. Working with the Optional class
8. Reduce
9. Collecting Results
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

Facts About Streams

1. *Streams do not store the elements they operate on*. Typically they are stored in an underlying collection, or they may be generated on demand.

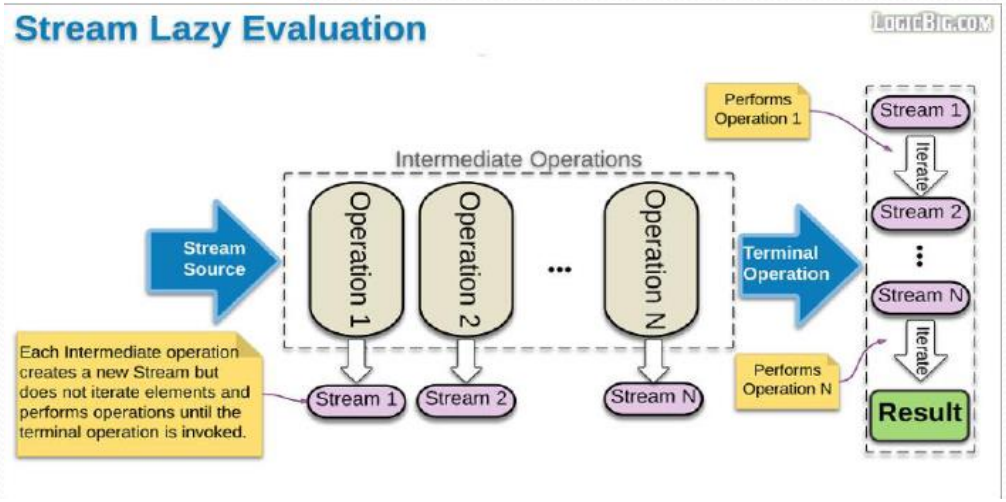
2. *Stream operations do not mutate their source*. Instead, they return new streams that hold the result.

3. *Java Implementation.* The methods on the `Stream` interface are implemented by the class `ReferencePipeline`. The method implementations involve a combination of technical operations internal to the `stream` package.

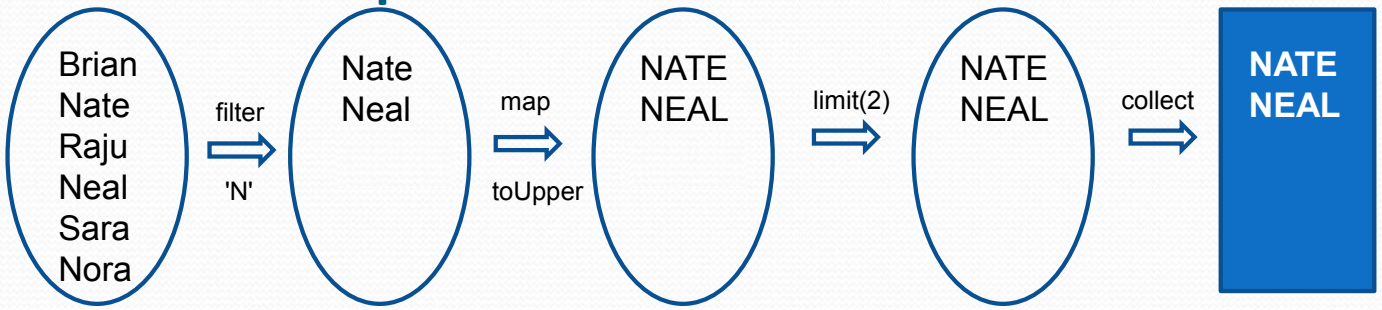
4. *Stream operations are lazy whenever possible.* Elements in the stream are accessed in a way analogous to streaming video – only the elements that are absolutely necessary are accessed, and the intermediate operations act on those. When more elements are needed, they are accessed. Diagram below shows **User View.** See `lesson9.lecture.lazystream.UserView.java`

```
public List<String> find(List<String> list,
                         String letter) {
      return list.stream()
                 .filter(name -> starts(name,letter))
                 .map(name -> toUpper(name))
                 .limit(2)
                 .collect(Collectors.toList());
      }
}
```

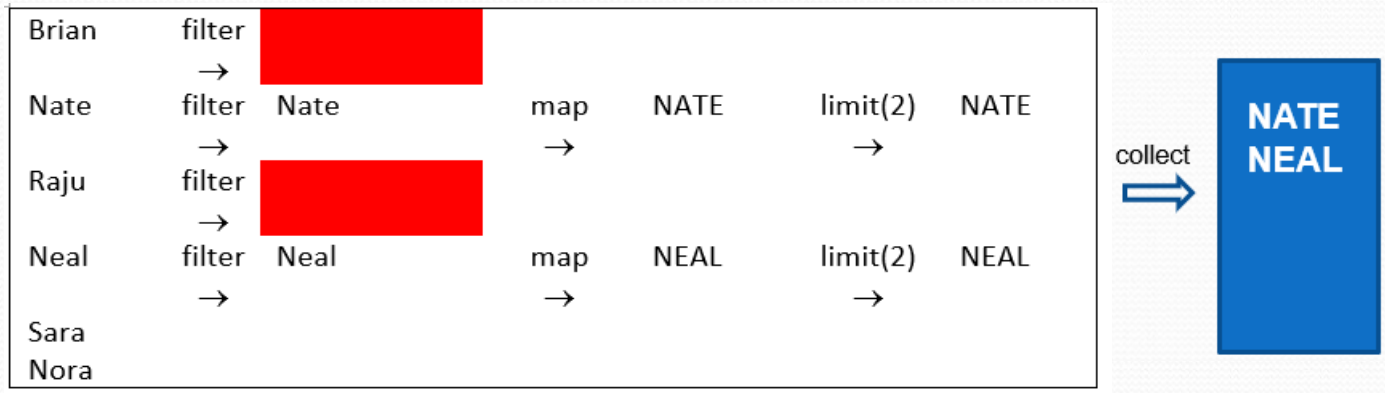
User view of a Stream pipeline



Stream Pipelines: Under the Hood



User View: New Stream after Each Operation



Under the Hood: *Stream Fusion.*

Loops are fused to produce single stream. Demo:

`lesson9.lecture.lazystream.LoopFusion.java`

Template for Using Streams

1. *Create a stream*. Typically, the stream is obtained from some kind of Collection, but streams can also be generated from scratch.

2. *Create a pipeline of operations*. Each of the operations performs some stream transformation and returns a new stream. Called *intermediate operations*.

3. *End with a terminal operation*. The terminal operation produces a result. It also forces lazy execution of the operations that precede it.

Example from Lesson 8:

```
List<String> startsWithLetter =
   list.stream()                                    //create the stream
       .filter(name -> name.startsWith(letter))   //build pipeline
       .collect(Collectors.toList()); //invoke terminal operation
```

Outline

1. Introduction to Java 8 Streams
2. Streams: Basic Facts and a 3-Step Template
3. **Different Ways to Create Streams**
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. Terminal Operations
7. Working with the Optional class
8. Reduce
9. Collecting Results
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

Ways of Creating Streams

1. Obtain a `Stream` from any `Collection` object with a call to `stream()` (this default method was added to the `Collection` interface in Java 8)

2. Get a `Stream` from an array of objects using the static *of* method:

```
Integer[] arrOfInt = {1, 3, 5, 7};
Stream<Integer> strOfInt = Stream.of(arrOfInt);
```

Cannot do the same for int[]

```
int[] arrOfInt = {1, 3, 5, 7};
   //one-element Stream
Stream<int[]> strOfInt = Stream.of(arrOfInt);
```

3. Get a Stream from any sequence of arguments: (the `of` method accepts a `varargs` argument – for a review of `varargs` see https://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html)

```
Stream<String> song
   = Stream.of("gently", "down", "the", "stream");
```

4. Two ways to obtain *infinite* streams: *generate* and *iterate* (remember stream operations are lazy)

   a. The `generate` function accepts a `Supplier<T>` argument.
   In practice, this means that it accepts functions (lambda expressions) with zero parameters.

```
interface Supplier<T> {
    T get();
}
```

Example: `Stream` of constant values ("Echo"):

```
Stream<String> echoes = Stream.generate(() -> "Echo");
```

Example: `Stream` of random numbers:

```
Stream<Double> randoms = Stream.generate(Math::random);
```

See `lesson9.lecture.generate`

b. The `iterate` function accepts a seed value (of type `T`) and a `UnaryOperator<T>` argument.

```
interface UnaryOperator<T> {
  T apply(T t);
}
```

Example: `Stream` of natural numbers: (Here, `T` is `Integer`)

```
Stream<Integer> stream2 = Stream.iterate(1, n -> n + 1));
```

Here, `1` is the first element of the generated `Stream`. Then `1` is substituted for n in the lambda to obtain the next element: `1 + 1 = 2`. Then `2` is substituted for n to obtain the next. And so on.

Note: You cannot use `int` in place of `Integer`. We discuss `Streams` based on primitives later in the lesson.

Demo: lesson9.lecture.iterate

Extracting Substreams and Combining Streams

1. `stream.limit(n)`. The call `stream.limit(n)` returns a new `Stream` that ends after n elements (or when the original `Stream` ends if it is shorter). This method is useful for cutting infinite streams down to size.

   Example:

   ```
   Stream<Double> randoms =
       Stream.generate(Math::random).limit(100);
   ```

   // Produces a stream with 100 random numbers.

2. `stream.skip(n)` The call `stream.skip(n)` *discards* the first n elements.

3. `Stream.concat(Stream, Stream)` You can concatenate two streams with the static `concat` method of the `Stream` class:

Example:

```
Stream<Character> combined =
        Stream.concat(characterStream("Hello"),
                      characterStream("World"));
// Produces the Stream
//    ['H', 'e', 'l', 'l', 'o', 'W', 'o', 'r', 'l', 'd']
```

Here is the `characterStream` method – transforms a `String` into a `Stream` of `Character`s:

```
public static Stream<Character> characterStream(String s) {
  List<Character> result = new ArrayList<>();
  for (char c : s.toCharArray())
      result.add(c);
  return result.stream();
}
```

Exercise 9.1

1. Use `Stream`'s `iterate` method to produce a `Stream` consisting of all the odd natural numbers 1, 3, 5, . . .
2. Modify your solution so that your `Stream` consists of exactly these numbers: 9, 11, 13, 15. Print your `Stream` to the console (somehow)

Main Point 1

The iterate operation on a Stream makes it possible to generate an infinite sequence of values based on two pieces of data: The initial value, and a rule or principle that tells how one value should be transformed to the next.

The iterate method is an expression of the Principle of Diving. The "correct angle" is the initial value. "Letting go" has the right effect because there is an underlying rule that directs flow from the initial value to subsequent values. During meditation, what guides awareness from one level to the next – the underlying principle or rule – is the gentle pull to move toward what is most charming at each moment while awareness remains lively and awake.

Outline

1. Introduction to Java 8 Streams
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. **Intermediate Operations on Streams**
5. Stateful Intermediate Operations
6. Terminal Operations
7. Working with the Optional class
8. Reduce
9. Collecting Results
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

Intermediate Operations on Streams:

Use ***filter*** to Extract a Substream that Satisfies Specified Criteria

- filter accepts as its argument a Predicate<T> interface.

```
interface Predicate<T> {
    boolean test(T t);
}
```

Recall the earlier example:

```
final long count = words.stream().filter(
        w -> w.length() > 12).count();
```

- The return value of filter is another Stream, so filters can be chained:

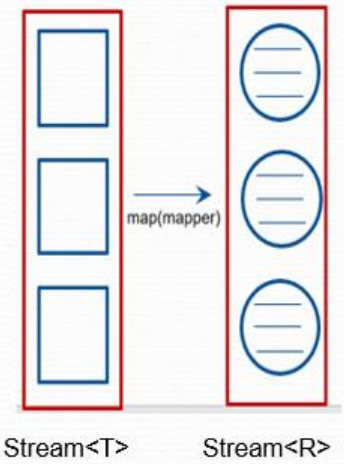
```
words.stream()
      .filter(name -> name.contains("" + c))
      .filter(name -> !name.contains("" + d))
      .filter(name -> name.length() == len)
      .count();
```

Intermediate Operations on Streams :

Use ***map*** to Transform Each Element of a Stream

- `map` accepts a `Function` interface. Typical special case of the `Function` interface is

```
interface Function<T,R> {
  R apply(T t);
}
```



- A `map` accepts a `Function` as input (sometimes called a "mapper") and returns a `Stream<R>` -- a stream of values each having type `R`, which is the return type of the `Function` interface. `maps` can therefore be chained.

- Example: Given a list `List<Integer> list` of Integers, obtain a list of Strings representing those Integers (T is `Integer`, R is `String`)

```
List<String> strings
  = list.stream().map(x -> x.toString())
        .collect(Collectors.toList())
```

Application: Using map with Constructor Method References

1. *Class::new* is a fourth type of method reference, where the method is the *new* operator. Typical translation: z -> new Class(z)
   Examples:
   A. **Button::new** - compiler must select which `Button` constructor to use; determined by context.

   When used with `map`, the `Button(String)` constructor would be used, and the constructor reference `Button::new` resolves to the following lambda:

   ```
   str -> new Button(str)
   ```

   (which realizes a `Function` interface, as required by `map`).

```
List<String> labels = ...;
Stream<Button> stream = labels.stream().map(Button::new);
List<Button> buttons = stream.collect(Collectors.toList());

//Output: a Stream of labeled Buttons, converted to a list
```

.

B. **String::new.** Again, the choice of `String` constructor depends on context.

```
public class StringCreator {
    public static void main(String[] args) {
      Function<char[], String> myFunc = String::new;
      char[] charArray =
              {'s','p','e','a','k','i','n','g','c','s'};
      System.out.println(myFunc.apply(charArray));
      System.out.println(new String(charArray));
    }

}
```

//output: speakingcs

In this case, `String::new` is short for the lambda expression

```
charArray -> new String(charArray),
```

which is a realization of the `Function` interface.

See Demo: `lesson9.lecture.newstring`

C. **int[]::new** is another constructor reference, short for the lambda expression `len -> new int[len]` (where `len` is an integer that is used as the new array length)

2. *Array constructor reference and the toArray method*

Problem

```
Stream<String> stringStream = // create Stream

//Can obtain a list of Strings like this
List<String> stringList = stringStream.collect(Collectors.toList());

//How to obtain an array of Strings? (Not like this!)
String[] vals = stringStream.toArray(); //compiler error
```

Solution: Use a constructor reference to specify the correct type:

```
String[] vals = stringStream.toArray(String[]::new);
```

```
public static void main(String[] args) {
    List<String> strings
      = Arrays.asList("Eleven", "strikes", "the", "clock");
    String[] stringArr2 = strings.stream().toArray(String[]::new);
    System.out.println(Arrays.toString(stringArr2));
}
    //Output:  [Eleven, strikes, the, clock]
```

See Demo: lesson9.lecture.constructorref.GenericArray.

Exercise 9.2

What is the following code doing? What is the output when it is run?

```
public static void main(String[] args) {
    List<Integer> ints = Arrays.asList(3,5,2,3,8);
    List<int[]> intArrs = ints.stream()
                              .map(int[]::new)
                              .collect(Collectors.toList());
    List<String> intArrsStr = intArrs.stream()
                                     .map(Arrays::toString)
                                     .collect(Collectors.toList());
    System.out.println(intArrsStr);
}
```

Solution

See `lesson9.lecture.constructorref.IntArrayExample`

Main Point 2

Java 8 makes it possible to dynamically construct objects using lambdas or method references. An application of this feature is that a developer can construct an entire Stream of new objects with very little code, by combining `map` with a constructor method reference.

This combination of new Java features reflects a deeper quality that is found in the field of intelligence itself: Intelligence naturally tends toward creative expression.

Contact with our own deepest levels of intelligence through transcending results in enhanced ability for creative expression.

Intermediate Operations on Streams :

Use ***flatMap*** to Transform Each Element to a Stream and Flatten the Result

Example Apply `characterStream` (see earlier slide) to each element of a list, using `map`:

```
List<String> list = Arrays.asList("Joe", "Tom", "Abe");
Stream<Stream<Character>> result
       = list.stream().map(s -> characterStream(s))
```

The `result Stream` is a `Stream` of `Streams`:

```
[['J', 'o', 'e'],['T', 'o', 'm'],['A', 'b', 'e']].
```

Typically, we would like to *flatten* the output, to obtain a single `Stream` of characters.

"Flattening" this `Stream` means putting all characters together in a single stream. This is accomplished using `flatMap` in place of `map`:

```
Stream<Character> flatResult
        = list.stream().flatMap(s -> characterStream(s))
```

Output:

```
['J', 'o', 'e', 'T', 'o', 'm', 'A', 'b', 'e'].
```

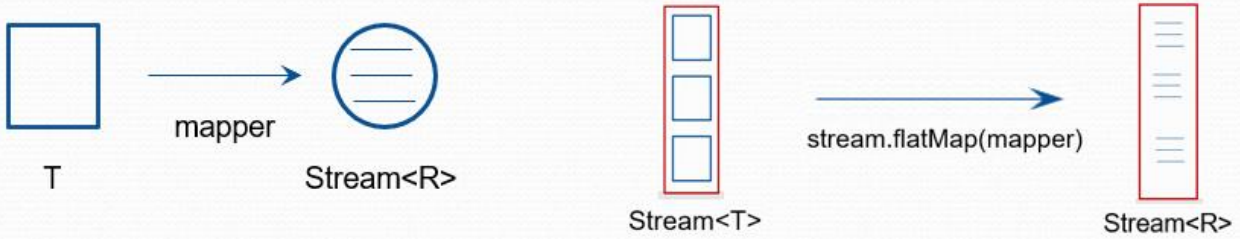
See `lesson9.lecture.flatmapstream.Test.java`.

See also `lesson9.lecture.flatmapstream.Test2.java` for a second example.

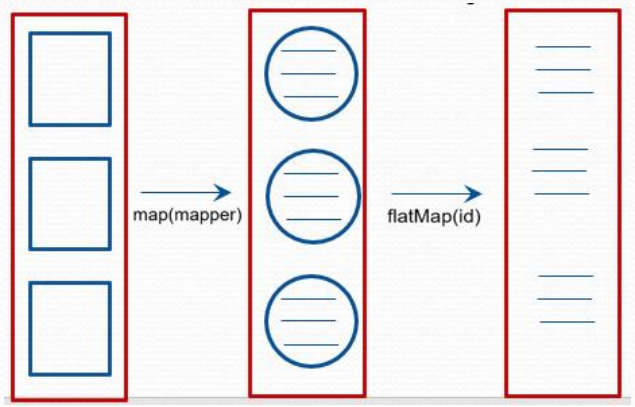
flatMap

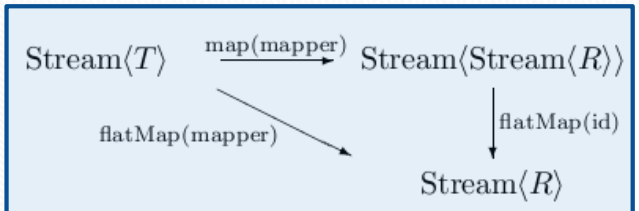
Demo: lesson9.lecture.flatmapstream

- Start with `mapper: T → Stream<R>` (like `String → Stream<Character>`)



- `flatMap` transforms a `Stream` of T's first to a `Stream` of `Stream<R>'s`, and then finally to a `Stream` of `R's`





flatMap(mapper) ==
map(mapper).flatMap((Stream s)-> s)

Outline

1. Introduction to Java 8 Streams
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. **Stateful Intermediate Operations**
6. Terminal Operations
7. Working with the Optional class
8. Reduce
9. Collecting Results
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

Stateful Intermediate Operations

1. Most of the operations discussed so far (`map`, `filter`, `concat`) have been *stateless:* each element of the stream is processed and forgotten. (Recall the "under the hood" view of `Streams`.)

2. Two *stateful* transformations we have seen so far are `limit` and `skip`. Two others that are available from a `Stream` are `distinct` and `sorted`.

3. Example of `distinct`:

```
Stream<String> uniqueWords
    = Stream.of("merrily", "merrily", "merrily","gently")
            .distinct();

//output: ["merrily", "gently"]
```

4. Example of sorted: (sorted accepts a `Comparator` parameter)

```
//sort by decreasing lengths of words
List<String> words
     = Arrays.asList("Tom", "Joseph", "Richard");
Stream<String> longestFirst
     = words.stream()
            .sorted((String x, String y) -> y.length()-x.length());
System.out.println(longestFirst.collect(Collectors.toList()));
//output: Richard, Joseph, Tom
```

Demo: `lesson9.lecture.comparators1`

Note: Sorting logic: x comes before y if `y.length() - x.length()` is negative, which happens when x is longer. So "Richard" comes before "Tom"

Note: `distinct()` and `sorted()` prevent stream fusion because they are stateful, so pipelines that use these are slightly less efficient

Note: This code uses some functional techniques, but notice that the `Comparator` still has the flavor of "how" rather than "what".

Implementing Comparators with More Functional Style

Demo: lesson9.lecture.comparators1

1. In previous example, we are seeking to sort "by `String` length", in reverse order. Rather than specifying *how* to do that, we can use the new static `comparing` method in `Comparator`:

```
Stream<String> longestFirst =
      words.stream().sorted(Comparator.comparing(String::length).reversed());
```

2. `Comparator.comparing` takes a `Function<T,U>` argument and returns another `Comparator`.

   The type `T` is the type of the `object` being compared – in the example, `T` is `String`.

   The type `U` is the type of `object` that will actually be compared - since we are comparing lengths of words, the type `U` is `Integer` in this case.

Can now write the call to `sort` even more intuitively.

```
Function<String, Integer> byLength = x -> x.length();
Stream<String> longestFirst
     = words.stream().sorted(
               Comparator.comparing(byLength).reversed())
```

Note: `reversed()` is a `default` method in `Comparator` that reverses the order defined by the instance of `Comparator` that it is being applied to.

Note: The lambda `x -> x.length()` is the same as `String::length`

3. Another example of `comparing` function: Create a `Comparator<Employee>` that compares `Employee`s by name, and another that compares by salary

```
Comparator<Employee> NameComparator
    = Comparator.comparing(Employee::getName);

Comparator<Employee> SalaryComparator
    = Comparator.comparing(Employee::getSalary);
```

4. Support for `Comparators` that are *consistent with equals*.

Demo: lesson9.lecture.comparators2
(Recall consistency with equals issue in Lab 8)

Recall when we wanted to sort `Employees` (where an `Employee` has a `name` and a `salary`) by `name`, we needed to consider also the `salary`, or else the `Comparator` is not consistent with `equals` – the following `Comparator`, passed to `Collections.sort`, is **not** consistent with equals

```
Collections.sort(emps, (e1,e2) -> {
  if(method == SortMethod.BYNAME) {
          return e1.name.compareTo(e2.name);
  } else {
          if(e1.salary == e2.salary) return 0;
          else if(e1.salary < e2.salary) return -1;
          else return 1;
  }
});
```

- This approach is "how"-oriented, and can be made more declarative by using the `comparing` and `thenComparing` methods of `Comparator`. At the same time, we make it consistent with `equals`

```
Function<Employee, String> byName = e -> e.getName();
Function<Employee, Integer> bySalary = e -> e.getSalary();

public void sort(List<Employee> emps, final SortMethod method) {
    if(method == SortMethod.BYNAME) {
        Collections.sort(emps, Comparator.comparing(byName).thenComparing(bySalary));
    } else {
        Collections.sort(emps, Comparator.comparing(bySalary).thenComparing(byName));
    }
}
```

Notes about `comparing` and `thenComparing`:

- `thenComparing` is a default method that also accepts a function of type `Function<T,U>` and it can be chained.
- `thenComparing` modifies current `Comparator` by applying its `compare` method just when the current `compare` method returns 0.

Exercise 9.3

Use the `comparing` and `thenComparing` methods of `Comparator` to sort a list of `Accounts` first by `balance`, then by `ownerName`. See lesson9.exercise_3 in the `InClassExercises` project.

```
public class Account {
    private String ownerName;
    private int balance;
    private int acctId;
    public Account(String owner, int bal, int id) {
        ownerName = owner;
        balance = bal;
        acctId = id;
    }
}
```

Solution

```
List<Account> sorted
  = accounts.stream()
            .sorted(Comparator.comparing((Account a) -> a.getBalance())
                    .thenComparing(a -> a.getOwnerName()))
            .collect(Collectors.toList());
```

Main Point 3

The `comparing` and `thenComparing` methods are examples of the new declarative style that can be used in working with `Streams` in Java 8. This style of programming makes it possible to obtain results simply by *declaring* what is needed, rather than specifying in detail *how* to obtain the results. In a similar way, as awareness becomes more and more familiar with its unbounded transcendental value, we are able to achieve goals with less effort, because transcending engages the deeper levels of nature's functioning, winning support for our individual intentions.

Outline

1. Introduction to Java 8 Streams
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. **Terminal Operations**
7. Working with the Optional class
8. Reduce
9. Collecting Results
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

Terminal Operations on Stream

1. The last step in a pipeline of `Streams` is an operation that produces a final output – such operations are called *terminal operations* because, once they are called, the stream can no longer be used.

   They are also called *reduction methods* because they reduce the `Stream` to some final value.

   We have already seen one example:
   `collect(Collectors.toList())`

2. *count:* Counts the number of elements in a `Stream`.

```
List<String> words = //…
int numLongWords
     = words.stream()
            .filter(w -> w.length() > 12)
            .count();
```

Terminal Operations

3. *max, min, findFirst, findAny* operations search a `Stream` for particular values and will throw an exception if not handled properly. An easy way to handle:

Example: `max`

```
Optional<String> largest = words.stream()
        .max(String::compareToIgnoreCase);
if (largest.isPresent())
    System.out.println("largest: " + largest.get());
```

An `Optional` is a wrapper for the answer – either the found `String` can be read via `get()`, or a `boolean` flag can be read that says no value was found (for example, if the `Stream` was empty).

You can call `get()` on an `Optional` to retrieve the stored value, but if the value was not found (in that case the `Optional` flag `isPresent` is false), calling `get()` produces a `NoSuchElementException`.

Terminal Operations

Example: `findFirst`

```
Optional<String> startsWithQ
     = words.stream()
           .filter(s -> s.startsWith("Q"))
           .findFirst();
```

Another Example: `findAny`. This operation returns an `Optional` that contains some value in an input `Stream`. (How this value is chosen from the `Stream` is not guaranteed.) If the `Stream` is empty, the returned `Optional` is empty. This operation works much better with parallel `Streams` than `findFirst`.

```
Optional<String> startsWithQ
     = words.parallelStream()
        .filter(s -> s.startsWith("Q"))
        .findAny();
```

Outline

1. Introduction to Java 8 Streams
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. Terminal Operations
7. **Working with the Optional class**
8. Reduce
9. Collecting Results
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

Working with Optional

– A Better Way to Handle Nulls

1. The previous slides introduced the `Optional` class. One reason `Optional` was added to Java was to make handling of `null`s less error prone. However notice

```
if (optionalValue.isPresent())
   optionalValue.get().someMethod();
```

has no real advantage over the following usual code:

```
if (value != null)
   value.someMethod();
```

The `Optional` class, however, supports other techniques that are superior to checking `null`s.

Working with Optional

orElse

2. The `orElse` method – if result is `null`, provides alternative output using `orElse`

```
//OLD WAY
public static void pickName(final List<String> names, final String startingLetter) {
    String foundName = null;
    for (String name : names) {
        if (name.startsWith(startingLetter)) {
            foundName = name;
            break;
        }
    }
    System.out.print(String.format("A name starting with %s: ", startingLetter));
    if (foundName != null) {
        System.out.println(foundName);
    } else {
        System.out.println("No name found");
    }
}
```

```java
//NEW WAY
public static void pickName(final List<String> names, final String startingLetter) {
    final Optional<String> foundName = names.stream()
                                            .filter(name -> name.startsWith(startingLetter))
                                            .findFirst();
    System.out.println(
            String.format("A name starting with %s: %s",
                          startingLetter,
                          foundName.orElse("No name found")));
}
```

The `orElse` method on an `Optional` returns the value stored in the `Optional`, if present. If not present, returns the alternative value supplied as an argument to `orElse`.

The alternative value supplied must have the same type as the value stored in the original `Optional`.

Creating Your Own Optionals

- **Using *of* and *empty*.** You can create an `Optional` instance in your own code using the static method *of*. However, if *of* is used on a `null` value, a `NullPointerException` is thrown, so the best practice is to use *of* together with *empty*, as in the following:

```
public static Optional<Double> inverse(Double x) {
     return x == 0 ? Optional.empty() : Optional.of(div(1,x);
}
//returns a/b if possible
private static Double div(Double a, Double b) {
     if(b == 0) return null;
     return a/b;
}
```

`Optional.empty()` creates an `Optional` with no wrapped value; in that case, the `isPresent` flag is set to `false`.

Creating Your Own Optionals - Continued

- **Using *ofNullable*.** The static method *ofNullable* lets you read in a possibly null value. In particular, `ofNullable` returns an `Optional` that embeds the specified value if non-null, otherwise returns an empty `Optional`.
- Can use `orElse/orElseGet` together with `Optional.ofNullable`.
- **Example**. Here, `readInput()` returns either null or a person's name.

```
void createOutput() {
    String outputMessage = "Hello ";
    outputMessage
       += Optional.ofNullable(readInput()).orElse("World!")
    return outputMessage;
}
```

ofNullable used with orElse/orElseGet

When you wish to obtain a value from a method call using `orElse`, use `orElseGet` instead.

- **Example**. Use `orElseGet` as a way to implement a lazy singleton. The following is an example in which a `Connection` object (JDBC) is accessed lazily within a `ConnectionManager` class.
  See `lesson5.lecture.factorymethods3.dataaccess.DataAccessSystem`

```
private static Connection conn = null;
private static Connection createConnection() {
    try {
        conn = DriverManager.getConnection(DB_URL, USERNAME, PASSWORD);
        return conn;
    } catch(SQLException e) {
        throw new RuntimeException(e);
    }
}
public static Connection getConnection()  {
    return Optional.ofNullable(conn).orElseGet(ConnectManager::createConnection);
}
```

orElseGet Syntax

- `orElseGet` expects an instance of `Supplier` as input

```
interface Supplier<T> {
    T get();
}
```

Difference Between orElse and orElseGet

- *Warning.* If you use `orElse` to call a method and use the return value, even if `ofNullable` encounters a non-null input, the method call in `orElse` will still execute, but the return value is ignored. (This is usually undesirable)

In the code on previous slide, if we write instead

```
public Connection getConnection() {
        return Optional.ofNullable(conn)
           .orElse(ConnectManager::createConnection);
}
```

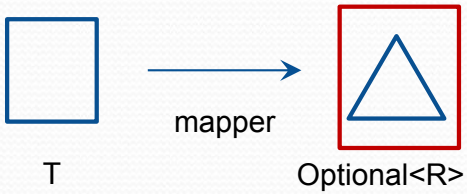
then, when `conn` is not null, `orElse` will cause `createConnection()` to be invoked (not good), but will ignore its return value and return `conn` instead.

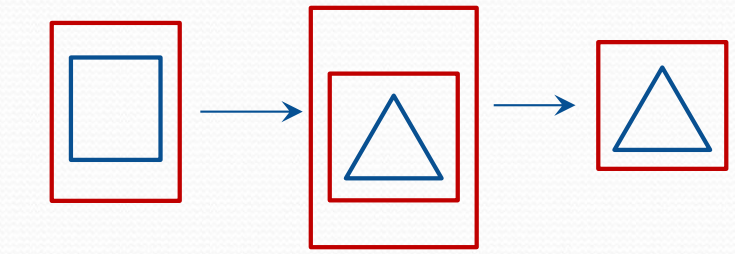
In the same scenario, the following code will *not* cause `createConnection()` to be invoked when `conn` is not null.

```
public Connection getConnection() {
   return Optional.ofNullable(conn)
       .orElseGet (ConnectManager::createConnection);
}
```

Using Optional : flatMap

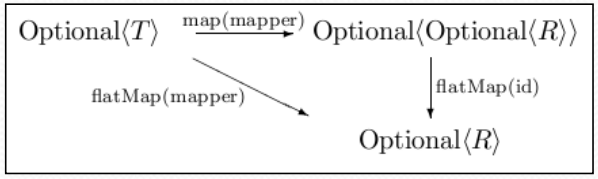
The `flatMap` method works the same way as it does on `Stream`. It allows you to chain method calls and skip null checks. Given a mapper from type `T` to an `Optional<R>`, `flatMap` transforms an `Optional<T>` first to an `Optional<Optional<R>>`, then to an `Optional<R>`





Optional<R>

`flatMap(mapper)`



Example of Optional's flatMap

Demo: lesson9.lecture.optional_flatmap.usingoptionals

Problem: At most one company is a winner in a multi-company contest. The winning company will have one or more employees with a winning ticket. Find the primary telephone number of the employee with the biggest winnings.

Imperative Solution

```
private static String telNumberOfBiggestWinnerImperative(List<Company> list) {
    boolean winningCompanyFound = false;
    for(Company c: list) {
        if(c.hasWinningTicket()) {
            winningCompanyFound = true;
            int largestSoFar = 0;
            Employee biggestWinner = null;
            List<Employee> emps = c.getEmployees();
            for(Employee e: emps) {
                int wins = e.getWinningAmount();
                if(wins > largestSoFar) {
                    largestSoFar = wins;
                    biggestWinner = e;
                }
            }
            List<String> telNums = biggestWinner.getTelephoneNumbers();
            if(telNums != null && telNums.size() > 0)
                return telNums.get(0);
            else
                return "Winner has no phone number";
        }
    }
    if(!winningCompanyFound) {
        return "No winning company";
    }
    return null;
}
```

Solution with flatMap and Optional

```
static Comparator<Employee> winnings
   = Comparator.comparing((Employee e) -> e.getWinningAmount());

private static String telNumberOfBiggestWinner(List<Company> list) {
    return list.stream()
               .filter((Company c) -> c.hasWinningTicket())
               .findAny()
               .flatMap((Company c) -> c.getEmployees().stream()
                   .max(winnings))
               .flatMap((Employee e) -> e.getTelephoneNumbers()
                   .stream().findFirst())
               .orElse("Not found");
}
```

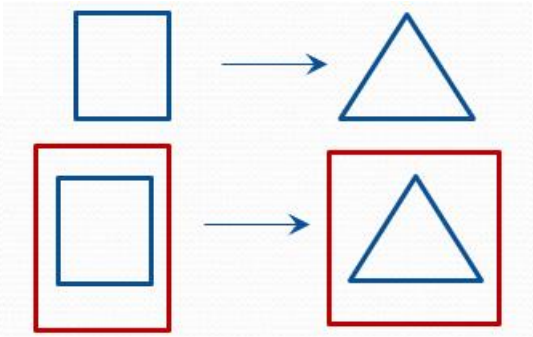
Example: Replacing null checks with Optional

Consider imperative code for seeing if a `Person` from Fairfield can be found in a list of `Person`s:

```
private static boolean personFromFairfield(List<Person> persons)  {
    Person foundPerson = null;
    for(Person p: persons) {
        if(p != null) {
            Address addr = p.getAddress();
            if(addr != null) {
                String city = addr.getCity();
                if(city != null) {
                    if(city.equals("Fairfield")) {
                        foundPerson = p;
                    }
                }
            }
        }
    }
    return foundPerson != null;
}
```

- Using `Optionals` with `map` completely eliminates these (unnecessary) `null` checks

```
private static boolean personFromFairfield(List<Person> persons) {
    for(Person p: persons) {
        if(Optional.ofNullable(p).map(x -> x.getAddress())
                .map(x->x.getCity())
                .orElse("").equals("Fairfield")) {
            return true;
        }
    }
    return false;
}
```



- **Notes.**
  1. If an `Optional opt` is empty, `opt.map(mapper)` returns an empty `Optional`
  2. See lesson9.lecture.optional_map.usingoptionals

Outline

1. Introduction to Java 8 Streams
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. Terminal Operations
7. Working with the Optional class
8. **Reduce**
9. Collecting Results
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

Terminal Operations continued: The reduce Operation

- The `reduce` operation lets you combine the terms of a `Stream` into a single value by repeatedly applying an operation.

Example We wish to sum the values in a list **`digits`** of numbers. Procedural code:

```
int sum = 0;
for(int x : digits) {
  sum += x;
}
```

Using the `reduce` operation, the code looks like this:

```
Integer sum2 = digits.stream().reduce(0, (x, y) -> x + y);
```

The reduce Operation (cont.)

```
digits.stream().reduce(0, (x, y) -> x + y)
```

- *First Argument: Initial Value.* First argument is the initial value; it is the value that is returned if the stream is empty and is the starting point for the computation. It is also (or *should be*) the *identity element* for the combining operation.
- *Second Argument:* A lambda of type `BinaryOperator<T>`.

```
interface BinaryOperator<T> {
    T apply(T a, T b);
}
```

- *The Computation.* The computation starts by applying the binary operator to the pair (e, f), where e is the initial value (0 in this example) and f is the first element of the `Stream`, producing `accum`. Then the binary operator is applied to (`accum`, g) where g is the next element in the `Stream`, and the result is stored in `accum` again. This continues until all `Stream` values have been used.

- When the `stream` consists of numbers and the binary operator is ordinary addition, the output is the sum of all the numbers in the stream. If the stream is empty, the initial value is returned – note that the sum of an empty collection of numbers is by convention 0.

- **Example**. Start with the following `stream`: `[2,1,4,3],` and use ordinary addition for the `BinaryOperator` functor. Then the `reduce` method performs the following computation:

```
(((0 + 2) + 1) + 4) + 3 = 10
```

- A parallel computation can improve performance. Say [2, 1, 4, 3] is broken up into [2, 3], [4, 1]. Then in parallel we arrive at the same answer in the following way:

```
sum1 = (0 + 2) + 3        sum2 = (0 + 4) + 1
combined = sum1 + sum2 = 10
```

The reduce Operation (continued)

- **Question:** How could we form the *product* of a list of numbers?

- **Answer:** We form the product of a list `numbers` of Integers. For the initial value, we ask, "What is the identity element for the multiplication operation?" The identity is 1. (**Note** that, by convention, the product of an empty list of numbers is 1.)

  Here is the line of code that does the job:

```
int product = numbers.stream().reduce(1, (x, y) -> x * y);
```

The reduce Operation (continued)

- Example. What about subtraction? What happens when the following line of code is executed? Try it when numbers is the list [2, 1, 4, 3].

```
int difference=numbers.stream().reduce(0, (a, b) -> a - b);
```

- Here, the computation proceeds like this:

```
((0 - 2) - 1) - 4) - 3)    //output: -10
```

- The problem here is that performing this computation in parallel gives a different result; subtractions are grouped differently for a parallel computation. For instance, during parallel computation, if [2, 1, 4, 3] is broken up into [2, 3] and [4, 1], the computation would look like this:

```
diff1 = (0 - 2) - 3   diff2 = (0 - 1) - 4
combined = diff1 - diff2 = 0
```

- For this reason, not only should the initial value be an identity element (it isn't an identity for subtraction) but also:

Use reduce only on operations that are commutative and associative.

(Note that + and * are commutative & associative, but subtraction is not.)

See the demo `lesson9.lecture.reduce`.

The reduce Operation (continued)

- The reduce method has an overloaded version with only one argument.

- Continuing with the sum example, here is a computation with the overridden version:

```
Optional<Integer> sum
   = Stream.of(digits).reduce((x, y) -> x + y);
```

- This version of `reduce` produces the same output as the earlier version *when the stream is nonempty,* but it is stored in an `Optional` in this case. When the `stream` is empty, the `reduce` operation returns a `null`, which is again embedded in an `Optional`.

Exercise 9.4

Use `reduce` to concatenate the `String`s in the `Stream` below to form a single, space-separated `String`. Print the result to the console. See `lesson9.exercise_4` in the `InClassExercises` project.

```
public static void main(String[] args) {
    Stream strings = Stream.of("A", "good", "day", "to", "write", "some", "Java");
}
```

Main Point 4

When a Collection is wrapped in a Stream, it becomes possible to rapidly make transformations and extract information in ways that would be much less efficient, maintainable, and understandable without the use of Streams. In this sense, Streams in Java represent a deeper level of intelligence inherent in the concept of "collection" that has been implemented in the Java language.

When intelligence expands, challenges and tasks that seemed difficult and time-consuming before can become effortless and meet with consistent success. This is one of the documented benefits of TM practice.

Outline

1. Introduction to Java 8 Streams
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. Terminal Operations
7. Working with the Optional class
8. Reduce
9. **Collecting Results**
10. Primitive Type Streams
11. Stream Reuse and Lambda Libraries

Collecting Results

- One kind of terminal operation in a stream pipeline is a *reduction* that outputs a single value, like `max` or `count` or `reduce`. Another kind of terminal operation collects the elements of the `Stream` into some type of collection, like an `array`, `list`, or `map`. We have seen examples already.

  Example: Collecting into an `array`

```
String[] result = stream.toArray(String[]::new);
```

  Example: Collecting into a `List`

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

  Example: Collecting into a `Set`

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

  Example: Collecting into a particular kind of `Set` (same idea for particular kinds of lists, maps)

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

Collecting Results (cont.)

Example Collect all strings in a `Stream` by concatenating them:

```
String result = stream.collect(Collectors.joining());
```

//separates `Strings` by commas

```
String result = stream.collect(Collectors.joining(", "));
```

//prepares objects as strings before joining

```
String result = stream.map(Object::toString)
                      .collect(Collectors.joining(", "));
```

Note: Here instead of `Object::toString` you can use your own object type, like `Employee::toString`. By polymorphism, either way works. See demo `lesson9.lecture.collect`

Collecting Results (cont.)

Example Collecting into a `map` – two typical examples.

Here, `personStrm` is a `Stream` of `Person` objects.

```
//key = id, value = name
Map<Integer, String> idToName =
   personStrm.collect(Collectors.toMap(Person::getId,
                                       Person::getName));
//key = id, value = the person object
Map<Integer, Person> idToPerson =
   personStrm.collect(Collectors.toMap(Person::getId,
                                       Function.identity()));
```

Note: `identity` is a `static` method on `Function` that returns a function that always returns its input argument. In the example, it is the function

```
(Person p) -> p
```

Collecting Results (cont.)

- Can collect "summary statistics" for `Streams` whose elements can be mapped to `ints`. `IntSummaryStatistics` provides sum, average, maximum, and minimum

```
IntSummaryStatistics summary =
    words.collect(Collectors.summarizingInt(String::length));
double averageWordLength = summary.getAverage();
double maxWordLength = summary.getMax();

//Recall: String::length means str -> str.length()
```

- Similar SummaryStatistics classes are available for `Double` and `Long` types too:

  - `DoubleSummaryStatistics` uses `Collectors.summarizingDouble`

  - `LongSummaryStatistics` uses `Collectors.summarizingLong`.

- **Note**: `IntSummaryStatistics` extracts `int` information from an input `Stream`. The elements of the `Stream` will therefore be converted at runtime to (primitive) `ints` in order for `summarizingInt` to perform its tasks.

- The `summarizingInt` method expects an argument that is an implementation of the `ToIntFunction<T>` interface:

```
interface ToIntFunction<T> {
      int applyAsInt(T value);
}
```

Exercise 9.5

Use `DoubleSummaryStatistics` to output to the console the top test score, lowest test score, and average among all test scores in a given list. See lesson9.exercise_5 in InClassExercises project.

```
public class ExamData {
    private String studentName;
    private double testScore;
    public ExamData(String name, double score) {
        studentName = name;
        testScore = score;
    }

}
```

```
public static void main(String[] args) {
    List<ExamData> data = new ArrayList<ExamData>() {
        {
            add(new ExamData("George", 91.3));
            add(new ExamData("Tom", 88.9));
            add(new ExamData("Rick", 80));
            add(new ExamData("Harold", 90.8));
            add(new ExamData("Ignatius", 60.9));
            add(new ExamData("Anna", 77));
            add(new ExamData("Susan", 87.3));
            add(new ExamData("Phil", 99.1));
            add(new ExamData("Alex", 84));
        }
    };
}
```

Main Point 5

To obtain some information about a collection, we can lift the collection to a Stream, perform transformations at the Stream level, and then, by way of some terminal operation, collapse the Stream to obtain a final output. It is by virtue of accessing the powerful dynamics available in Stream that the result is obained.

In a similar way, experience of transcending makes it possible to fulfill desires and intentions with a minimum of effort: When the mind transcends, awareness become saturated with the level from which all the laws of nature begin to operate – the unified field. Having the support of this powerful level, desires can effortlessly meet with fulfillment.

Outline

1. Introduction to Java 8 Streams
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. Terminal Operations
7. Working with the Optional class
8. Reduce
9. Collecting Results
10. **Primitive Type Streams**
11. Stream Reuse and Lambda Libraries

Primitive Type Streams

There are variations of `Stream` specifically designed for primitives: `int`, `double`, and `long`:

- `IntStream`
- `DoubleStream`
- `LongStream`

For primitive types `short`, `char`, `byte`, and `boolean`, use `IntStream`; for `floats`, use `DoubleStream`.

1. Creation methods are similar to those for `Stream`:
   a. `IntStream ints = IntStream.of(1, 2, 4, 8);`
   b. `IntStream ones = IntStream.generate(() -> 1);`
   c. `IntStream naturalNums = IntStream.iterate(1, n -> n+1);`

2. `IntStream` (and also `LongStream`) have static methods `range` and `rangeClosed` that generate integer ranges with step size one:

```
// Upper bound is excluded
IntStream zeroToNinetyNine = IntStream.range(0, 100);

// Upper bound is included
IntStream zeroToHundred = IntStream.rangeClosed(0, 100);
```

3. To convert a `primitive` type stream to an stream of objects, use the `boxed()` method:

```
Stream<Integer> integers = IntStream.range(0, 100).boxed();
```

4. To convert an object stream to a primitive type stream, there are methods `mapToInt`, `mapToLong`, and `mapToDouble`. In the example, a `Stream` of strings is converted to an `IntStream` (of lengths).

```
Stream<String> words = ...;
IntStream lengths = words.mapToInt(String::length);
```

5. The methods on primitive type streams are analogous to those on object streams. Here are the main differences:
   - The `toArray` methods return primitive type arrays.
   - Methods that yield an `Optional` result return an `OptionalInt`, `OptionalLong`, or `OptionalDouble`. These classes are analogous to the `Optional` class, but they have methods `getAsInt`, `getAsLong`, and `getAsDouble` instead of the `get` method.
   - There are methods `sum`, `average`, `max`, and `min` that return the sum, average, maximum, and minimum. These methods are not defined for object streams. (Note that the functions `max` and `min` defined on an ordinary `Stream`, require a `Comparator` argument, and return an `Optional`.)

Outline

1. Introduction to Java 8 Streams
2. Streams: Basic Facts and a 3-Step Template
3. Different Ways to Create Streams
4. Intermediate Operations on Streams
5. Stateful Intermediate Operations
6. Terminal Operations
7. Working with the Optional class
8. Reduce
9. Collecting Results
10. Primitive Type Streams
11. **Stream Reuse and Lambda Libraries**

Can Streams Be Re-Used?

- Once a terminal operation has been called on a `Stream`, the `Stream` becomes unusable, and if you do try to use it, you will get an `IllegalStateException`.
- But sometimes it would make sense to have a `Stream` ready to be used for multiple purposes.
- **Example** We have a `Stream<String>` that we might want to use for different purposes:

```
Folks.friends.stream().filter(name -> name.startsWith("N"))
```

Typical Uses:

1. count the number of names obtained
2. output the names in upper case to a List

But once the `Stream` has been used once, we can't use it again.

Stream Re-use Techniques

- Solution #1 One solution is to place the stream-creation code in a method and call it for different purposes. See `Good` solution in package `lesson9.lecture.streamreuse`

- Solution #2 Another solution is to capture all the free variables in the first approach as parameters of some kind of a `Function` (might be a `BiFunction`, `TriFunction`, etc, depending on the number of parameters). See `Reuse` solution in package `lesson9.lecture.streamreuse`

- The second solution leads to a useful way of storing stream pipelines for reuse similar to techniques from database management

Creating a Lambda Library

Java 8 lets you perform *queries* to work with data in a `Collection` of some kind. The query style is similar to SQL queries. And queries can be stored in a way that is similar to SQL queries.

Database Problem. You have a database table named `Customer`. Return a collection of the names of those `Customers` whose city of residence begins with the string "Ma", arranged in sorted order.

Solution: `SELECT name FROM Customer WHERE city LIKE 'Ma%' ORDER BY name`

We can store the query by naming it and replacing parameters with wildcards:

```
SEARCH_CITY_IN_CUST = "SELECT name FROM Customer WHERE city LIKE '?'
ORDER BY name"
```

Then use in various applications:

```
Connection conn = ConnectManager.getConnection();
PreparedStatement stat = conn.prepareStatement(SEARCH_CITY_IN_CUST );
stat.setString(1, 'Ma%');
```

Similar Java Problem: You have a List of `Customers`. Output to a list, in sorted order, the names of those `Customers` whose city of residence begins with the string "Ma."

Solution:

```
List<String> listStr =
   list.stream()
       .filter(cust -> cust.getCity().startsWith("Ma"))
       .map(cust -> cust.getName())
       .sorted().collect(Collectors.toList());
```

Turning Your Stream Pipeline into a Library Element

How to turn your solution into a reusable Lambda Library element:

> *Identify the parameters and treat them as arguments for some kind of Java function-type interface* (`Function`, `BiFunction`, `TriFunction`, etc).

Parameters in this problem:

- An input list of type `List<Customer>`
- A target string used to compare with name of city, of type `String`
- Return type: a list of strings: `List<String>`

Turning Your Stream Pipeline into a Library Element (cont.)

These suggest using a `BiFunction` as follows:

```
public static final BiFunction<List<Customer>, String,
List<String>> NAMES_IN_CITY = (list, searchStr) ->
    list.stream()
        .filter(cust -> cust.getCity()
        .startsWith(searchStr))
        .map(cust -> cust.getName())
        .sorted()
        .collect(Collectors.toList());
```

The Java solution can now be rewritten like this:

```
List<String> listStr =
   LambdaLibrary.NAMES_IN_CITY.apply(list, "Ma");
```

See the code in `lesson9.lecture.lambdalibrary`.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Lambda Libraries

1. Prior to the release of Java 8, extracting or manipulating data in one or more lists or other Collection classes involved multiple loops and code that is often difficult to understand.
2. With the introduction of lambdas and streams, Java 8 makes it possible to create compact, readable, reusable expressions that accomplish list-processing tasks in a very efficient way. These can be accumulated in a Lambda Library.

---

3. *Transcendental Consciousness* is the field that underlies all thinking and creativity, and, ultimately, all manifest existence.
4. *Impulses Within the Transcendental Field*. The hidden self-referral dynamics within the field of pure intelligence provides the blueprint for emergence of all diversity. This blueprint is formed from compact expressions of intelligence are coherently arranged in the first sprouting of existence.
5. *Wholeness Moving Within Itself.* In Unity Consciousness, the fundamental forms out of which manifest existence is structured are seen to be vibratory modes of one's own consciousness.


MAHARISHI UNIVERSITY of MANAGEMENT
Engaging the Managing Intelligence of Nature
Computer Science Department
CS401 Modern Programming Practices (MPP)
Professor Paul Corazza
1

Lecture 11: Java Generics:

Weaving the Universal into the Fabric of the Particular

Wholeness Statement

Java generics facilitate stronger type-checking, making it possible to catch potential casting errors at compile time (rather than at runtime), and in many cases eliminate the need for downcasting. Generics also make it possible to support the most general possible API for methods that can be generalized. We see this in simple methods like max and sort, and also in the new Stream methods like filter and map. Generics involve type variables that can stand for any possible type; in this sense they embody a universal quality. Yet, it is by virtue of this universal quality that we are able to specify particular types (instead of using a raw List, we can use List<T>, which allows us to specify a list of Strings – List<String> -- rather than a list of Objects, as we have to do with the raw List). This shows how the lively presence of the universal sharpens and enhances the particulars of individual expressions. Likewise, contact with the universal level of intelligence sharpens and enhances individual traits.

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Generic programming with generics

Introducing Generic Parameters

- Prior to jdk 1.5, a collection of any type consisted of a collection of Objects, and downcasting was required to retrieve elements of the correct type.

```
List words = new ArrayList();
words.add("Hello");
words.add(" world!");
String s = ((String)words.get(0)) + ((String)words.get(1));
System.out.print(s);  //output: Hello world!
```

- In jdk 1.5, generic parameters were added to the declaration of collection classes, so that the above code could be rewritten as follows:

```
List<String> words = new ArrayList<String>();
words.add("Hello");
words.add(" world!");
String s = words.get(0) + words.get(1);
System.out.print(s);  //output: Hello world!
```

Benefits of Generics

1. *Stronger type checks at compile time*. A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Detecting errors at compile time is always preferable to discovering them at runtime (especially since, otherwise, the problem might not show up until the software has been released).

   Example of poor type-checking

```
List myList = new myList();
myList.add("Tom");
myList.add("Bob");
    . . .
// no compiler check to prevent this
Employee tom = (Employee)myList.get(0);
```

2. *Reduced Downcasting*. Downcasting is considered an "anti-pattern" in OO programming. Typically, downcasting should not be necessary (though there are plenty of exceptions to this rule); finding the right subtype should be accomplished with late binding.

Example of bad downcasting

```
//Populate with Triangles and Rectangles
ClosedCurve[] closedCurves = . . .
if(closedCurves[0] instanceOf Triangle) {
    print((Triangle)closedCurve[0].area());
}
else {
    print((Rectangle)closedCurve[0].area());
}
```

3. *Supports the most general possible API for methods that can be generalized .*

Example *Task*: get the `max` element in a list (difficult to do without generics)

```
public static Integer max0(List<Integer> list) {
            Integer max = list.get(0);
            for(Integer i : list) {
                  if(i.compareTo(max) > 0) {
                        max = i;
                  }
            }
            return max;
      }
```

```
public static <T extends Comparable<T>> T max1(List<T> list) {
   T max = list.get(0);
   for(T i : list) {
      if(i.compareTo(max) > 0) {
         max = i;
      }
   }
   return max;
}
```

Generics Terminology and Naming Conventions

1. In the following code:

```
List<String> words = new ArrayList<String>();
words.add("Hello");
words.add(" world!");
String s = words.get(0) + words.get(1);
System.out.print(s);  //output: Hello world!
```

the class (found in the Java libraries) with declaration

```
class ArrayList<T> { . . . }
```

is called a *generic class*, and T is called a *type variable* or *type parameter*.

2. The delcaration

```
List<String> words; //read:"List of String"
```

is called a *generic type invocation,* `String` is (in this context) a *type argument*, and `List<String>` is called a *parametrized type.* Also, the class `List`, with the type argument removed, is called a *raw type.*

Note the difference between *generic class* and *parameterized type*, and between *type variable* and *type argument.* In each case, the first makes use of a type variable (like *T*) and the second provides a concrete type (like `String`).

Note: When raw types are used where a parametrized type is expected, the compiler issues a warning because the compile-time checks that can usually be done with parametrized types cannot be done with a raw type.

3. Commonly used type variables:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

Creating Your Own Generic Class or Interface

```
public class SimplePair<K,V> {
    private K key;
    private V value;

    public SimplePair(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

Notes:

1. The class declaration introduces type variables `K, V`. These can then be used in the body of the class as types of variables and method arguments and return types. The same principle applies when defining a generic interface.
2. The type variables may be realized as any Java object type (even user-defined), but not as a primitive type.

Usage Example:

```
SimplePair<Integer, String> pair
    = new SimplePair<>(10123, "Jim");
Integer employeeId = pair.getKey(); //returns Jim's ID
```

Implementing a Generic Interface

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}
```

- One way: Create a parametrized type implementation

```
public class MyPair implements Pair<String, Integer>{
    private String key;
    private Integer value;

    public MyPair(String key, Integer value) {
        this.key = key;
        this.value = value;
    }

    @Override
    public String getKey() {
        return key;
    }
    @Override
    public Integer getValue() {
        return value;
    }
```

Like: `MyComparator implements Comparator<Employee>`

- Another way: Create a generic class implementation

```
public class OrderedPair<K, V> implements Pair<K, V> {
    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
```

Same as: `ArrayList<T> implements List<T>`

See Demos: lesson11.lecture.generics, lesson11.lecture.generics.pairexamples

Extending a Generic Class

The same points apply for extending a generic class.

Either: Create a generic subclass

```
public class MyList<T> extends ArrayList<T>{
  …
}
```

Or: Create a parametrized type subclass

```
public class MyList extends ArrayList<String>{
  …
}
```

How Java Implements Generics: *Type Erasure*

The compiler transforms the following generic code

```
List<String> words = new ArrayList<String>();
words.add("Hello");
words.add(" world!");
String s = words.get(0) + words.get(1);
System.out.print(s);  //output: Hello world!
```

into the following non-generic code:

```
List words = new ArrayList();
words.add("Hello");
words.add(" world!");
String s = ((String)words.get(0)) +
((String)words.get(1));
System.out.print(s);  //output: Hello world!
```

How Java Implements Generics: *Type Erasure (cont.)*

1. Java is said to implement generics *by erasure* because the parametrized types like `List<String>`, `List<Integer>` and `List<List<Integer>>` are all represented at runtime by the single type `List`.

2. Also *erasure* is the process of converting the first piece of code to the second.

3. The compiled code for generics will carry out the same downcasting as was required in pre-generics Java.

How Java Implements Generics: *Type Erasure (cont.)*

Benefits of this implementation approach:

A. No increase in the number of types in the language (in C++, each parametrized type is a genuinely different type)
B. Backwards compatibilty with non-generic code – for instance, in both generic and non-generic code, there is, at runtime, only one type List, so legacy code and generic code can intermingle without any difficulty.

Ways That Java's Implementation of Generics Is Unintuitive

1. *Generic Subtyping Is Not Covariant.*

`Manager`	is a subclass of	`Employee`
	BUT	
`ArrayList<Manager>`	is NOT a subclass of	`ArrayList<Employee>`

2. *Array Subtyping Is Covariant*

`Manager`	is a subclass of	`Employee`
	AND	
`Manager[]`	is a subclass of	`Employee[]`

Exercise 11.1

This exercise illustrates one reason why generic subtyping is not allowed to be covariant. Examine the following code and answer the following:

1. In what step is the programmer attempting to use (generic subtype) covariance?
2. Assuming the Java designers had decided to permit covariant generic subtyping, what happens in the code that is undesirable – and should not be allowed? (Hint: What is contained in the `ints` list at the end?)

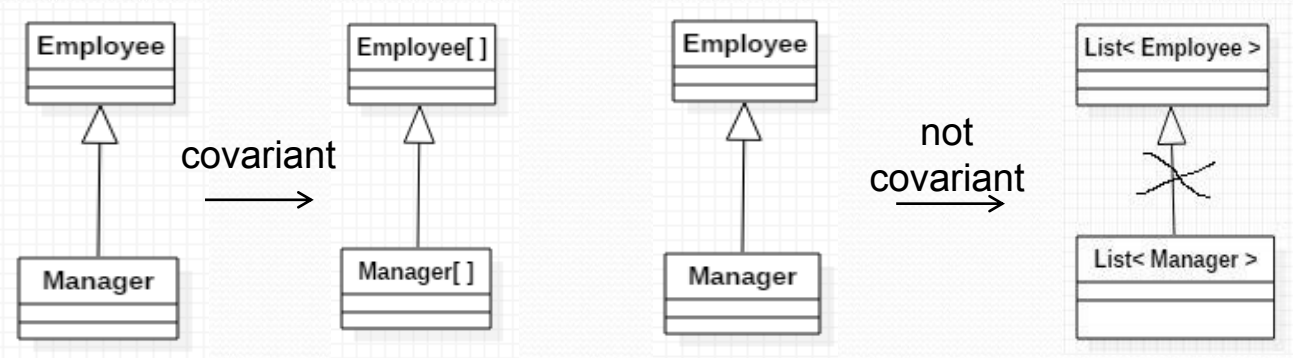
```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<Number> nums = ints;
nums.add(3.14);
System.out.print(ints);
```

Solution to Exercise 11.1

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);

//assuming covariance, this step would be allowed
List<Number> nums = ints;
nums.add(3.14);
System.out.print(ints);   //output: [1, 2, 3.14] – not desirable
```

Array Subtyping vs Generic Subtyping



Ways That Java's Implementation of Generics Is Unintuitive (continued)

2. *Component type of an array is not allowed to be a type variable*. For example, we cannot create an array like this (the compiler has no information about what type of object to create)

```
T[] arr = null; //this is ok so far
arr = new T[5]; //this produces a compiler error
```

Example: [This issue arises in Java's Collection classes]

```
class AbstractCollectionFirstTry {
  public static <T> T[] toArray(Collection<T> coll) {
        T[] arr = new T[coll.size()]; //compiler error
        int k = 0;
        for(T element : coll)
           arr[k++] = element;
           return arr;
  }
}
```

See demo: lesson11.lecture.generics.toarray (resolves the compiler error)

Ways That Java's Implementation of Generics Is Unintuitive (cont)

3. *Component type of an array is not allowed to be a parametrized type.* For example: you cannot create an array like this:

```
List<String>[] = new List<String>[5];
```

Example:

```
class Another {
   public static List<Integer>[] twoLists() {
        List<Integer> list1 = Arrays.asList(1, 2, 3);
        List<Integer> list2 = Arrays.asList(4, 5, 6);

        //compiler error
        return new List<Integer>[] {list1, list2};
   }
}
```

Ways That Java's Implementation of Generics Is Unintuitive (cont)

Reifiable Types

The reason for rule (3) (previous slide) is that the component type of an array must be a reifiable type.

Consider the analogous situation with arrays: The following statement

```
new String[size]
```

allocates an array, and stores in that array an indication that its components are of type String. However, executing

```
new  ArrayList<String>()
```

allocates a list, but does not store in the list any indication of the type of its elements. We say that Java reifies array component types but does not reify list element types (or other generic types). In the case of

```
new List<Integer>[]      //not allowed
```

because the List type does not store component type information, the resulting array is unable to store component type information (which violates rules for arrays). We say parametrized types (as well as type variables) are not reifiable.

Precise definition: A type is reifiable if the type is completely represented at run time — that is, if erasure does not remove any useful information.

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Generic programming with generics

Generic Methods

Generic methods are methods that introduce their own type parameters. This is similar to declaring a generic type, but the type parameter's scope is limited to the method where it is declared. Static and non-static generic methods are allowed, as well as generic class constructors.

// A method in a Util class

```
public static <K, V> boolean compare(SimplePair<K, V> p1, SimplePair<K, V> p2) {
    return p1.getKey().equals(p2.getKey()) &&
           p1.getValue().equals(p2.getValue());
}
```

Calling a Generic Method

Earlier versions of Java required that you specify the generic type arguments when calling a generic method (see below), but current versions are always able to infer types, so the type arguments can be left out.

The complete syntax for invoking this method would be:

```
SimplePair<Integer, String> p1 = new SimplePair<>(1, "apple");
SimplePair<Integer, String> p2 = new SimplePair<>(2, "pear");
boolean areTheySame = Util.<Integer, String>compare(p1, p2);
```

The generic type(s) can always be inferred by the compiler, and can be left out.

```
SimplePair<Integer, String> q1 = new SimplePair<>(1, "apple");
SimplePair<Integer, String> q2 = new SimplePair<>(2, "pear");
boolean areTheySame2 = Util.compare(q1, q2);
```

Using Generic Methods to Generalize Behavior

The following code counts occurrences of a target String inside a given input array.

```
public static int countOccurrences(String[] arr, String target) {
    int count = 0;
    if (target == null) {
        for (String item : arr)
            if (item == null)
                count++;
    } else {
        for (String item : arr)
            if (target.equals(item))
                count++;
    }
    return count;
}
```

But the same procedure could be used to find a target of any given type in an array of the same type. Generic methods allow us to generalize from type String to an arbitrary type T.

Exercise 11.2

The code for `countOccurrences` is in the package `lesson11.exercise_2` in the `InClassExercises` project.

Do the following:

1. Turn the method into a generic method so that it can be used to count occurrences of an object of any type in an array whose components are of the same type.
2. Then try writing the code for your generic method using a `Stream` pipeline instead of imperative code

Main Point 1

Generic methods make it possible to create general-purpose methods in Java by declaring and using one or more type variables in the method. This allows a user to make use of the method using any data type that is convenient, with full compiler support for type-checking. Likewise, when individual awareness has integrated into its daily functioning the universal value of transcendental consciousness, the awareness is maximally flexible, able to flow in whatever direction is required at the moment, free of rigidity and dominance of boundaries.

Another Generalization Example: Finding the max

Problem: Find the max value in a `List`.

1. **Easy Case**: First try finding the max of a list of Integers:

```
public static Integer max(List<Integer> list) {
    Integer max = list.get(0);
    for(Integer i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

2. **Try to generalize** to an arbitrary type T (this first try doesn't quite work…)

```
public static <T> T max1(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

Problem: `T` may not be a type that has a `compareTo` operation – we get a compiler error

Solution: It is possible to use the "extends" keyword to force the type T to be an implementer of the Comparable interface. This produces a *bounded type variable* `T extends Comparable`

```
public static <T extends Comparable> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

Demo: lesson11.lecture.generics.genericprogrammingmax

Generalizing Even Further

- The `Comparable` interface is also generic. For a given class C, implementing the `Comparable` interface implies that comparisons will be done between a current instance of C and another instance; the other instance type is the type argument to use with `Comparable`. For example, `String implements Comparable<String>`. This leads to:

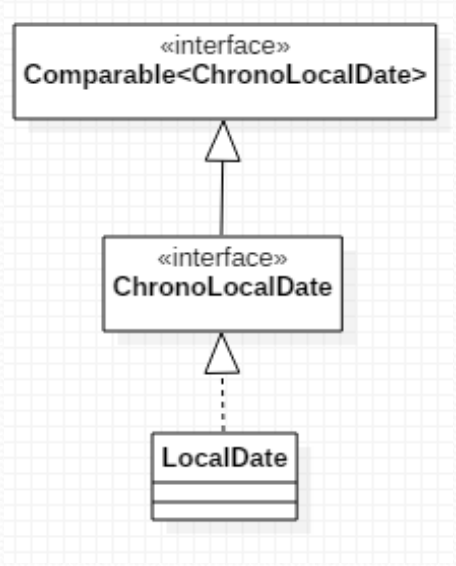
```
public static <T extends Comparable<T>> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

- This version of max can be used for most kinds of Lists, but there are exceptions. Example:

```
public static void main(String[] args) {
    List<LocalDate> dates = new ArrayList<>();
    dates.add(LocalDate.of(2011, 1, 1));
    dates.add(LocalDate.of(2014, 2, 5));
    LocalDate mostRecent = max(dates); //compiler error
}
```

Finding the max (cont.)

- **The Problem**: `LocalDate` does not implement `Comparable<LocalDate>`. Instead, the relationship to `Comparable` is the following:



`LocalDate` implements `Comparable<ChronoLocalDate>`

What is needed is a max function that accepts types `T` that implement not just `Comparable<T>`, but even `Comparable<S>` for any supertype `S` of `T`.

Here, `T` is `LocalDate`. We want max to accept a list of `LocalDate`s using a `Comparable<S>` for any supertype `S` of `LocalDate`.

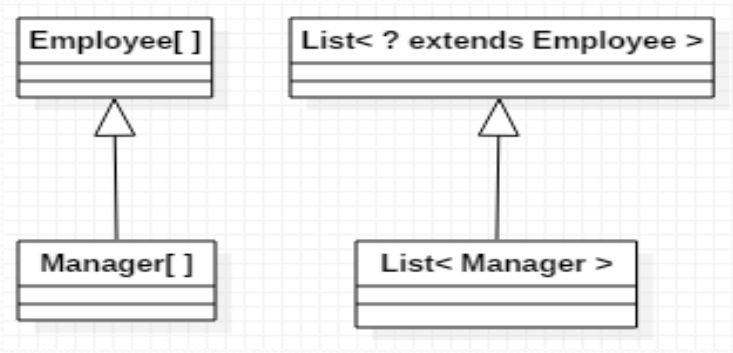
The solution lies in the use of *bounded wildcards.*

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Generic programming with generics

The `? extends` Bounded Wildcard

- The fact that generic subtyping is not covariant – as in the example that `List<Manager>` is not a subtype of `List<Employee>` – is inconvenient and unintuitive. This is remedied to a large extent with the extends *bounded wildcard*.



The ? extends Bounded Wildcard (cont.)

- The ? is a *wildcard* and the "bound" in `List<? extends Employee>` is the class Employee. `List<? extends Employee>` is called a *parametrized type with a bound.*
- For any subclass `C` of Employee, `List<C>` is a subclass of `List<? extends Employee>`.
- So, even though the following gives a compiler error:

```
List<Manager>  list1 = //... populate with managers
List<Employee> list2 = list1;   //compiler error
```

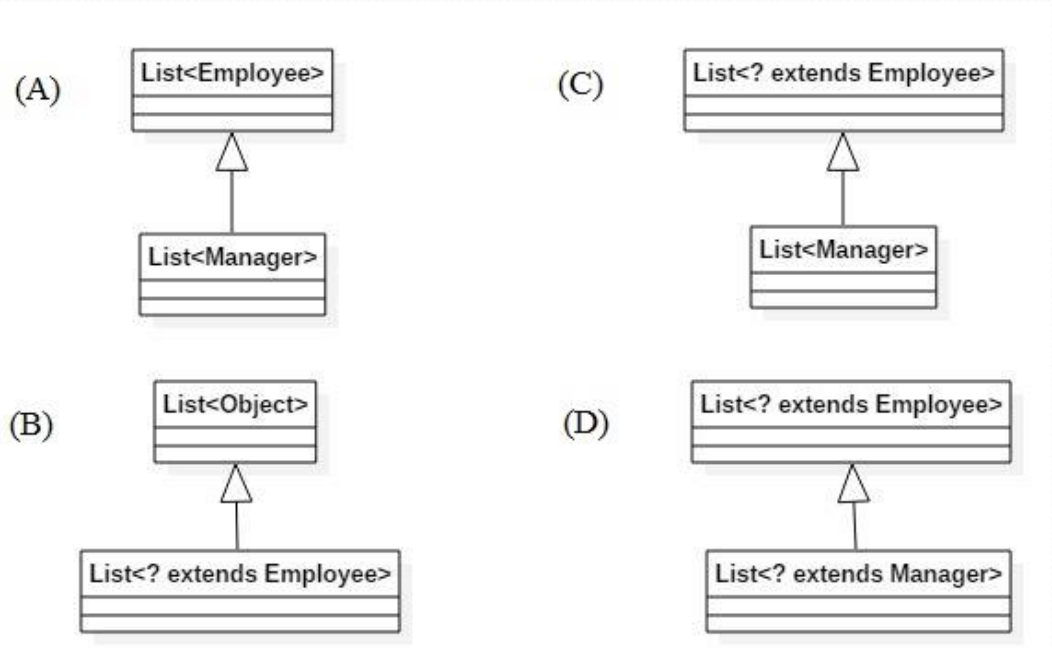
the following <u>does</u> work:

```
List<Manager> list1 = //... populate with managers
List<? extends Employee> list2 = list1;   //compiles
```

(See demo lesson11.lecture.generics.extend)

Exercise 11.3

Determine which diagrams are correct and which are not.



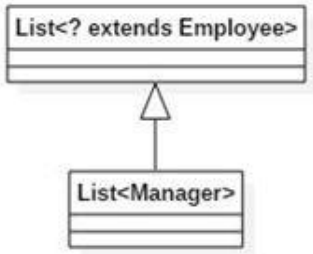
Exercise 11.3

Determine which diagrams are correct and which are not.

(A)

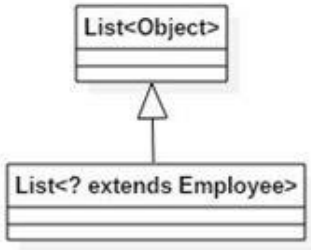
X

(C)



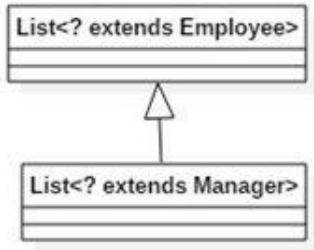
√

(B)



X

(D)



√

Applications of the ? extends Wildcard

The Java `Collection` interface has an `addAll` method:

```
interface Collection<E> {
    .  .  .
  public boolean addAll(Collection<? extends E> c);
    .  .  .
}
```

The `? extends` wildcard in the definition makes the following possible:

```
List<Employee> list1 = //....populate (here, E is Employee)
List<Manager> list2 = //... populate
list1.addAll(list2);    //OK
```

Applications of the ? extends Wildcard

If the interface method had been declared like this:

```
interface Collection<E> {
     . . .
     public boolean addAll(Collection<E> c);
     . . .
}
```

it would mean for example that `addAll` could accept only a `Collection` of *Employees*:

```
List<Employee> list1 = //….populate
List<Employee> list2 = //…populate
list1.addAll(list2);   //OK
```

BUT

```
List<Employee> list1 = //….populate
List<Manager> list2 = //…populate
list1.addAll(list2); //compiler error
```

See the demo: lesson11.lecture.generics.addall

Another Example Using addAll

```
List<Number> nums = new ArrayList<Number>();
List<Integer> ints = Arrays.asList(1, 2);
List<Double> doubles = Arrays.asList(2.78, 3.14);
nums.addAll(ints);
nums.addAll(doubles);
System.out.println(nums);   //output: [1, 2, 2.78, 3.14]
```

Here, since `Integer` and `Double` are both subtypes of `Number`, it follows that `List<Integer>` and `List<Double>` are subtypes of `List<? extends Number>`, and `addAll` may be used on `nums` to add elements from both `ints` and `doubles`.

Limitations of the extends Wildcard

When the extends wildcard is used to define a parametrized type, the type *cannot be used for adding new elements*.

Example:

Recall the `addAll` method from `Collection`:

```
interface Collection<E> {
    . . .
    public boolean addAll(Collection<? extends E> c);
    . . .
}
```

The following produces a compiler error:

```
List<Integer> ints = new ArrayList<Integer>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
//nums.add(3.14);          //compiler error
//System.out.println(ints.toString());   //output: [1, 2, 3.14]
nums.add(null);     //OK
```

Limitations of the extends Wildcard (cont.)

- The error arises because an attempt was made to insert a value in a parametrized type with `extends` wildcard parameter. With the `extends` wildcard, values can be *gotten* but not *inserted*.
- The difficulty is that adding a value to nums makes a commitment to a certain type (`Double` in this case), whereas nums is defined to be a `List` that accepts subtypes of `Number`, but *which* subtype is not determined. The value 3.14 cannot be added because it might not be the right subtype of `Number`.

NOTE: Although it is not possible to add to a list whose type is specified with the extends wildcard, this does not mean that such a list is read-only. It is still possible to do the following operations, available to any `List`:

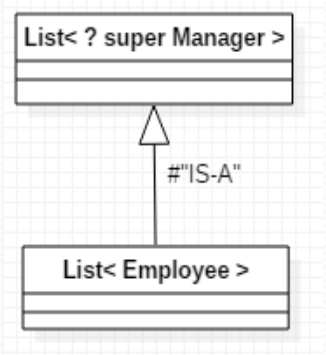
```
remove, removeAll, retainAll
```

and also execute the static methods from `Collections`:

```
sort, binarySearch, swap, shuffle
```

The ? super Bounded Wildcard

- The type `List<? super Manager>` consists of objects of any supertype of the `Manager` class, so objects of type `Employee` and `Object` are allowed.



- This diagram can be read as follows: A `List<Employee>` is a `List` whose type argument `Employee` is a supertype of `Manager`. Therefore, a `List<Employee>` IS-A `List<? super Manager>`.

Limitations of the super Wildcard

When the super wildcard is used to define a `Collection` of parametrized type, it is inconvenient to *get* elements from the `Collection`; elements can be gotten, but not typed.

Example (special case in which getting is possible):

```
List<? super Integer> test = new ArrayList<>();
test.add(5);
System.out.println(test.get(0));
```

However, if we try to assign a type to the return of the get method, we get a compiler error – the compiler has no way of knowing which supertype of `Integer` is being gotten.

```
Integer val = test.get(0);      //compiler error
Number val = test.get(0);       //compiler error
Comparable val = test.get(0);   //compiler error
Object val = test.get(0);       //OK
```

The Get and Put Principle for Bounded Wildcards

The Get and Put Principle:

Use an extends wildcard when you only *get* values out of a structure. Use a super wildcard when you only *put* values into a structure. And don't use a wildcard at all when you *both get and put* values.

Example-1 This method takes a collection of numbers, converts each to a `double`, and sums them up. The loop *gets* from an `? extends` type of Collection.

```
public static double sum(Collection<? extends Number> nums) {
    double s = 0.0;
    for(Number num: nums)
            s += num.doubleValue();
    return s;
}
```

Since `List<Integer>`, `List<Double>` are subtypes of `Collection<? extends Number>`, the following are legal:

```
List<Integer> ints = Arrays.asList(1, 2, 3);
Double val = sum(ints);   //output: 6.0

List<Double> doubles = Arrays.asList(2.78, 3.14);
Double val = sum(doubles);  //output 5.92
```

The Get and Put Principle for Bounded Wildcards (continued)

Example-2 (from the `Collections` class)

```
public static <T> void copy(List<? super T> destination,
    List<? extends T> source) {
    for(int i = 0; i < source.size();  ++i) {
          destination.set(i, source.get(i));
    }
}
```

Note that we *get* from `source`, which is typed using `extends`, and we *insert* into `destination`, which is typed using `super`. It follows that any subtype of `T` may be *gotten* from `source`, and any supertype of T may be *inserted* into `destination`.

Sample usage:

```
List<Object> objs = Arrays.asList(2, 3.14, "four");
List<Integer> ints = Arrays.asList(5, 6);
//copy the narrow type (Integer) into the wider type (Object)
Collections.copy(objs, ints);
System.out.println(objs.toString());      //output: [5, 6, four]
```

The Get and Put Principle for Bounded Wildcards (continued)

Example-3 (using `? super`) Whenever you use the `add` method for a `Collection`, you are inserting values, and so `? super` should be used.

```
public static void count(Collection<? super Integer> ints, int n) {
    for(int i = 0; i < n; ++i) {
        ints.add(i);
    }
}
```

The count method "counts" from 0 to n-1, adding these numbers to the input Collection ints.

The Get and Put Principle for Bounded Wildcards (continued)

Since super was used, the following are legal:

```
List<Integer> ints1 = new ArrayList<>();
count(ints1, 5);
System.out.println(ints1); //output: [0,1,2,3,4]

List<Number> ints2 = new ArrayList<>();
count(ints2, 5);
ints2.add(5.0);
System.out.println(ints2); //output: [0,1,2,3,4, 5.0]

List<Object> ints3 = new ArrayList<>();
count(ints3, 5);
ints3.add("five");
System.out.println(ints3); //output: [0,1,2,3,4, five]
```

- In the second call, `ints2` is of type `List<Number>` which "IS-A" `Collection<? super Integer>` (since `Number` is a superclass of `Integer`), so the `count` method can be called.
- In the third call, `ints3` is of type `List<Object>` which also "IS-A" `Collection<? super Integer>` (since `Object` is a superclass of `Integer`), so the `count` method can be called here too.
- Note that the `add` methods shown here have nothing to do with the `? super` declaration – you can add a `double` to a `List<Number>` and a `String` to a `List<Object>` for the usual reasons.

The Get and Put Principle for Bounded Wildcards (cont.)

Example-4 Improving implementation of the `max` function

We saw before that this implementation of `max` was not general enough

```
public static <T extends Comparable<T>> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

We encountered a compiler error here:

```
public static void main(String[] args) {
    List<LocalDate> dates = new ArrayList<>();
    dates.add(LocalDate.of(2011, 1, 1));
    dates.add(LocalDate.of(2014, 2, 5));
    LocalDate mostRecent = max(dates); //compiler error
}
```

- To ensure that the type `T` extends `Comparable<S>` for any supertype of `T`, we can use `? super`

```
public static <T extends Comparable<? super T>> T max(List<T> list) {
    T max = list.get(0);
    for(T i : list) {
        if(i.compareTo(max) > 0) {
            max = i;
        }
    }
    return max;
}
```

Using this version eliminates the earlier compiler error.

When You Need to Do Both Put and Get

Whenever you both put values into and get values out of the same structure, you should not use a wildcard.

```
public static double sumCount(Collection<Number> nums, int n) {
        count(nums, n);
        return sum(nums);
}
```

The collection is passed to both `sum` and `count`, so its element type must both extend `Number` (because of `sum`) and be a superclass of `Integer` (because of the `count` method). The only two classes that satisfy both requirements are `Number` and `Integer`. In this code, `Number` was chosen.

```
List<Number> nums = new ArrayList<Number>();
double sum = sumCount(nums, 5);
//sum is 10
```

Two Exceptions to the Get and Put Rule

1. In a Collection that uses the extends wildcard, null can always be added legally (null is the "ultimate" subtype)

```
List<Integer> ints = new ArrayList<>();
ints.add(1);
ints.add(2);
List<? extends Number> nums = ints;
nums.add(null);  //OK
System.out.println(nums.toString()); //output: [1, 2, null]
```

2. In a Collection that uses the super wildcard, any object of type Object can be read legally (Object is the "ultimate" supertype).

```
List<? super Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
Object ob = list.get(0);
System.out.println(ob.toString());  //output: 1
```

Main Point 2

The Get and Put Rule describes conditions under which a parametrized type should be used only for reading elements (when using a list of type ? extends T), other conditions under which the parametrized type should be used only for inserting elements (when using a list of type ? super T), and still other conditions under which the parametrized type can do both (when no wildcard is used). The Get and Put principle brings to light the fundamental dynamics of existence: there is dynamism (corresponding to Put); there is silence (corresponding to Get) and there is wholeness, which unifies these two opposing natures (corresponding to Both).

Unbounded Wildcard, Wildcard Capture, Helper Methods

1. The wildcard `?`, without the `super` or `extends` qualifier, is called the *unbounded wildcard*.

2. `Collection<?>` is an abbreviation for
   ```
   Collection<? extends Object>
   ```

3. `Collection<?>` is the supertype of all parametrized type `Collections`.

4. Important application of the unbounded wildcard involves *wildcard capture:*

Example Try to copy the 0th element of a general list to the end of the list

First Try

```
public void copyFirstToEnd(List<?> items) {
    items.add(items.get(0)); //compiler error
}
```

Compiler error arises because we are trying to add to a `List` whose type involves the extends wildcard.

Solution: Write a helper method that *captures the wildcard*.

```
public void copyFirstToEnd2(List<?> items) {
    copyFirstToEndHelper(items);
}

private <T> void copyFirstToEndHelper(List<T> items) {
    T item = items.get(0);
    items.add(item);
}
```

Notes:

A. Passing items into the helper method causes the unknown type `?` to be "captured" as the type `T`.

B. In the helper method, getting and setting values is legal because we are not dealing with wildcards in that method.

Exercise 11.4

The following code attempts to reverse the elements of a generic list, but the code does not compile. Fix the code by creating a helper method that captures the wildcard. Startup code is in the `lesson11.exercise_4 package` of the `InClassExercises` project. Test your code with the main method that has been provided.

```
public static void reverse(List<?> list) {
   List<Object> tmp = new ArrayList<Object>(list);
   for (int i = 0; i < list.size(); i++) {
      list.set(i, tmp.get(list.size()-i-1));
   }
}
```

Lesson Outline

1. Introduction to generics
2. Generic methods
3. Wildcards
4. Generic programming with generics

Generic Programming Using Generics

1. Generic programming is the technique of implementing a procedure so that it can accommodate the broadest possible range of inputs.
2. For instance, we have considered several implementations of a max function. The goal of generic programming in this case is to provide the most general possible `max` implementation.
3. See demos `lecture.generics.max.BoundedTypeVariable` and `lecture.generics.max.BoundedTypeVariable2` for a development of examples leading to the most general possible version.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Generic Programming Using Java's Generic Methods

1. Using the raw Lists of pre-Java 1.5, one can accomplish the generic programming task of swapping two elements in an arbitrary list using the signature void swap(List, int pos1, int pos2). Using this swap method requires the programmer to recall the component types of the List, and there are no type checks by the compiler.
2. Using generic Lists of Java 1.5 and the technique of wildcard capture, it is possible to swap elements of an arbitrary List with compiler support for type-checking, using the following signature:

```
<T> void swap(List<?> list, int pos1, int pos2)
```

3. *Transcendental Consciousness* is the universal value of the field of consciousness present at every point in creation.
4. *Impulses Within the Transcendental Field*. The presence of the transcendental level of consciousness within every point of existence makes individual expressions in the manifest field as rich, unique, and diversified as possible.
5. *Wholeness Moving Within Itself.* In Unity Consciousness, life is appreciated in the fullest possible way because the source of both unity and diversity have become a living reality.