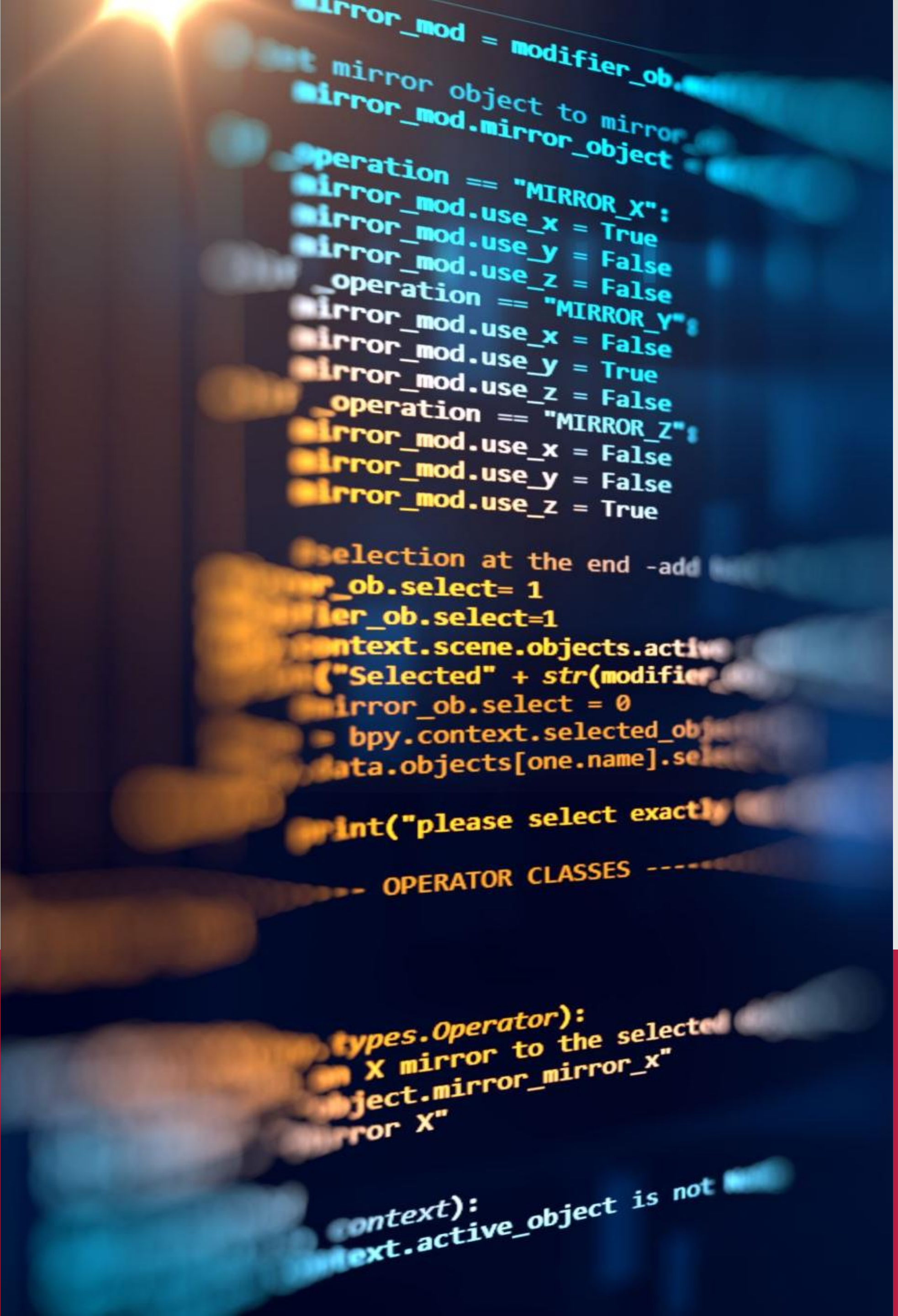


COMP 348 PRINCIPLES OF PROGRAMMING LANGUAGES

LECTURE 1-C – ADVANCED TOPICS IN JAVA

Dept. of Computer Science and Software Engineering, Concordia University

Fall 2020

Advanced Topics in Java

Abstract Classes and Interfaces

added

Method binding

- The process of associating a method definition with a method invocation is called binding

- If the method definition is associated with its invocation when the code is compiled, that is called early binding

- If the method definition is associated with its invocation when the method is invoked (at run time), that is called late binding or dynamic binding

added

Inheritance vs. Polymorphism

- Inheritance allows a base class to be defined, and other classes derived from it
  - Code for the base class can then be used for its own objects, as well as objects of any derived classes

- Polymorphism allows changes to be made to method definitions in the derived classes, and have those changes apply to the software written for the base class

- Not all programming languages support polymorphic functions by default. An example is C++ language, in which virtual functions must explicitly be specified.

- In java however, instance methods are polymorphic. Such functions use late binding.

added

Forms of inheritance

- Inheritance for Specialization: probably the most common type of inheritance.
  - the subclass is a special case of the parent class
  - Example: Control and TextBox

- Inheritance for Specification: the superclass specifies which methods should be available but doesn't give code.
  - Example: interfaces and abstract classes;

- Inheritance for Construction: the superclass is just used to provide behavior, but instances of the subclass don't really act like the superclass. This violates **substitutability**. Basically there is no logical is-a relationship between the parent class and the subclass.
  - Example: to defining Stack as a subclass of Vector. The design may be improved by defining a vector field inside the stack class.
  - Another example would be to implement a Set by inheriting from an existing List class. In general, not a good idea!

added

Forms of inheritance

- Inheritance for Generalization or Extension: subclass adds new methods (extends the behavior): Using extension, the child class does not change anything in the parent but adds to it.
  - Example: Java Properties inherited from Hashtable

- Inheritance for Limitation: in which the subclass restricts the inherited behavior. This too violates **substitutability.**
  - Example: subclassing List as a Stack class. Not a good idea! But very common.

- Inheritance for Combination: is essentially multiple inheritance. In this type of inheritance, two or more classes exist but it is not clear which one should be used as parent.
  - Example: Mouse, Touchpad, and Joystick.
  - Solution: introducing an abstract class that provides the common behavior so that it can serve as the parent to all related classes.

added

Postponing the method definition

- In order to postpone the definition of a method, Java allows an abstract method to be declared
  - An abstract method has a heading, but no method body
  - The body of the method is defined in the derived classes
- The class that contains an abstract method is called an abstract class

added

Notes on Abstract Methods

- An abstract method is like a placeholder for a method that will be fully defined in a descendent class
- It has a complete method heading, to which has been added the modifier **`abstract`**
- It cannot be private
- It has no method body, and ends with a semicolon in place of its body

```
public abstract double getPay();
public abstract void doIt(int count);
```

added

Notes on Abstract Methods

- An abstract class is a type
- You cannot create instances of an abstract class
- An abstract class can only be used to derive more specialized classes
- An abstract class constructor cannot be used to create an object of the abstract class
  - However, a derived class constructor will include an invocation of the abstract class constructor in the form of **`super`**

added

Interfaces

- An interface is something like an extreme case of an abstract class
  - However, an interface is not a class
  - It is a type that can be satisfied by any class that implements the interface
- The syntax for defining an interface is similar to that of defining a class except the word `interface` is used in place of `class`
- An interface specifies a set of methods that any class that implements the interface must have
  - It contains **method headings** and **constant definitions** only
  - It contains **no instance variables** nor any complete method definitions

added

Interfaces

- An interface serves a function similar to a base class, though it is not a base class
  - Some languages allow one class to be derived from two or more different base classes
  - This multiple inheritance is not allowed in Java
  - Instead, Java's way of approximating multiple inheritance is through interfaces

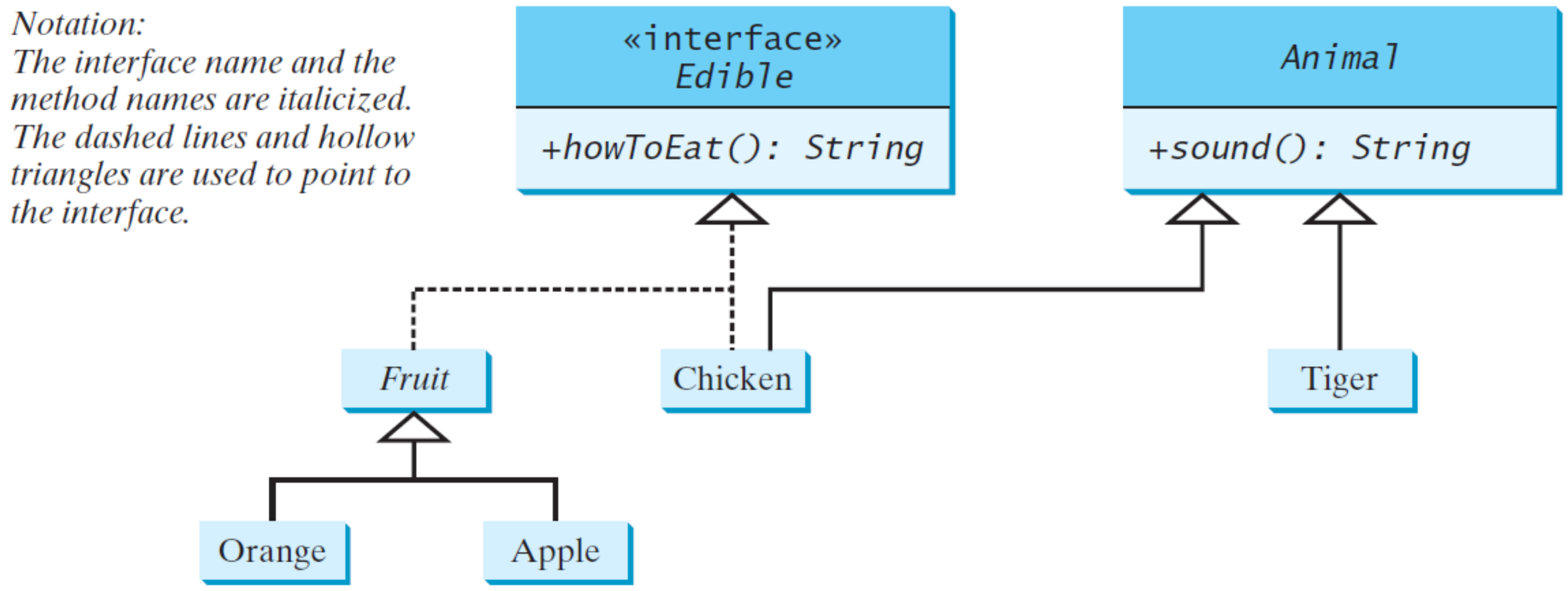
```
public interface InterfaceName {
  constant declarations;
  abstract method signatures;
}
```

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

added

Interfaces

- An interface serves a function similar to a base class, though it is not a base class



Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.
«interface»
Edible
+howToEat(): String
Animal
+sound(): String
Fruit
Chicken
Tiger
Orange
Apple

added

Omitting modifiers in interfaces

- Since all methods are public and all attributes are static final, the public and static final modifiers may be omitted.

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T1 {
  int K = 1;

  void p();
}
```

added

Overlapping Method Signatures

- If a Java class implements multiple Java interfaces, there is a risk that some of these interfaces may contain methods with the same signature (name + parameters). Since a Java class can only implement at method with a given signature once, this could potentially lead to some problems.

- The Java specification does not give any solution to this problem. It is up to you to decide what to do in that situation.

- Some other programming languages (i.e. C#) provide limited solution to this problem. See explicit interface implementation.

added

Interface static methods

- As of Java8, static methods in interfaces are also supported.

```
public interface MyInterface {

    public static void someHelperMethod(String s){
        ...
    }
}


MyInterface.print("Hello static method!");
```

- This avoids defining an additional helper classes for some public methods that are related to the interface only.

added

Interface Default Methods

- Suppose an API just adds the method to one of its published interfaces, as such all classes that implements the interface must also implement that new method. This results in a backward incompatibility.

- To overcome the above problem, as of Java 8 default methods are supported. A default method is considered as the default implemmentation of the abstact method in case the implmentation is missing in the implementing class.

added

Interface Default Method Example

- The following illustrates a default method example defined in the interface ResourceLoader.

```
public interface URIResolver {

    String resolve(URI resourcePath);
    default String resolve(URI resourcePath) {
        ...
    }
}


public class MyResolver implements URIResolver {
    // missing the resolve implementation
}
```

added

Interface Default Method Example

- Default methods may also be used to implement overloaded methods in interfaces:

```
public interface URIResolver {
    String resolve(URI resourcePath,
                   boolean local /* = false*/);

    default String resolve(URI resourcePath) {
        this.resolve(resourcePath, false);
    }
}

public class MyResolver implements URIResolver {
    /* only resolve(URI resourcePath, boolean local)
       needs to be implemented here. */
    public String resolve(URI resourcePath, boolean local) {
        ...
    }
}
```

- Note that unlike C#, Java does not support default parameters. This indeed may alternatively be used for that.

added

Interface private methods

- As of Java 9, private methods are also allowed in interfaces.
- **static private** methods may be called by other static methods.
- **non-static private** methods are also allowed and can only be called in **default** methods. private non-static methods cannot be used inside private static methods.

```
public interface MyInterface {
    public abstract void aMethod();
    public default void dMethod() {
        pMethod();
        psMethod();
    }
    public static void sMethod() {
        psMethod();
    }
    private void pMethod(){}
    private static void psMethod(){}
}
```

added

Applications of Interfaces

- Interfaces are also closely tied with Java API as well as the java language itself. For instance, try-with-resources construct may be used on classes that implement AutoCloseable interface.

```
static String firstLine(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
    // optional catch clause may follow
}
```

- Or for instance Java Iterable generic* interface, if implemented by a class, may be used in foreach statement:

```
for (String s : some_collection) {
       . . .
```

- The above interface is extensively used in java8 stream framework.

Advanced Topics in Java

Nested and Anonymous Classes

added

Nested Classes

The Java programming language allows you to define a class within another class. Such a class is called a nested class and is illustrated here:

```
class OuterClass {
    ...
    class NestedClass {
        ...
    }
}
```

Benefits of using nested classes:

- It is a way of logically grouping classes that are only used in one place
- It increases encapsulation
- It can lead to more readable and maintainable code

added

Nested Classes Modifiers

- Unlike top level classes, nested classes may have all four types of access modifiers: public, protected, private, and default (package).

- Static modifier may also be used in the same fashion as applied to other members of the outer class.

added

Non-Static Nested Classes

A normal (non-static) inner class has a connection between its objects and the outer class object that created the inner class object.

This allows an inner class definition to **reference** an **instance** variable, or **invoke** a **method** of the **outer class**.

There are certain situations, however, when an inner class must be static:

- If an object of the inner class is created within a static method of the outer class
- If the inner class must have static members

added

Static Nested Classes

As with class methods and variables, a static nested class is associated with its outer class.

And like static class methods, a static nested class cannot refer directly to **instance** variables or methods defined in its enclosing class: it can use them only through an **object reference**. In other words, a static nested class interacts with the instance members of its outer class (and other classes) just like any other top-level class.

In effect, a static nested class is behaviorally a top-level class that has been nested in another top-level class for **packaging convenience**.

added

Static vs. non-static nested classes

Static nested classes are accessed using the enclosing class name:

```
OuterClass.StaticNestedClass
```

For example, to create an object for the static nested class, use this syntax:

```
OuterClass.StaticNestedClass nestedObject =
     new OuterClass.StaticNestedClass();
```

added

Static vs. non-static nested classes

It is legal to nest inner classes within inner classes. The rules are the same as before, but the names get longer.

Given class A, which has public inner class B, which has public inner class C, then the following is valid:

```
A aObject = new A();
A.B bObject = aObject.new B();
A.B.C cObject = bObject.new C();
```

added

Compiling nested classes

- Compiling any class in Java produces a **`.class`** file named ***`ClassName.class`***

- Compiling a class with one (or more) inner classes causes both (or more) classes to be compiled, and produces two (or more) .class files

  – Such as ***`ClassName.class`*** **and** ***`ClassName$InnerClassName.class`***

added

Local Classes

- Java also supports defining local classes.
- A local class is a class that is locally defined inside a method.

```
public class MyApplication {
    void run() {
        class Local {
            void method() {
                ...
            }
        }
        Local localc = new Local();
        localc.method();
    }
    public static void main() {
        new MyApplication().run();
    }
}
```

- Note that a local class is indeed a type and is only visible within the scope of the block in which in is declared.

added

Variable Capture

Local classes capture local variables that are in the scope of the block in which the class is declared, with a restriction that only **immutable** or **final** local objects are allowed.

```
public class MyApplication {
     void run() {
          String msg = "This is a test"; // immutable
          class Local {
               void method() {
                    // msg may be captured here
                    System.out.println(msg);
               }
          }
          Local local = new Local();
          local.run();
     }
     public static void main() {
          new MyApplication().run();
     }
}
```

added

Anonymous Classes

- If an object is to be created, but there is no need to name the object's class, then an anonymous class definition can be used.
- The class definition is embedded inside the expression with the new operator.
- Anonymous classes are sometimes used when they are to be assigned to a variable of another type.
- The other type must be such that an object of the anonymous class is also an object of the other type; The other type is usually a Java interface.

added

Anonymous Classes

Despite the fact that the anonymous classes are valid types, they do not have a type name available during the compile time.

Hence the class is referred to as anonymous.

Recall anonymous arrays in Java:

```
new int[] { 1, 2, 3, 4 };
```

Example:

```
static long sum(int[] numbers) {
  long s = 0;
  for(int i : numbers) {
    s += i;
  }
  return s;
}

System.out.println(sum(new int[] { 1, 2, 3, 4 }));
```

added

Side Note: Using ellipses (varargs in Java)

The previous example may be implanted using java ellipses or varargs, as well:

```
static long sum(int... ints) {
  long s = 0;
  for(int i : ints) {
    s += i;
  }
  return s;
}

System.out.println(sum(1, 2, 3, 4));
```

added

A Threading Example

Consider the example of creating a thread in Java:

```
public class MyThreadExample {

    private static class ThreadImpl implements Runnable {
        public void run() {
                System.out.println("Child Thread");
            }
    }

    public static void main(String[] args) {
        Thread t = new Thread(new ThreadImpl());
        t.start();
        System.out.println("Main Thread");
    }
}
```

added

A Threading Example (modified)

The previous example may be rewritten using an anonymous class:

```
public class MyThreadExample {
    public static void main(String[] args) {
        Thread t = new Thread(new Runnable( {
            public void run() {
                System.out.println("Child Thread");
            }
        });
        t.start();
        System.out.println("Main Thread");
    }
}
```

added

Types of Anonymous Classes

- Anonymous Inner class that implements an interface
  - i.e. the threading example

- Anonymous Inner class that extends a class
  - The syntax is very similar, however, in case the base class has a no-arg constructor, the anonymous class must explicitly call the super constructor.
  - Note that anonymous classes **cannot have constructors** (since they are unnamed)

added

The Threading Example again

The previous example may be rewritten using an anonymous class that extends the Thread class:

```
public class MyThreadExample {
    public static void main(String[] args) {
        Thread t = new Thread() {
            public void run() {
                System.out.println("Child Thread");
            }
        };
        t.start();
        System.out.println("Main Thread");
    }
}
```

The above example is simpler compared to the Runnable implementation. This implementation uses the Thread class directly and does not require passing a Runnable implementation to it, as the child class itself override the run() method of the Thread class.

added

Passing Parameters to anonymous classes

In case the base class does not have a no-arg constructor, the anonymous class must explicitly pass parameters to the base class. An example is illustrated in the following:

```
public class Book {
    public Book(String name) { ... }
    public String description() { ... }
}

Book anonymous = new Book("Design Patterns") {
    @Override
    public String description() {
        return "Famous GoF book.";
    }
}
```

Since anonymous classes are **unnamed**, the Base class type is used to reference the anonymous type.

added

Notes on Anonymous Classes

Anonymous classes cannot have any static members except for those that are constant.

```
new Runnable() {
    static final int x = 0;
    static int y = 0; // compilation error!

    @Override
    public void run() {...}
};
```

Anonymous classes capture local variables that are in the scope of the block in which we have declared the class

```
int count = 1;
Runnable action = new Runnable() {
    @Override
    public void run() {
        System.out.println("Capturing: " + count);
    }
};
```

added

Notes on Anonymous Classes

Anonymous classes are extensively used in event listeners.

Examples:

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        ...
    }
}
```

We will see, later on, how this can be even more simplified using lambda expressions.

Advanced Topics in Java

Generics

added

Generics

- Generics are a facility of generic programming that were added to the Java programming language in 2004 within version 5.0.

- They extend Java's type system to allow a type or method to operate on objects of **various types** while providing **compile-time type safety**.

- Generics use **type checking** during **compile time**; however **errors** may still happen **during runtime**.

added

Motivation

- Consider the following example:

```
List v = new ArrayList(); // to contain integers only
v.add("test"); // OK
Integer i = (Integer)v.get(0); // Run time error
```

- Using Generics, such error may be prevented during the compile time:

```
List<Integer> v = new ArrayList<String>();
v.add("test"); // compiler error
Integer i = (Integer)v.get(0);
```

- Alternatively:

```
List<String> v = new ArrayList<String>();
v.add("test"); // OK
Integer i = (Integer)v.get(0); // compiler error
```

added

Example

- Let us use the following non-generic class:

```
class DemoClass {
    private Object t;
    public void set(Object t) { this.t = t; }
    public Object get() { return t; }
}
```

- The above example may be rewritten using generics:

```
class DemoClass<T> {  // T stands for "Type"
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

added

Generics vs. C++ templates

- Java generics **differ** from C++ templates.
- Java generics generate only **one compiled version** of a generic class or function regardless of the number of parameterizing types used.
- Furthermore, the Java run-time environment does not need to know which parameterized type is used because the type information is validated at compile-time and is not included in the compiled code.

added

Generics use Type Erasure

- Generics are checked at compile-time for type-correctness. The generic type information is then removed in a process called type erasure. For example, List<Integer> will be converted to the non-generic type List, which ordinarily contains arbitrary objects. The compile-time check guarantees that the resulting code is type-correct.

```
List<String> v = new ArrayList<String>();

List v = new ArrayList();
```

- Because of type erasure, type parameters **cannot** be determined at **run-time**. For example, when an ArrayList is examined at runtime, there is no general way to determine whether, before type erasure, it was an ArrayList<Integer> or an ArrayList<String>.

added

Generics use Type Erasure

- Another consequence of type erasure is that generic class cannot extend the Throwable class in any way, directly or indirectly. For example the following code does NOT compile:

```
public class GenericException<T> extends Exception
   . . .

try {
    throw new GenericException<Integer>();
}
catch(GenericException<Integer> e) {
    System.err.println("Integer");
}
catch(GenericException<String> e) {
    System.err.println("String");
}
```

- Due to type erasure, the runtime will not know which catch block to execute, so this is prohibited by the compiler.

added

Generic parameters cannot be instantiated

- Consequently, instantiating a Java class of a parameterized type is impossible because instantiation requires a call to a constructor, which is unavailable if the type is unknown. For instance the following code does NOT compile:

```
<T> T instantiateElementType(List<T> arg) {
     return new T(); //compiler error
}
```

- Similarly, generic arrays cannot be defined.

```
new T[] // This is not allowed either
```

added

Generics and Inheritance

- Generics may be used anywhere within the class hierarchy, including interfaces. An example is given in the following:

```
//Generic interface definition that uses two generic types
interface DemoInterface<T1, T2>
{
   T2 doSomeOperation(T1 t);
   T1 doReverseOperation(T2 t);
}

//A class implementing generic interface
class DemoClass implements DemoInterface<String, Integer>
{
   public Integer doSomeOperation(String t)
   {
      //some code
   }
   public String doReverseOperation(Integer t)
   {
      //some code
   }
}
```

added

Constructors in Generics

- In defining generics constructors, the generic parameters are not used as part of the constructor name:

```
class Dimension<T>
{
   private T length;
   private T width;
   private T height;

   //Generic constructor
   public Dimension(T length, T width, T height)
   {
      super();
      this.length = length;
      this.width = width;
      this.height = height;
   }
}
```

added

Generics parameters are reference types

- Note that generics are reference types. That means value types may not be used in generics.

```
List<int> ids = new ArrayList<>(); // Not allowed
```

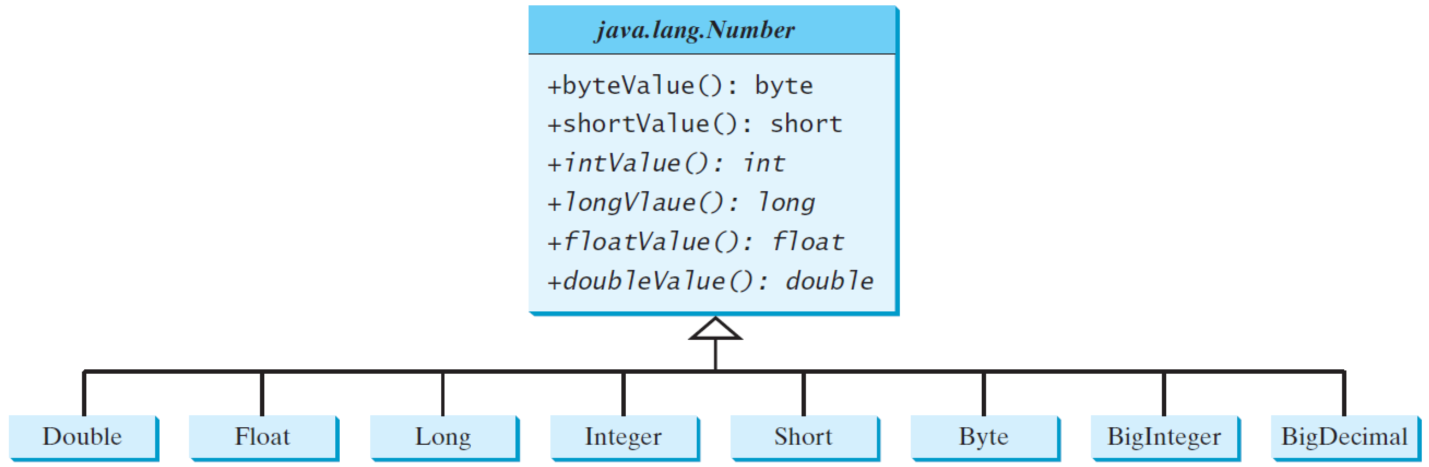
- To support value types, java provide automatic boxing and unboxing using wrapper classes:

```
List<Integer> ids = new ArrayList<>(); // Allowed, no need to repeat


ids.add(12);          // automatic boxing using Integer class
int i = ids.get(0)   // automatic unboxing to int value type
```

added

Wrapper Classes in java



added

Generics and static declarations

- Due to the type erasure, static parameter types are NOT supported by generics:

```
private static T member; // This is not allowed
```

- Static methods however are fully supported:

```
public static <T> int countAllOccurrences(T[] list, T item) {
    int count = 0;
    if(item == null) {
        for(T listItem : list)
            if(listItem == null)
                count++;
    }
    else {
        for(T listItem : list)
            if(item.equals(listItem))
                count++;
    }
    return count;
}
```

Note that the type T above is used int the static method only and has no relation to the enclosing class.

Advanced Topics in Java

Variance, Covariance, and Contravariance

added

Variance, Covariance, and Contravariance

Variance refers to how subtyping between more complex types relates to subtyping between their components.

For instance, given Cat is a subtype of Animal, variance is referred to how a list of Cats relate to a list of Animals, Or how should a function that returns Cat relate to a function that returns Animal.

added

Variance, Covariance, and Contravariance

Within the type system of a programming language, a typing rule or a type constructor is:

- covariant if it preserves the ordering of types (≤), which orders types from more specific to more generic; also known as sub-typing
- contravariant if it reverses this ordering;
- bivariant if both of above apply
- variant if covariant, contravariant or bivariant;
- invariant or nonvariant if not variant.

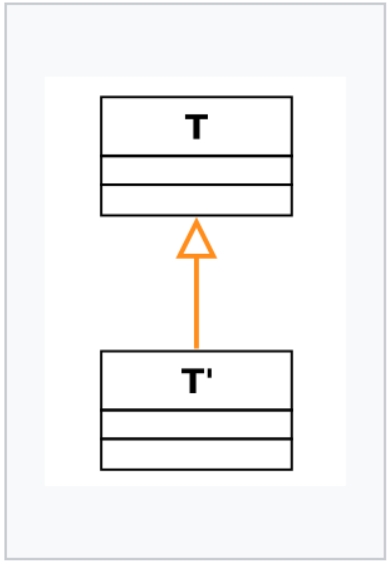
Example:

- covariant: a Cat[] is an Animal[];
- contravariant: an Animal[] is a Cat[];
- invariant: an Animal[] is not a Cat[] and a Cat[] is not an Animal[].

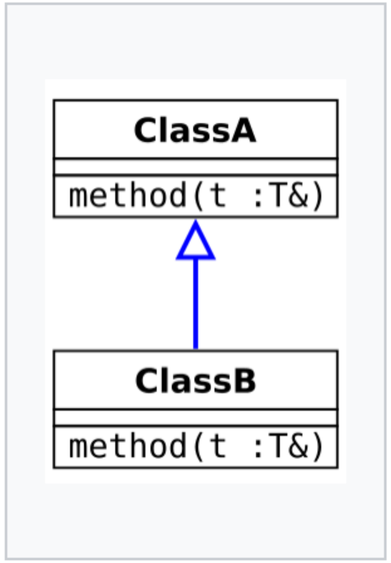
added

Variance, Covariance, and Contravariance

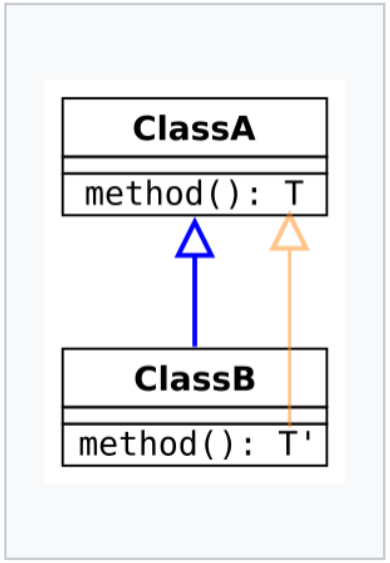
Variance and method overriding: overview



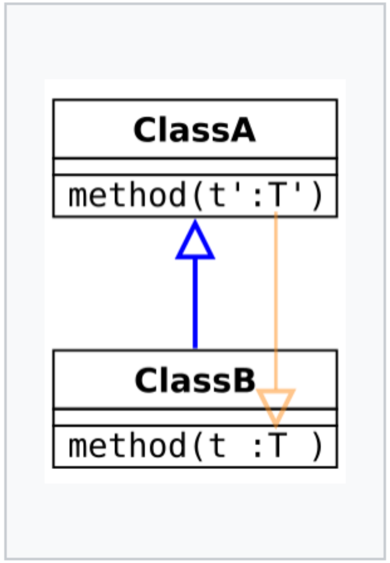
Subtyping of the parameter/return type of the method.



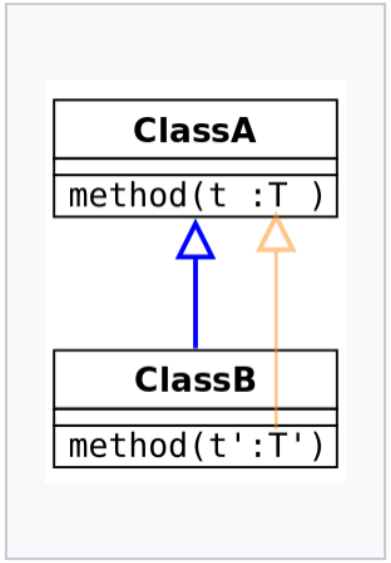
Invariance. The signature of the overriding method is unchanged.



Covariant return type. The subtyping relation is in the same direction as the relation between ClassA and ClassB.



Contravariant parameter type. The subtyping relation is in the opposite direction to the relation between ClassA and ClassB.



Covariant parameter type. Not type safe.

https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)

added

Java Arrays are covariant

In Java, Arrays are said to be covariant which basically means that, given the subtyping rules of Java, an array of type T[] may contain elements of type T or any subtype of T. For instance:

```
Number[] numbers = newNumber[3];
numbers[0] = newInteger(10);
numbers[1] = newDouble(3.14);
```

Additionally, the subtyping rules of Java also state that an array S[] is a subtype of the array T[] if S is a subtype of T.

```
Integer[] ia = {1, 2, 3};
Number[] na = ia;
```

Now, what if one does the following?

```
na[0] = 3.14;
```

added

Heap Pollution

What if we try the following?

```
Integer[] ia = {1, 2, 3};
Number[] na = ia;
na[0] = 3.14;
```

The Above complies but it causes ArrayStoreException during runtime. The issue is referred to as the heap pollution.

In the Java programming language, heap pollution is a situation that arises when a variable of a parameterized type refers to an object that is not of that parameterized type.

added

Covariance and functions

Consider the following function

```
static long sum(Number[] numbers) {
  long s = 0;
  for(Number number : numbers) {
    s += number.longValue();
  }
  return s;
}
```

The above function may be used to calculate the sum of arrays of numbers, integers, longs, etc.

```
Integer[] myInts = {1,2,3,4,5};
Long[] myLongs = {1L, 2L, 3L, 4L, 5L};
Double[] myDoubles = {1.0, 2.0, 3.0, 4.0, 5.0};
System.out.println(sum(myInts));
System.out.println(sum(myLongs));
System.out.println(sum(myDoubles));
```

Advanced Topics in Java

Wildcards and Generics

added

Wildcards and Generics

Java generics use type erasure. Therefore the type information is not only discarded by the compiler and therefore not available at run time.

Consider the following example:

```
List<Integer> myInts = newArrayList<Integer>();
myInts.add(1);
myInts.add(2);
```

Unlike arrays, Java Generics are invariant. For instance, the following is illegal“:

```
List<Number> nums = myInts; //compiler error
```

added

Wildcards and Generics

Consider the following example:

```
static long sum(List<Number> numbers) {
  long s = 0;
  for(Number number : numbers) {
    s += number.longValue();
  }
  return s;
}
```

The above function may NOT be used, because generics are not covariant.

```
List<Integer> myInts = Arrays.asList(1,2,3,4,5);
List<Long> myLongs = Arrays.asList(1L, 2L, 3L, 4L);
List<Double> myDoubles = Arrays.asList(1.0, 2.0, 3.0);
System.out.println(sum(myInts)); // compiler error
System.out.println(sum(myLongs)); // compiler error
System.out.println(sum(myDoubles)); // compiler error
```

added

Wildcards and Generics

Java wildcards provide a solution to this. Instead of using a type T as the type argument of a given generic type, we can use a wildcard declared as ? extends T, where T is a known base type.

```
static long sum(List<? extends Number> numbers) {
  long s = 0;
  for(Number number : numbers) {
    s += number.longValue();
  }
  return s;
}
```

The above function may NOT be used, because generics are not covariant.

```
List<Integer> myInts = Arrays.asList(1,2,3,4,5);
List<Long> myLongs = Arrays.asList(1L, 2L, 3L, 4L);
System.out.println(sum(myInts)); // OK
System.out.println(sum(myLongs)); // OK
```

added

Side Note: Static Imports

Using **static import**, one may rewrite the code as:

```
import static java.util.Arrays.asList;
.
.
.

List<Integer> myInts = asList(1,2,3,4,5);  // calls Arrays.asList() method
.
.
.
```

This sometimes is very useful if a common static method is extensively used in a class file.

Note: static imports may not be compared to “include” feature in other programming languages. In java, it is purely used as a short cut for coding, and the imported methods are purely called and are not added to the class at all. Static imports are used to translate function calls during the compile time.

added

Side Note: Static Imports

Static import is a feature introduced in the Java 5 that allows members (fields and methods) which have been scoped within their container class as public static, to be used in Java code without specifying the class in which the field has been defined.

When defined in the class

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Considering a circle with a diameter of 5 cm, it has");
        System.out.println("a circumference of " + (PI * 5) + " cm");
        System.out.println("and an area of " + (PI * pow(2.5, 2)) + " sq. cm");
    }
}
```

Question: Can we apply this on System.out.println as well?

added

Side Note: Static Imports

Static imports are heavily used in JUnit.

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;

public class MyTests {
   @Test
   public void myTestProc() {
         MyClass tester = new MyClass(); // MyClass is being tested
         // assert statement
         assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");
   }
}
```

The @Test above is an annotation class which is discussed at the end of this chapter.

added

Types of Wildcards

Java supports three types of wildcards:

- The unbounded wildcard type is specified using the wildcard character (?)

```
Collection<?>
```

- The upper bounded wildcard type restricts the unknown type to be a specific type or a subtype of that type and is represented using the extends keyword:

```
List<? extends Number>
```

- A lower bounded wildcard is expressed using the super keyword, followed by its lower bound:

```
Comparator<? super String>
```

added

Wildcards and Covariance

With covariance we can read items from a structure, but we cannot write anything into it.

```
List<? extends Number> na = new ArrayList<Integer>();
// List<? extends Number> na = new ArrayList<Float>();
// List<? extends Number> na = new ArrayList<Double>();

Number n = na.get(0); // OK

na.add(45L); //compiler error
```

added

Wildcards and Contravariance

For contravariance we use a different wildcard called ? super T, where T is our base type. With contravariance we can do the opposite. We can put things into a generic structure, but we cannot read anything out of it.

```
// List<? super Number> na;

Number n = na.get(0); //compiler error
```

Contravariance may be used as output parameters.

added

Covariance and Contravariance in parameters

Reading and Writing to generics may be illustrated using the following example:

```
static void copy(List<? extends Number> src,
                 List<? super Number> dest) {
    for(Number n : src) {
        dest.add(n);
    }
}
```

```
List<Integer> ia = Arrays.asList(1, 2, 3, 4);
List<Double> da = Arrays.asList(3.14, 6.28);
List<Object> oa = new ArrayList<Object>();
copy(ia, oa); // OK
copy(da, oa); // OK
```

added

More on Wildcards

- Wildcard and multiple types:

```
Pair<String, ?>
```

- Wildcards and multiple extensions (inheritance)

```
List<? extends Edible & Animal>
```

Note the use of & character (as opposed to using , in extending multiple interfaces).

Question: What does the following mean?

```
List<? extends Edible, Animal>
```

Advanced Topics in Java

Java Annotations

added

Java Annotations

- In the Java, an annotation is a form of syntactic metadata that can be added to Java source code.

- Classes, methods, variables, parameters and Java packages may be annotated.

- Java annotations can be **read** from **source files** (i.e. Javadoc) as well as be **embedded** in after they are **compiled**.

- Therefore annotations can be retained by the Java virtual machine at run-time and read via reflection.

- Additionally, It is possible to create meta-annotations out of the existing ones in Java.

added

Java Annotations

- The @ symbol is used to specify annotations.

- Java defines a set of annotations that are built into the language, examples: @Override, @Deprecated, @SuppressWarnings, …

- Custom annotations are supported as well.

added

Applications of Annotations

- Annotations are the meta data that are attached to the classes, methods, etc. This meta data may be retrieved during the runtime. The process is referred to as reflection.
- Annotations and their attribute values may be examined during the runtime which enables us with code injection during runtime.
- Java Persistence API (JPA) is an example of using annotations for code injection. Another example is aspect-oriented programming using spring AOP.
- Annotations may have different names in other languages. For instance C# supports the very concept by Attributes. The concept is referred to as attribute programming.

added

Defining Custom Annotations

- Annotation type declarations are similar to normal interface declarations. An at-sign (@) precedes the interface keyword.

```
public @interface MyAnnotation {
}
```

- Once declared and imported, it can simply be used in other classes. For instance, the above annotation is used to annotate a method:

```
@MyAnnocation
public void toggle() {
}
```

added

Annotations and Reflection

- Annotations are the meta data that are attached to the classes, methods, etc. This meta data may be retrieved during the runtime. The process is referred to as reflection.
- For example suppose you want to examine if a Subject class is annotated with an MyAnnotation or not.

```
@MyAnnocation
Public class MyAnnocation {
    ...
}
```

- This may be done using the following code sample:

```
import java.lang.annotation.Annotation;
import java.lang.reflect.Method;

Class<AnnotatedClass> c = AnnotatedClass.class;
if (c.isAnnotationPresent(MyAnnotation.class)) {
    Annotation ac = c.getAnnotation(MyAnnotation.class);
    MyAnnotation a = (MyAnnotation)ac;
    // read values of the attributes of the annotation…
```

added

Annotations and Reflection

- In the previous example, the target of the annotation was a class.
- Java annocation framework provides controlling the target of a custom annotation through some other built in annotations.

```
@Retention(RetentionPolicy.RUNTIME) // Make this annotation accessible at
                                     // runtime via reflection.
@Target({ElementType.METHOD})       // This annotation can only be applied
                                     // to class methods.
public @interface MyAnnocation {

}
```

added

Annotations Properties

- Annotations may have properties. Consider the following example:

```
@Author(first = "John", last = "Doe")
Book book = new Book();
```

- In the above example, a local variable book is annotated with @Author with specific first and last time. The above properties are defined using interface methods:

```
public @interface Author {
      String first();
      String last();
}
```

Advanced Topics in Java

Lambda Expressions and Functional Interfaces

added

Functional Interfaces and SAMs

- A class with Single Abstract Method is called SAM.
- A functional interface is an interface with single abstract method.
- An interface with more than one abstract method is not a functional interface unless all other methods have **default implementation**.

```
public interface MyInterface {
     void printIt(String text);
}
```

- Functional Interfaces essentially simulate a callbacks, a mechanism that does not exist in java. This is however supported differently in other languages (see delegates in C#).

added

Example

- The following interface is indeed a valid functional interface:

```
public interface MyInterface {
     void printIt(String text);

     default public void printUtf8To(String text,
               OutputStream outputStream){
        try {
            outputStream.write(text.getBytes("UTF-8"));
        } catch (IOException e) {
            throw new RuntimeException(
               "Error writing String as UTF-8 to OutputStream",
                e);
        }
    }

     static void printItToSystemOut(String text){
        System.out.println(text);
    }
}
```

added

Lambda Calculus

- The lambda calculus was introduced in the 1930s by Alonzo Church as a mathematical system for defining computable functions.

- The lambda calculus serves as the computational model underlying functional programming languages such as Lisp, Haskell, and Ocaml.

- Lambda expressions have been incorporated into many widely used programming languages like C++ and now very recently Java 8.

added

What is a Lambda Expression?

- In computer programming, an anonymous function (function literal, lambda abstraction, or lambda expression) is a function definition that is not bound to an identifier.

- A Java 8 lambda is basically a method in Java **without** a **declaration** usually written as

(parameters) -> { body }

- Examples of lambda expressions are given in the following:

```
x -> x * x // a lambda with one parameter

// typed parameters with code block
(int x, int y) -> { return x + y; }

() -> x // a lambda with no parameter but returning a value
```

added

Lambda Expressions and Anonymous Classes

- Lamda Expressions are an alternative way of creating anonymous classes.
- Given a functional interface or a SAM base class, a lamda expression may simply replace the anonymous class.
- Suppose we want to implement the following interface:

```
public interface MyEventConsumer {
    public void consume(Object event);
}
```

added

Lambda Expressions and Anonymous Classes

- Using anonymous classes, we may write:

```
MyEventConsumer consumer = new MyEventConsumer() {
    public void consume(Object event){
        System.out.println(event.toString() + " consumed");
    }
};
```

- The above example may be simply written using lambda expression:

```
MyEventConsumer consumer = (Object e) -> {
        System.out.println(e.toString() + " consumed"); };
```

- The above lambda may even be more simplified as:

```
MyEventConsumer consumer =
        e -> System.out.println(e.toString() + " consumed");
```

Note that the type inference is automatically done at the compile type.

added

Examples of Lambda Expressions

- Returning values:

```
(param) -> {
        System.out.println("param: " + param);
        return "return value"; }




(a1, a2) -> { return a1 > a2; } // is greater than?

(a1, a2) -> a1 > a2    // is greater than (simplified version)
```

added

Lambdas are Objects

- Lambda expressions are replacements for anonymous classes that implement functional interfaces or SAMs.

- They are indeed anonymous types and are complied to one.

- As a result, Objects of the related types may hold a reference to lambdas.

  Example:

```
public interface MyComparator {
       public boolean isgreaterthan(int a1, int a2);
}

MyComparator myComparator = (a1, a2) -> return a1 > a2;
boolean result = myComparator.compare(2, 5);
```

added

Lambdas and Variable Capture

- Similar to anonymous and local classes, local variables may also be captured by lambda expressions:

```
String myString = "Test"; // immutable

MyFactory myFactory = (chars) -> {
        return myString + ":" + new String(chars); };
```

- However, only **immutable** or **final** objects are allowed.

```
// the following causes compiler error since Object a is not final.
/* final */ ClassA a = new ClassA();
MyLambda l = () -> "Capturing: " + a.toString();
```

Advanced Topics in Java

Java Stream API

added

Stream Processing

- Stream represents a sequence of objects from a source, which supports aggregate operations.
- Stream processing is a computer programming paradigm, equivalent to dataflow programming, event stream processing, and reactive programming, that allows some applications to more easily exploit a limited form of parallel processing.
- Most programming languages, such as Java or C++, provide supports for stream processing. This is not to be confused with I/O Streams.

added

Java Stream API

- Java Stream API released as of Java 8, provides such a support.

- The idea of Java Stream API was based on the implementation of Microsoft .Net enumerable queries that had been used in Microsoft **Language Integrated Query (LINQ)**.

- Java Stream API is defined in java.util.stream package.

- The stream package defines classes and interfaces to support functional-style operations on streams of elements, such as map-reduce transformations on collections.

added

Java Stream API

- Stream API using functional programming and provides sets of methods that receive functional interfaces as parameters.

- Any iterable class may simply be transformed into a stream object which inherently supports the above methods.

- Most iterable classes, however, may directly be used as streams.

added

Deferred Execution and Lazy Evaluation

- Deferred execution means that the evaluation of an expression is delayed until its realized value is actually required. Deferred execution can greatly improve performance when you have to manipulate large data collections, especially in programs that contain a series of chained queries or manipulations.

- In lazy evaluation, a single element of the source collection is processed during each call to the iterator. This is the typical way in which iterators are implemented.

- In eager evaluation, the **first call** to the iterator will result in the **entire collection** being processed. For example, applying an order-by operation has to sort the entire collection before returning the first element.

added

Parallelism

- Both Java Stream Processing and C# LINQ provide support for parallel stream processing.
- This is extremely useful in processing big pipelines of data which may be executed over multiple cores.

added

Stream Operations

- Constructing streams:
  - done via a call to stream() or parallelStream() methods.
- forEach():
  - Iterates through every element of the stream
- map():
  - used to map each element to a desired result
- filter():
  - to filter data based on selection criteria (similar to a where clause)
- limit():
  - To limit the resulting data to certain numbers (similar to a top clause)
- sorted():
  - Used to sort the data in the stream (similar to order by)
- Collectors:
  - done via a call to collect() or groupingBy(), … (similar to group by)
- Statistics:
  - builtin class to provide statistics (such as max, min, sum, average, …)

added

Using the forEach() method

- Below is an example of using forEach() method:

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

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

- The lambda expression x -> System.out.println(x) defines an anonymous class with one parameter named x of type Integer and no return.

- Output:

```
1
2
3
```

added

Using the forEach() method

- Another example using forEach() method:

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

intSeq.forEach(x -> {
   x += 2;
   System.out.println(x);
});
```

- Note that braces are needed to enclose a multiline body in a lambda expression.

- Output:

```
3
4
5
```

added

Using the forEach() method

- Another example using forEach() method:

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

intSeq.forEach(x -> {
   int y = x * 2;
   System.out.println(y);
});
```

- Just as with ordinary functions, you can define local variables inside the body of a lambda expression

- Output:

```
2
4
6
```

added

Using the forEach() method

- Defining explicit types in lambda often prevents unintended errors.

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

intSeq.forEach((Integer x -> {
   x += 2;
   System.out.println(x);
});
```

- Output:

```
2
4
6
```

added

Method references

- Method reference may be compared to the notion of a pointer to function (i.e. a callback, or a delegate in C#).

- Although the notion a function pointer does not exist in java, the concept is somehow supported through functional interfaces.

- As such, java 8 provides using method references, which in turn is translated into a lambda equivalent.

- Method reference are represented using scope resolution operation (::), a notation borrowed from the C++ language.

added

Method references

- Method references can be used to pass an existing function in places where a lambda is expected
- The signature of the referenced method needs to match the signature of the functional interface method
- The following table summarizes the different types of method references.

Method Reference Type	Syntax	Example
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

added

Method reference example

- Consider the following lambda expression:

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

- The above lambda expression may be rewritten more concisely using a method reference as:

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

added

Chaining methods

- Method chaining is a common syntax for invoking multiple method calls in object-oriented programming languages.
- Each method returns an object, allowing the calls to be chained together in a single statement without requiring variables to store the intermediate results.
- An example is given below:

```
int sum = widgets.stream()
                 .filter(w -> w.getColor() == RED)
                 .mapToInt(w -> w.getWeight())
                 .sum();
```

- The above is an example of filter and map operations.
- Since almost all methods in Stream API return a stream object, the methods may be chained together.
- A counter example to the above is the use of forEach() method which has a void return type and therefore cannot not be further used in methods chaining .

added

Building a map-reduce

- A map-reduce program is composed of
  - a map procedure, which performs **filtering** and **sorting** (such as sorting students by first name into queues, one queue for each name),
  - and a reduce method, which performs a **summary** operation (such as counting the number of students in each queue, yielding name frequencies).
- Map-reduce programming is extensively used in big-data processing.

- Below is a simple example of a map reduce implementation:

```
double avgsal =
      employees.stream().mapToDouble(e -> e.getSalary()).
            average().getAsDouble();
```

added

A group-by example

- Below is an example of using map reduce to count the number of occurrence per item.

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

Map<Integer, Long> result =
    numbers.stream().collect(
        Collectors.groupingBy(
            Function.identity(), Collectors.counting()));

result.entrySet().stream()
    .sorted(
        Map.Entry.<Integer, Long>comparingByValue().reversed())
    .forEachOrdered(
          e -> System.out.println(
              "Number " + e.getKey() + " occurred " +
              e.getValue() + " times"));
```

- Output:

```
Number 1 occurred 3 times
Number 2 occurred 2 times
Number 3 occurred 1 times
```

added

A group-by example explained

```
Map<Integer, Long> result =
    numbers.stream().collect(
        Collectors.groupingBy(
            Function.identity(), Collectors.counting()));
```

- numbers.stream() constructs a stream that is ready to be processed.
- The collect() method receives a group by method reference. A built-in group by method is used: Collectors.groupingBy()
- The Collectors.groupingBy() method requires two lambda arguments.

added

A group-by example explained

```
Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

- The first argument of the groupingBy collector is a map function:
  - identity function in this example, i.e. x -> x

- The second argument of the groupingBy collector is a reduce function:
  - Count function, an aggregate function specified as

```
public static <T> Collector<T, ?, Long> counting()
```

That received has three generic types:

- the type of input elements (T),
- the mutable accumulation type of the reduction operation (?),
- and The result type of the reduction operation (Long)

added

A group-by example explained

```
result.entrySet().stream()
    .sorted(
        Map.Entry.<Integer, Long>comparingByValue().reversed())
    .forEachOrdered(
          e -> System.out.println(
              "Number " + e.getKey() + " occurred " +
              e.getValue() + " times"));
```

- The output result is further processed by sorting out the key-value pairs of the resulting map.
- The sorting is performed by calling the sorted() method, which receives a Comparator.

added

A group-by example explained

```
result.entrySet().stream()
    .sorted(
        Map.Entry.<Integer, Long>comparingByValue().reversed())
    .forEachOrdered(
          e -> System.out.println(
              "Number " + e.getKey() + " occurred " +
              e.getValue() + " times"));
```

- This code uses a builtin comparator that is provided by comparingByValue() method from the Map.Entry generic class.
- The result is then reversed in order to maintain the **ascending** order.
- And finally each item is printed out by using forEachOrdered() method.

added

forEach vs. forEachOrdered

```
result.entrySet().stream()
    .sorted(
        Map.Entry.<Integer, Long>comparingByValue().reversed())
    .forEachOrdered(
          e -> System.out.println(
              "Number " + e.getKey() + " occurred " +
              e.getValue() + " times"));
```

- The use of forEachOrdered() is preferred over forEach() as it **guarantees** the order, especially in parallel processing.

added

Acknowledgements

- https://dzone.com/articles/what-are-nested-classes-and
- http://tutorials.jenkov.com/java/interfaces.html
- https://www.geeksforgeeks.org/wildcards-in-java/
- https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html
- https://dzone.com/articles/covariance-and-contravariance
- https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html
- https://www.geeksforgeeks.org/anonymous-inner-class-java/
- https://www.baeldung.com/java-anonymous-classes
- https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)
- https://en.wikipedia.org/wiki/Generics_in_Java
- http://tutorials.jenkov.com/java/lambda-expressions.html
- http://web.engr.oregonstate.edu/~budd/Books/oopintro3e/info/slides/chap08/slide15.htm
- https://en.wikipedia.org/wiki/Java_annotation
- https://docs.oracle.com/javase/6/docs/technotes/guides/language/annotations.html
- https://www.baeldung.com/java-8-streams
- https://stackify.com/streams-guide-java-8/
- https://www.tutorialspoint.com/java8/java8_streams.htm
- https://en.wikipedia.org/wiki/Method_chaining
- https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html