9a

Lambda Expressions

OBJECTIVES

In this lecture you will learn:

- **The syntax of Lambda expressions.**
- **Passing values to Lambda expressions**
- **Referencing Lambda expressions by Functional interfaces**
- **Different kinds of method references**
- **Differences between Lambda expressions and anonymous inner classes**
- **Using standard types of Functional interfaces in typical cases**
- **Using default and static methods in interfaces**

9a.1 Introduction

- **Prior to Java SE 8, Java supported three programming paradigms- procedural programming, object- oriented programming and generic programming. Java SE 8 adds functional programming.**
- **The new language and library capabilities that support functional programming were added to Java as part of Project Lambda.**
- **In this lecture we learn many examples of functional programming, often showing simpler ways to implement tasks that you programmed in earlier lectures (http://www.java2s.com/Tutorials/Java_Lambda/index.htm)**

9a.2 Lambda Expressions

The **biggest language change** in Java 8 **is the introduction of lambda expressions-** a compact way for passing a method as an argument to another method.

- **Concise syntax-** More succinct and clear than anonymous inner classes
- **Remove deficiencies with anonymous inner classes (** take more space, difficult to read, naming confusions about the `this` and other references, no non-final variables, hard to optimize)
- **Convenient for new Streams API**

```
shapes.forEach(s -> s.setColor(Color.RED));
```

- **Similar constructs used in other languages**
  – Callbacks, closures, map/reduce pattern

9a.2 Lambda Expressions

In the following example, we're creating a new object that provides an implementation of the **ActionListener** class. This interface has a **single method**, **actionPerformed**, which is called by the button instance when a user actually clicks the on-screen button. The **anonymous inner class** provides the implementation of this method.

```
button.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent event) {
                System.out.println("button clicked");
        }
});
```

9a.2 Lambda Expressions

This is actually an example of using *code as data*, we're giving the button an object that represents an action. **Anonymous inner classes** were designed for this purpose.

However, these classes take a lot of space to write and they are fairly hard to read because their code obscures the programmer's intent. In fact, in such cases we don't want to pass in an object, what we really want to do is pass in some behavior.

The business logic is interwined with the technical garbage. The core logic is very much tied to the implementing class

9a.2 Lambda Expressions

- **Anonymous inner classes are inconvenient means** to support dozens of such requirements. It would require to create the additional logic path using a control statement (if-else or switch) or create a new anonymous class for each piece of logic. **Tightly coupling your business logic to its implementing class** is asking for trouble-especially, if we have frequently changing business rules.
- **Lambda expressions** are a better way to make it work.

9a.2 Lambda Expressions

A **lambda expression** associates the required behavior with a button click as follows

```
button.addActionListener(event ->
    System.out.println("button clicked"));
```

Instead of passing in an object that implements an interface, we're **passing in a block of code-** a **function without a name**.

`event` is the **name of a parameter**, the same **parameter** as in the anonymous inner class example.

`->` separates the parameter from the **body of the lambda expression**, which is just some code that is run when a user clicks our button.

9a.2 Lambda Expressions

Summary

Replace this anonymous class

```
new SomeInterface() {
     @Override
      public SomeType someMethod(args) {
                                body
        }
}
```

with this lambda expression

```
(args) -> { body }
```

9a.2 Lambda Expressions

Old style

```
Arrays.sort(testStrings,
                     new Comparator<String>() {
     public int compare(String s1, String s2)
     {
           return(s1.length() - s2.length());
     }
});
```

New style

```
Arrays.sort(testStrings,
              (String s1,String s2)->
                  s1.length()-s2.length());
```

9a.2 Lambda Expressions

- Another **difference** between **this example** and the **anonymous inner class** is how we declare the variable **`event`**. Previously, we needed to **explicitly provide its type**, **`ActionEvent`** event.
- In this example, we **haven't provided the type** at all, yet this example still compiles. What is happening under the hood is that **the Java compiler infers the type** of the variable event from its context from the signature of **`addActionListener`**.
- This means is that **you don't need to explicitly write out the type** **when it's obvious.** For the sake of readability and familiarity, you have the option to include the type declarations

9a.2 Lambda Expressions

- **A lambda expression a shorthand that allows you to write a method in the same place you are going to use it. Especially useful in places where a method is being used only once, and the method definition is short. It saves you the effort of declaring and writing a separate method to the containing class.**
- **Lambda expressions in Java is usual written using the syntax**

```
(argument) -> (body)
```

For example

```
(arg1, arg2...) -> { body }
(type1 arg1, type2 arg2...) -> { body }
```

9a.2 Lambda Expressions

Examples of Lambda expressions

```
Arrays.sort(testStrings, (s1, s2) ->
                s1.length() - s2.length());
(int a, int b) -> {  return a + b; }
() -> System.out.println("Hello World");

(String s) -> { System.out.println(s); }
() -> 42
() -> { return 3.1415 };
```

9a.2 Lambda Expressions

More Examples of Lambda expressions

```
// Concatenating strings
(String s1, String s2) -> s1+s2;

// Squaring up two integers
(i1, i2) -> i1*i2;

// Summing up the trades quantity
(Trade t1, Trade t2) -> {
  t1.setQuantity(t1.getQuantity() +
t2.getQuantity());
  return t1;
};

// Expecting no arguments and invoking another method
() -> doSomething();
```

9a.2 Lambda Expressions

The structure of lambda expressions.

- **A lambda expression can have zero, one or more parameters.**
- **The type of the parameters can be explicitly declared or it can be inferred from the context.**

`(int a)` **is same as just** `(a)`

- **Parameters are enclosed in parentheses and separated by commas.**

  `(a, b)` **or** `(int a, int b)` **or** `(String a, int b, float c)`

- **Empty parentheses are used to represent an empty set of parameters. i.e.** `() -> 42`

9a.2 Lambda Expressions

The structure of lambda expressions (cont'd).

- **When there is a single parameter, if its type is inferred, it is not mandatory to use parentheses.**

```
a -> return a*a
```

- **The body of the lambda expressions can contain zero, one or more statements.**
- **If body of lambda expression has single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.**
- **When there is more than one statement in body than these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or `void` if nothing is returned.**

9a.2 Lambda Expressions

Passing a value to a Lambda expression

Using a variable from the method embedding anonymous inner classes requires to declare such a variable as `final`, as in the following example.

```
final String name = getUserName();
button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent event) {
                        System.out.println("hi " + name);
                }
});
```

Making a variable `final` means that **you can't reassign to that variable**. It also means that whenever you're using a `final` variable, you know you're using a specific value that has been assigned to the variable

9a.2 Lambda Expressions

Passing an effective `final` value to a Lambda expression

This restriction is relaxed a bit in Java 8. It's possible to **refer to variables that aren't `final`**; however, they still have to be ***effectively* final**. **Although** you haven't declared the variable(s) as `final`, you still **cannot use them as nonfinal variable(s)** if they are to be used in lambda expressions. If you do use them as nonfinal variables, then the compiler will show an error.

```
String name = getUserName();
button.addActionListener(event ->
                         System.out.println("hi " + name));
```

This behavior also helps explain one of the reasons some people refer to lambda expressions as "**closures**." The **variables that aren't assigned to** are ***closed*** over the surrounding state in order to bind them to a value

9a.2 Lambda Expressions

Difference between Lambda Expression and Anonymous class

- **One key difference between using Anonymous class and Lambda expression is the use of `this` keyword. For anonymous class '`this`' keyword resolves to anonymous class, whereas for lambda expression '`this`' keyword resolves to enclosing class where lambda is written.**
- **Another difference between lambda expression and anonymous class is in the way these two are compiled. Java compiler compiles lambda expressions and convert them into `private` method of the class. It uses `invokedynamic` instruction that was added in Java 7 to bind this method dynamically.**

9a.3 Functional interfaces

Lambda expressions are statically typed, so let's investigate **the types of lambda expressions** themselves. These types are called ***functional interfaces***.

A **functional interface** is an `interface` with a **single `abstract` method (SAM)** that is used as the type of a lambda expression.

In Java, all method **parameters have types**. When we were passing the integer 33 as an argument to a method, then parameter would be an `int`. Similarly, when we are passing a **Lambda** expression the parameter type would be a `functional interface` **matching** the **Lambda expression**.

9a.3 Functional interfaces

For example, the following snippet defines an `IAddable` interface. This is a functional interface whose job is to simply add two identical items of type T.

```
@FunctionalInterface
public interface IAddable<T> {
    // To add two objects
    public T add(T t1, T t2);
}
```

Because this interface has one and only one abstract method and it is annotated with `@FunctionalInterface`, it can be used as a type for referencing lambda functions

9a.3 Functional interfaces

```
// Our interface implementations using Lambda expressions
// Joining two strings—note the interface is a generic type

IAddable<String> stringAdder = (String s1, String s2) ->
s1+s2;

// Squaring the number
IAddable<Integer> square = (i1, i2) -> i1*i2;

// Summing up the trades quantity
IAddable<Trade> tradeAdder = (Trade t1, Trade t2) -> {
  t1.setQuantity(t1.getQuantity() + t2.getQuantity());
  return t1;
};
```

9a.3 Functional interfaces

Once we have the implementations ready, we can use them in our class by simply invoking the respective method.

```
// A lambda expression for adding two
// strings.
IAddable<String> stringAdder = (s1, s2) ->
s1+s2;
// this method adds the two strings using
// the first lambda expression
private void addStrings(String s1,String s2)
{
   log("Concatenated Result: " +
                    stringAdder.add(s1, s2));
}
```

9a.3 Functional interfaces

```
// Summing up the trades quantity
IAddable<Trade> aggregatedQty = (t1, t2) -> {
  t1.setQuantity(t1.getQuantity() + t2.getQuantity());
  return t1;
};
// Return a large trade
IAddable<Trade> largeTrade = (t1, t2) -> {
if (t1.getQuantity() > 1000000)
   return t1;
else
   return t2;
};
// Encrypting the trades (Lambda uses an existing method)
IAddable<Trade> encryptTrade = (t1, t2) ->
encrypt(t1,t2);
```

9a.3 Functional interfaces

We have declared lambdas for each our respective functionalities. The method can now be modeled to expect an expression, as shown below.

```
//A generic method with an expected lambda
public void applyBehaviour(IAddable addable,
                            Trade t1, Trade t2){
  addable.add(t1, t2);
}
```

The method is generic enough that it can **apply functionality to any two trades** with the **given behavior** using the given lambda expression (the **`IAddable`** interface).

9a.3 Functional interfaces

- Now, the **client has the control** of c**reating the behaviors and pass it on to the remote server** for application of them. This way, the **client cares about *what* to do**, **while the server cares about *how* to do it**. As long as the interface is designed to accept the lambda expression, the client can create a number of those expressions according to its requirements and invoke the method.
- Before we sum up, let's take an existing **`ActionListener`** interface, which comes with lambda support, and see how it can be used.

9a.3 Functional interfaces

References to Lambda expressions are widely used in handling events in the GUI in particular:

```
ActionListener listener = event ->
    System.out.println("button clicked");
```

where `ActionListener` is the functional interface matching the Lambda expression

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent event);
}
```

As a result of the above assignment you get an instance of a class that implements the interface that was expected in that place

9a.3 Functional interfaces

`ActionListener` **has only one** `abstract` method, `actionPerformed`, and we use it to ***represent*** an **action** that **takes one argument** and **produces no result**. Remember, because `actionPerformed` is defined in an `interface`, it **doesn't actually need** the `abstract` keyword in order to be abstract. It also **has a parent** `interface`, `EventListener`, with no methods at all.

So it's a functional interface**. It doesn't matter what the single method on the interface is called-** it'll get **matched up** to your **lambda expression as long as it has a compatible method signature**. Functional interfaces also let us give **a useful name to the type** of the parameter

9a.3 Functional interfaces

- Functional interfaces are also known as Single Abstract Method (SAM) interfaces.
- Package `java.util.function`
  - Six basic functional interfaces
  - The following slide shows the basic generic functional interfaces.
- Many specialized versions of the basic functional interfaces
  - Use with `int`, `long` and `double` primitive values.
- Also generic customizations of `Consumer`, `Function` and `Predicate`
  - for binary operations(methods that take two arguments).

Interface name	Arguments	Returns	Example
Predicate<T>	T	boolean	Has this album been released yet?
Consumer<T>	T	void	Printing out a value
Function<T,R>	T	R	Get the name from an Artist object
Supplier<T>	None	T	A factory method
UnaryOperator<T>	T	T	Logical not (!)
BinaryOperator<T>	(T, T)	T	Multiplying two numbers (*)

Fig. 9a.1 | Using an anonymous inner class to associate behavior with a button click.

9a.3 Functional interfaces

Function<T, R>

Contains method apply that takes a T argument and returns a value of type R. Calls a method on the T argument and returns the method's result.

BinaryOperator<T>

Contains method apply that takes two T arguments, performs an operation on them (such as calculation) and returns a value of type T

UnaryOperator<T>

Contains method apply that takes no arguments and returns a value of type T. Inherits interface Function<T,T>

9a.3 Functional interfaces

`Predicate<T>`

Contains method `test` that takes a T argument and returns a boolean. Tests whether T argument satisfies a condition

`Consumer<T>`

Contains method `accept` that takes a T argument, performs a task with its T argument such as outputting the object, invoking a method on the object.

`Supplier<T>`

Contains method `get` that takes no arguments and returns (produces, delivers) a value of type T. Often used to create a collection object in which stream operations are placed.

9a.3 Functional interfaces

Example 1.

Consider a lambda that tests whether an Integer is greater than 5. The type of this lambda is actually a `Predicate`, a **functional interface** that checks whether something is true or false.

```
Predicate<Integer> atLeast5 = x -> x > 5;
```

`Predicate` **has a single generic type**, here we've used an `Integer`. The only argument of the lambda expression implementing `Predicate` is therefore **inferred** as an `Integer`. The Java compiler makes a check whether the return value is a `boolean`, as that is the return type of the `Predicate` method

9a.3 Functional interfaces

Example 2.

The **BinaryOperator** functional interface takes two arguments and returns a value, all of which are the same type. In the code example here, this type is **Long**.

```
BinaryOperator<Long> addLongs = (x, y)-> x + y;
```

The inference is smart, but **if it doesn't have enough information**, it won't be able to make the right decision. In these cases, the compiler issues a compile error. For example, if we remove the type information

```
BinaryOperator add = (x, y) -> x + y;
```

we get a **compile error**.

9a.3 Functional interfaces

Example 3. We use a `Function` for transformation purposes, such as converting temperature from Centigrade to Fahrenheit, transforming a String to an Integer, etc

```
//convert centigrade to fahrenheit
Function<Integer,Double> centigradeToFahrenheitInt =
                              x -> new Double((x*9/5)+32);
// String to an integer
Function<String, Integer> stringToInt = x ->
                                     Integer.valueOf(x);
// tests
System.out.println("Centigrade to Fahrenheit:" +
           centigradeToFahrenheitInt.apply(centigrade))
System.out.println(" String to Int:  " +
                              stringToInt.apply("4"));
```

9a.3 Functional interfaces

Example 4. A bit more sophisticated requirement, like aggregating the trade quantities for a given list of trades, can be expressed as a `Function`

```
Function<List<Trade>,Integer> aggegatedQuantity =
 trades -> {   int aggregatedQuantity = 0;
               for (Trade t: trades){
                   aggregatedQuantity+=t.getQuantity();
               }
               return aggregatedQuantity;
            };
```

9a.3 Functional interfaces

Two argument functions

Until now, we have dealt with functions that only accept a single input argument. There are use cases that may have to operate on two arguments. For example, a function that expects two arguments but produces a result by operating on these two arguments. This type of functionality fits into two-argument functions bucket such as `BiPredicate, BiConsumer, BiFunction`, etc. They are pretty easy to understand too except that the signature will have an additional type (two input types and one return type)

9a.3 Functional interfaces

Example: The `BiFunction` interface definition is shown below

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
   R apply(T t, U u);
}
```

The above function has three types `T, U` and `R`

9a.3 Functional interfaces

Example: `BiFunction` usage

Accepts two `Trades` to produce sum of trade quantities. The input types are `Trade` and return type is `Integer`:

```
BiFunction<Trade,Trade,Integer> sumQuantities =
   (t1, t2) -> {
                 return t1.getQuantity()+t2.getQuantity();
                };
```

9a.3 Functional interfaces

Example: `BiPredicate` usage

`BiPredicate` expects two input arguments and returns a `boolean` value :

```
// Predicate expecting two trades to compare and
// returning the condition's output
BiPredicate<Trade,Trade> isBig =
            (t1, t2) -> t1.getQuantity() > t2.getQuantity();
```

9a.3 Functional interfaces

Example: `Runnable` **usage**

```
@FunctionalInterface
public interface Runnable {
    public abstract void run();
}
// Executes a method without args
// and returns void
Runnable action = () ->
System.out.println("Printing …");
         action.run();
```

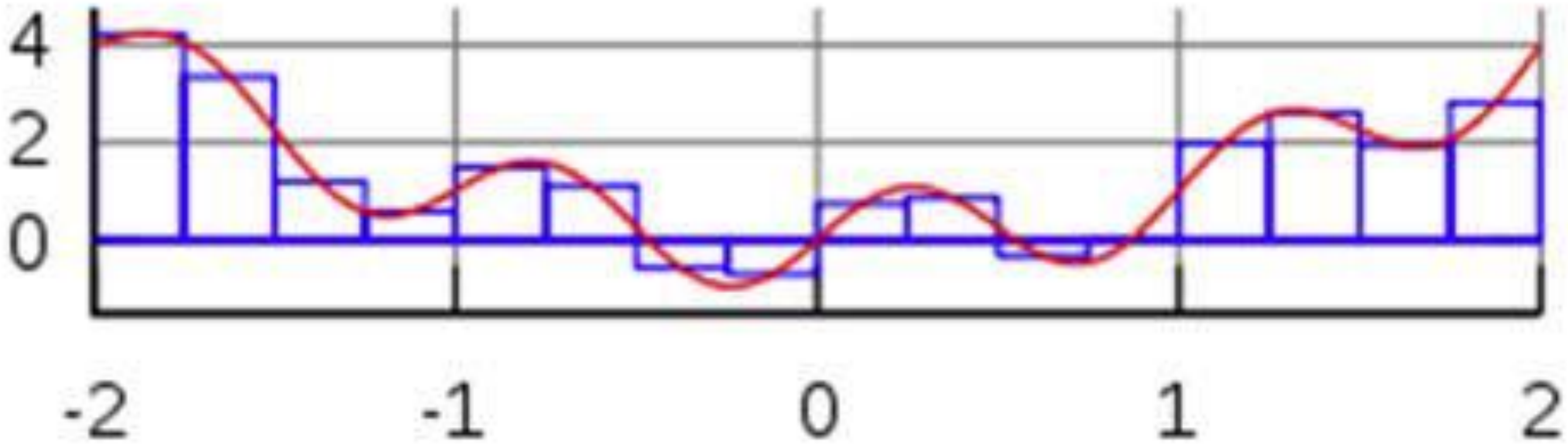
9a.3 Functional interfaces

When you want **compile-time checking** that interface is in right form (exactly one abstract method), and want to **alert other developers that lambdas can be used** use `@FunctionalInterface`

Example

```
@FunctionalInterface
public interface Integrable {
        double eval(double x);
}
```

Method for numerical integration using the Method of the rectangles



9a.3 Functional interfaces

```
@FunctionalInterface
public interface Integrable {
        double eval(double x);
}
```

9a.3 Functional interfaces

```
public static double integrate(Integrable function,
                               double x1, double x2,
                               int numSlices){
  if (numSlices < 1) {
    numSlices = 1;
  }
  double delta = (x2 - x1)/numSlices;
  double start = x1 + delta/2;
  double sum = 0;
  for(int i=0; i<numSlices; i++) {
    sum += delta * function.eval(start + delta * i);
  }
  return(sum);
}
```

9a.3 Functional interfaces

```
public static void integrationTest(Integrable function,
                                   double x1, double x2) {
  for(int i=1; i<7; i++) {
    int numSlices = (int)Math.pow(10, i);
    double result =
      MathUtilities.integrate(function, x1, x2, numSlices);
    System.out.printf("  For numSlices =%,10d result = %,.8f%n",
                      numSlices, result);
  }
}
```

```
MathUtilities.integrationTest(x -> x*x, 10, 100);
MathUtilities.integrationTest(x -> Math.pow(x,3), 50, 500);
MathUtilities.integrationTest(x -> Math.sin(x), 0, Math.PI);
MathUtilities.integrationTest(x -> Math.exp(x), 2, 20);
```

9a.3 Functional interfaces

Interfaces like `Integrable` are widely used and

`java.util.function` defines many simple functional (SAM) interfaces that are named according to arguments and return values

Hence, replace my `Integrable` with built-in `DoubleFunction<Double>`.

You need to look in API for the method names. Although lambdas don't refer to method names, your code that *uses* the lambdas will need to call the methods.

9a.3 Functional interfaces

Because `DoubleFunction` is a **functional interface,** containing a method with same signature as the method of the `Integrable` interface, **you may omit the definition of `Integrable` entirely and replace**

```
public static double integrate(
                                    Integrable function, …) {
                      … function.eval(…); …
}
```

with

```
public static double integrate(
                  DoubleFunction<Double>function, … ){
                        … function.apply(…); …
}
```

9a.4 Method references

Method references are shortcuts that you can use anywhere you would use a lambda expression**.** They are **especially useful** in cases **when a lambda expression** does nothing but **call an existing method**. In those cases, it's often clearer to refer to the existing method by name. Method references enable you to do this because they are compact, easy-to-read lambda expressions for methods that already have a name.

Kind	Example
Reference to a static method	`ContainingClass::staticMethodName`
Reference to an instance method of a particular object	`ContainingObject::instanceMethodName`
Reference to an instance method of an arbitrary object of a particular type	`ContainingType::methodName`
Reference to a constructor	`ClassName::new`

9a.4 Method references

Kind	Syntax	Example
Reference to a static method	`Class::staticMethodName`	`String::valueOf`
Reference to an instance method of a specific object	`object::instanceMethodName`	`x::toString`
Reference to an instance method of a arbitrary object supplied later	`Class::instanceMethodName`	`String::toString`
Reference to a constructor	`ClassName::new`	`String::new`

Kind	Syntax	As Lambda
Reference to a static method	`Class::staticMethodName`	`(s) -> String.valueOf(s)`
Reference to an instance method of a specific object	`object::instanceMethodName`	`() -> "hello".toString()`
Reference to an instance method of a arbitrary object supplied later	`Class::instanceMethodName`	`(s) -> s.toString()`
Reference to a constructor	`ClassName::new`	`() -> new String()`

9a.4 Method references

Method reference in Java 8 is the ability to use a method as an argument for a matching functional interface.

The method in the functional interface and the passing method reference should match for the argument and return type. Method reference can be done for both `static` and `class` methods.

9a.4 Method references

```
public static void integrationTest(DoubleFunction<Double>  function,
            double x1, double x2) {
        for (int i = 1; i < 7; i++) {
            int numSlices = (int) Math.pow(10, i);
            double result
               = MathUtilsWithStandardSAM.integrate(function, x1, x2, numSlices);
            System.out.printf(" For numSlices =%,10d result = %,.8f%n",
                    numSlices, result);
        }
}
public static void main(String[] args) {
        MathUtilsWithStandardSAM.integrationTest(Math::sin, 0, Math.PI);
        MathUtilsWithStandardSAM.integrationTest(x -> Math.sin(x), 0, Math.PI);
}
```

9a.4 Method references

There are several different kinds of method references, each with slightly different syntax:

- **A `static` method (`ClassName::methName`)**
- **An instance method of a particular object (`instanceRef::methName`)**
- **A `super` method of a particular object (`super::methName`)**
- **An instance method of an arbitrary object of a particular type (`ClassName::methName`)**
- **A `class` constructor reference (`ClassName::new`)**
- **An array constructor reference (`TypeName[]::new`)**

9a.4 Method references

For a `static` method reference, the `class` to which the method belongs precedes the `::` delimiter, such as in `Integer::sum`.

For a reference to an instance method of a particular object, an expression evaluating to an object reference precedes the delimiter:

```
ArrayList<String> knownNames =//some List
Predicate<String> isKnown =
                        knownNames::contains;
```

Here, the implicit lambda expression would capture the `String` object referred to by `knownNames`, and the body would invoke `ArrayList.contains` using that object as the receiver.

9a.4 Method references

The ability to reference the method of a specific object provides a convenient way to convert between different functional interface types:

```
Comparable<Computer> c = ...
PrivilegedAction<Computer> a = c::compareTo;
```

9a.4 Method references

For a reference to an instance method of an arbitrary object, the type to which the method belongs precedes the delimiter, and the invocation's receiver is the first parameter of the functional interface method:

```
Function<String, String> upperfier =
                                String::toUpperCase;
```

Here, the implicit lambda expression has one parameter, the string to be converted to upper case, which becomes the receiver of the invocation of the `toUpperCase()` method. If the class of the instance method is generic, its type parameters can be provided before the :: delimiter or, in most cases, inferred from the target type

9a.4 Method references

Constructors can be referenced in much the same way was as static methods by using the name `new`:

```
Supplier<Computer> computerFactory =
                                  Computer::new;
System.out.print(computerFactory.get()
                          instanceof Computer);
// prints 'true'
```

If a class has multiple constructors, the target type's method signature is used to select the best match in the same way that a constructor invocation is resolved.

9a.4 Method references

If we want to obtain the reference of the general purpose constructor `Computer(String name)` constructor, the target type should allow one parameter of type `String`, as we see below:

```
Function<String,Computer> compFactory = Computer::new;
System.out.print(compFactory.apply("HP").getText());
 // prints 'HP'
```

9a.4 Method references

Our last example deals with an **array constructor reference**. If we would like to create an array of ten `Computer` objects we would write

```
new Computer [10];
```

Consequently the **target variable type** should allow **one parameter** of type `Integer` (defining the array's `length`) and a `Computer[]` `return` type:

```
Function<Integer,Computer[]> computerArrayFactory =
                                            Computer[]::new;
System.out.print(computerArrayFactory.apply(10).length);
// prints '10'
IntFunction<int[]> arrayMaker = int[]::new;
int[] array = arrayMaker.apply(10);
// creates an int[10]
```

9a.5 `default` Interface Methods

- **Prior to Java SE 8, interface methods could be *only* `public abstract` methods.**
  - **An interface specified *what* operations an implementing class must perform but not *how* the class should perform them.**
- **In Java SE 8, interfaces also may contain `public default` methods with concrete default implementations that specify how operations are performed when an implementing class does not override the methods.**
- **If a class implements such an interface, the class also receives the interface's `default` implementations (if any).**
- **To declare a default method, place the keyword `default` before the method's return type and provide a concrete method implementation.**

9a.5 `default` Interface Methods

Default methods provide a more object-oriented way to add concrete behavior to an interface. These are a new kind of method- `interface` method can either be abstract or default.

Default methods have an implementation that is inherited by classes that do not override it.

Default methods in a functional interface don't count against its limit of one abstract method.

For example, we could have (though did not) add a `skip` method to `Iterator`, as shown in the next slide:

9a.5 `default` Interface Methods

```
interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();

    default void skip(int i) {
        for (; i > 0 && hasNext(); i--)
                next();
    }
}
```

9a.5 `default` Interface Methods

From a client's perspective, method `skip` is just another virtual method provided by the `interface`.

Any class that implements the original interface will *not* break when a `default` method `skip` is added.

– The class simply receives the new `default` method.

When a class implements a Java SE 8 interface, the class "*signs a contract*" with the compiler that says,

– "I will declare all the *`abstract`* methods specified by the interface or I will declare my class `abstract`"

The implementing class is not required to override the interface's `default` methods, but it can if necessary.

9a.5 `default` Interface Methods

Note:

Java SE 8 default (virtual) methods enable you to evolve existing interfaces by adding new methods to those interfaces without breaking code that uses them.

Default methods are also referred to as Defender Methods or Virtual extension methods

9a.5 `default` Interface Methods

For example, let’s assume that every component is said to have a `name` and `creation date` when instantiated. However, should the implementation doesn’t provide concrete implementation for the name and creation date, they would be inherited from the interface by default. For out example, the `IComponent` interface defines a set of default methods as shown on the following slide.

In this case the interfaces now have an implementation rather than just being `abstract`. The methods are prefixed with a keyword `default` to indicate them as the default methods

9a.5 `default` Interface Methods

```
@FunctionalInterface
public interface IComponent {

// Functional method - note we must have one of
// these functional methods only
  public void init();

// default method - note the keyword default
  default String getComponentName(){
    return "DEFAULT NAME";
  }
// default method - note the keyword default
  default Date getCreationDate(){
    return new Date();
  }
}
```

9a.5 `default` Interface Methods

When you `extend` an `interface` that contains a `default` method, you can do the following:

- **Not mention the default method at all, which lets your extended interface inherit the default method. Any class that implements the derived interface will get the implementation specified by the default method in the base interface.**
- **Redeclare the default method, which makes it abstract. Any class that implements the derived interface has to implement that method.**
- **Redefine the default method, which overrides it. Any class that implements the derived interface will use the implementation of the default method in that interface**

9a.5 default Interface Methods

```
interface IAA {
    default void foo() {
        System.out.println("Calling IAA.foo()");
    }
}
interface IBB extends IAA { // inherits default method
}
interface IC extends IAA {

    void foo();// default method converted to an abstract method
}
public interface IDD extends IAA {
    default void foo() { // overrides the default method from IAA
        System.out.println("Calling IDD.foo()");
    }
}
```

Examples for default methods

```
public class CClass implements IAA, IDD {
      public void foo() {
        IAA.super.foo();// call IAA foo()
        IDD.super.foo();// calls IDD foo()
     }
}
public class BClass implements IC {
      public void foo() {
        System.out.println(“Implement foo()");
    }
}
public class DClass implements IBB {
      public void foo() {
            IBB.super.foo();// calls IAA foo()     }
}
```

9a.5 `default` Interface Methods

Multiple inheritance is not new to Java. Java has provided multiple inheritance of types since its inception. If we have an object hierarchy implementing various interfaces, there are a few rules help us understand which implementation is applied to the child class.

The fundamental rule is that the closest concrete implementation to the subclass wins the inherited behavior over others. The immediate concrete type has the precedence over any others

9a.5 `default` Interface Methods

```
// Person interface with a
// concrete implementation of name
interface Person{
  default String getName(){
    return "Person";
  }
}
// Faculty interface extending Person
// but with its own name implementation
interface Faculty extends Person{
  default public String getName(){
    return "Faculty";
  }
}
```

9a.5 `default` Interface Methods

```
// The Student inherits Faculty's name
// rather than Person
class Student implements Faculty, Person{
.. }

// the getName() prints Faculty
private void test() {
   String name = new Student().getName();
   System.out.println("Name is "+name);
}
output: Name is Faculty
```

9a.5 default Interface Methods

```
interface Person{ .. }
// Notice that the faculty is NOT
// implementing Person
interface Faculty { .. }

// As there's a conflict, out Student class must
// explicitly declare whose name it's going to
// inherit!
class Student implements Faculty, Person{
  @Override
  public String getName() {
    return Person.super.getName();

  }
}
```

9a.5 `default` Interface Methods

It is forbidden to define `default` methods in interfaces for methods in `java.lang.Object`, since the default methods would never be "reachable".

Default interface methods can be overwritten in classes implementing the interface and the class implementation of the method has a higher precedence than the interface implementation, even if the method is implemented in a superclass. Since all classes inherit from `java.lang.Object`, the methods in `java.lang.Object` would have precedence over the `default` method in the interface and be invoked instead.

9a.5 `default` Interface Methods

Simply put, `default` methods were designed to provide the default behavior where there is no other definition and not to provide implementations that will "*compete*" with other existing implementations.

The "base class always wins" rule has its solid reasons, too. It is supposed that classes define real implementations, while interfaces define default implementations, which are somewhat weaker.

9a.6 `static` Interface Methods

It is common to associate with an interface a class containing `static` helper methods for working with objects that implemented the interface.

For example class `Collections` contains many `static` helper methods for working with objects that implement interfaces `Collection`, `List`, `Set` and more.

`Collections` method `sort` can sort objects of *any* class that implements interface `List`.

With `static` interface methods, such helper methods can now be declared directly in interfaces rather than in separate classes.

9a.6 `static` Interface Methods

`Static` methods are similar to **`default`** methods except that **we can't override them** in the implementation classes. This feature helps us in **avoiding undesired results in case of poor implementation** in child classes.

Consider **`interface MyData`** with a **`static isNull()`** method and **`class MyDataImpl`** with an existing poor implementation of method **`isNull()`** .

Note, that method **`isNull()`** in **`class MyDataImpl`** is **not overriding** the **`interface`** method. For example, if we will add **`@Override`** annotation to the **`isNull()`** method, it will result in compiler error

9a.6 static Interface Methods

```
public interface MyData {
   default void print(String str) {
     if (!isNull(str)) //call the static method
      System.out.println("MyData Print::" + str);
   }
   static boolean isNull(String str) {
     System.out.println("Interface Null Check");
     return str == null ? true : "".equals(str) ? true : false;
   }
}
```

9a.6 static Interface Methods

```
public class MyDataImpl implements MyData {
    public boolean isNull(String str) {
        System.out.println("Impl Null Check");
        return str == null ? true : false;
    }
    public static void main(String args[]){
        MyDataImpl obj = new MyDataImpl();
        obj.print("");// call the static method
        obj.isNull("abc"); // call the instance method
    }
}
// Output
```

Interface Null Check

Impl Null Check

9a.6 `static` Interface Methods

If we make the `interface` method from `static` to `default`, we will get following output.

```
Impl Null Check
MyData Print::
Impl Null Check
```

The `static` methods are visible to `interface` methods only, if we remove the `isNull()` method from the `MyDataImpl` class, we won’t be able to use it for the `MyDataImpl` object. However like other `static` methods, we can use `interface` `static` methods using `class` name. For example, a valid statement will be:

```
boolean result = MyData.isNull("abc");
```

Questions