Lambda Functions

Lambda Functions

vs Inner classes

Functional interface

- Interface with exactly one abstract method

```
@FunctionalInterface
interface Helloable {
    String hello();
}
```

Interface as argument

- Object of interface type as argument

```
class InnerVsLambda {
    private void yo(Helloable h) {
        System.out.println(h.hello());
    }
    other code
}
```

Interface as argument

- Instantiating object via inner class

```
class InnerVsLambda {
    private void yo(Helloable h) {
        System.out.println(h.hello());
    }
    public static void main(String... args) {
        InnerVsLambda ivsl = new InnerVsLambda();
        ivsl.yo(new Helloable() {
            @Override
            public String hello() {
                return "Hello from inner class version!";
            }
        });
}
```

Argument reconsidered

- Don’t do anything with argument object except execute its method

```
private void yo(Helloable h) {
    System.out.println(h.hello());
}
```

- Only really need a method to execute

Argument reconsidered

- Only really need a method to execute
- So instead of

```
ivsl.yo(Helloable object);
```

- ...give it a ~~method~~function

```
ivsl.yo(() -> "Hello from functional version!");
```

Inner class vs lambda function

- Which is more understandable?

```
class InnerVsLambda {
    private void yo(Helloable h) {
        System.out.println(h.hello());
    }
    public static void main(String... args) {
        InnerVsLambda ivsl = new InnerVsLambda();
        ivsl.yo(new Helloable() {
            @Override
            public String hello() {
                return "Hello from inner class version!";
            }
        });
        ivsl.yo(() -> "Hello from functional version!");
}
```

Lambda Functions

But wait, there's more...

Simpler and simpler

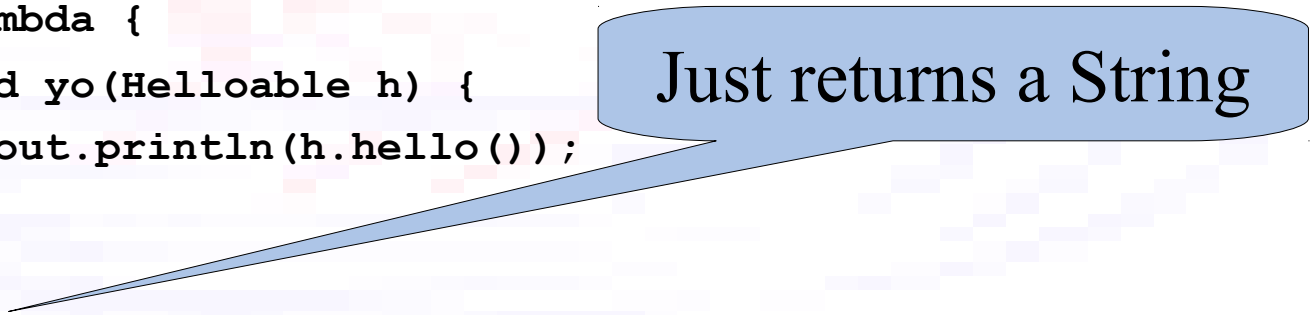
- Suppose there's an additional method:

```
class InnerVsLambda {
    private void yo(Helloable h) {
        System.out.println(h.hello());
    }

    private String appropriateMethod() {
        return "Hello from method reference version!");
    }

    public static void main(String... args) {
        InnerVsLambda ivsl = new InnerVsLambda();
        ivsl.yo(() -> "Hello from functional version!");
}
```

Simpler and simpler

- Suppose there's an additional method:

```
class InnerVsLambda {
    private void yo(Helloable h) {
        System.out.println(h.hello());
    }

    private String appropriateMethod() {
        return "Hello from method reference version!");
    }

    public static void main(String... args) {
        InnerVsLambda ivsl = new InnerVsLambda();
        ivsl.yo(() -> "Hello from functional version!");
}
```



Simpler and simpler

- Suppose there's an additional method:

```
class InnerVsLambda {
    private void yo(Helloable h) {
        System.out.println(h.hello());
    }

    private String appropriateMethod() {
        return "Hello from method reference version!");
    }

    public static void main(String... args) {
        InnerVsLambda ivsl = new InnerVsLambda();
        ivsl.yo(() -> "Hello from functional version!");
}
```

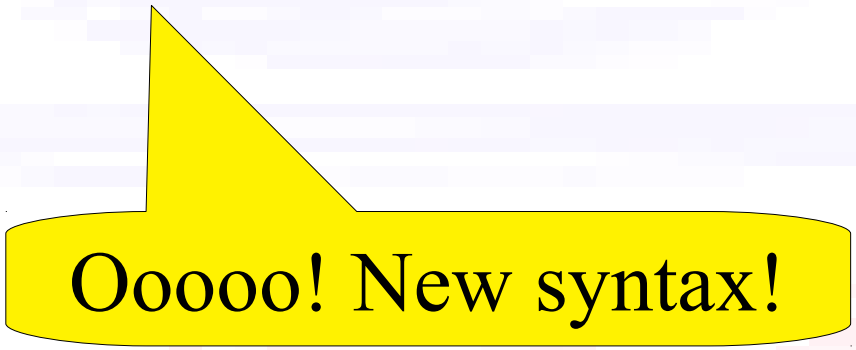
Just returns a String

Just returns a String

Argument reconsidered

- Only really need a method to execute
- Already have an appropriate method
- Can use a *method reference* as argument

```
ivsl.yo(ivsl::appropriateMethod);
```



Simpler and simpler

- Suppose there's an additional method:

```
class InnerVsLambda {
    private void yo(Helloable h) {
        System.out.println(h.hello());
    }

    private String appropriateMethod() {
        return "Hello from method reference version!");
    }

    public static void main(String... args) {
        InnerVsLambda ivsl = new InnerVsLambda();
        ivsl.yo(() -> "Hello from functional version!");
        ivsl.yo(ivsl::appropriateMethod);
}
```

Lambda Functions

∃ In API

API functional interfaces

- Instead of writing your own

```
@FunctionalInterface
interface Helloable {
    String hello();
}
```

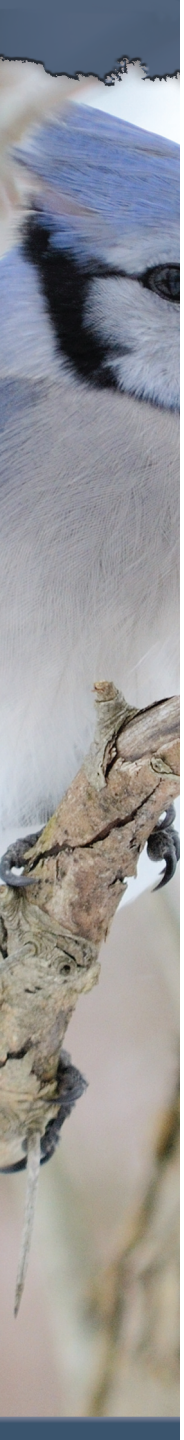
- Use one from the API, eg,

```
import java.util.function.Supplier
```

- Emphasizes importance of the function as opposed to an object

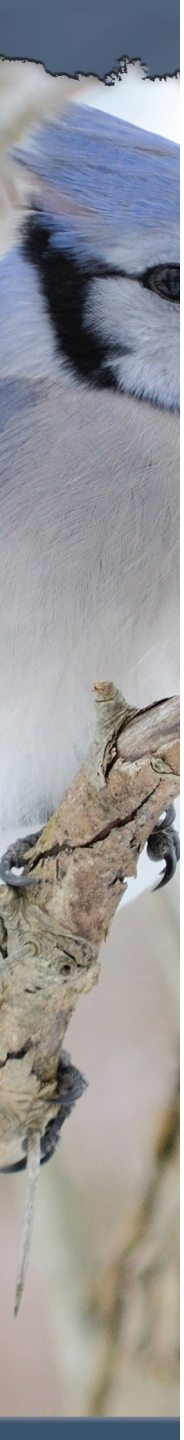
API functional interfaces

```
import java.util.function.Supplier;

class ArgumentWithSupplier {
    private void yo(Supplier<String> f) {
        System.out.println(f.get());
    }

    private String appropriateMethod() {
        return "Hello from method reference version!");
    }

    public static void main(String... args) {
        ArgumentWithSupplier aws = new ArgumentWithSupplier();
        aws.yo(() -> "Hello from supplier version!");
        aws.yo(aws::appropriateMethod) ;
    }
}
```



Functional interfaces and lambdas

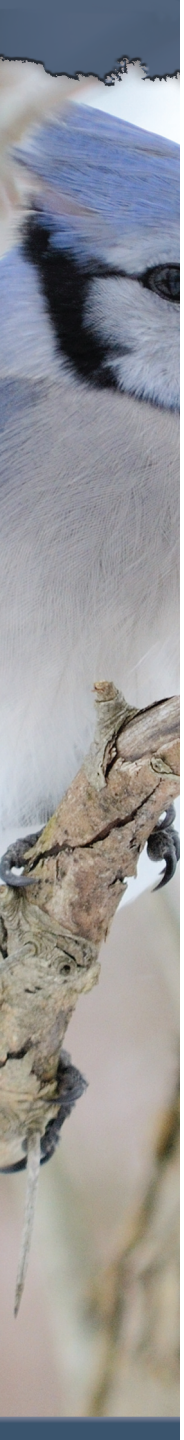
- Interfaces with a single abstract method are *functional interfaces*.
- **`@FunctionalInterface`** is the associated annotation.
- A lambda may be used where a functional interface is required.
- **`java.util.function`** defines some functional interfaces.



Common functional interfaces

- **`Consumer`**: for lambdas with a **`void`** return type.

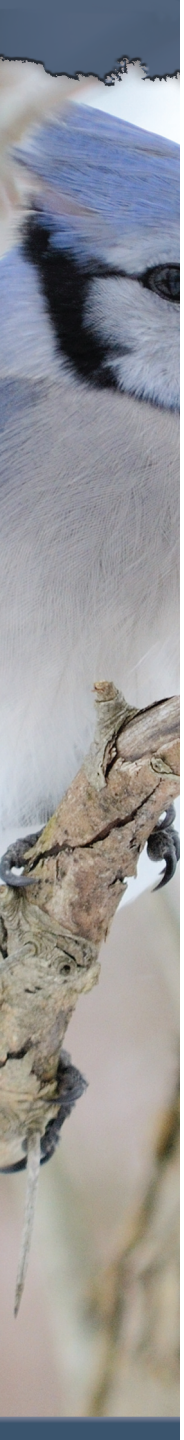
```
void sumer(String s, Consumer<String> c) {
    c.accept(s);
}

sumer("foo", e -> System.out.println(e));
sumer("foo",
    e -> System.out.print(e.toUpperCase()));
```



Common functional interfaces

- **`BinaryOperator`**: for lambdas with two parameters and a matching result type.

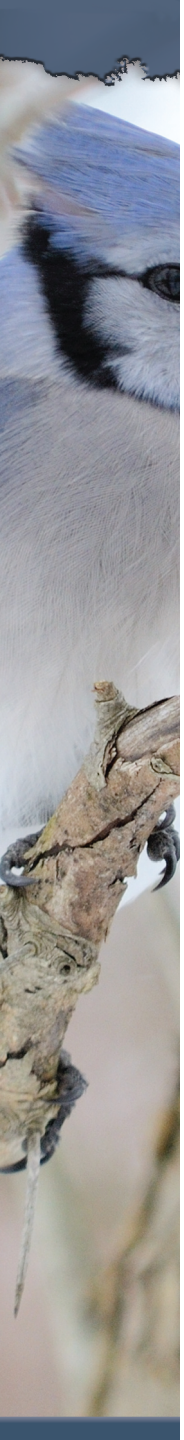
```
double boper(double a, double b,
    BinaryOperator<Double> bop) {
    return bop.apply(a, b);
}

boper(42, 1.0, (a, b) -> 2 * (a + b));
boper(42, 1.0, (a, b) -> a - b);
```



Common functional interfaces

- **`Supplier`**: for lambdas returning a result.

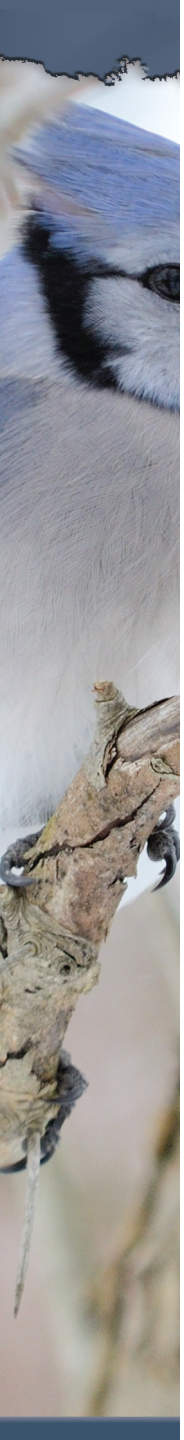
```
int supp(Supplier<Integer> s) {
    return s.get();
}

supp(() -> new Random().nextInt(10));
supp(
    () -> (int) Math.abs(new Random().nextInt()));
```



Common functional interfaces

- **`Predicate`**: returns a **`boolean`**.

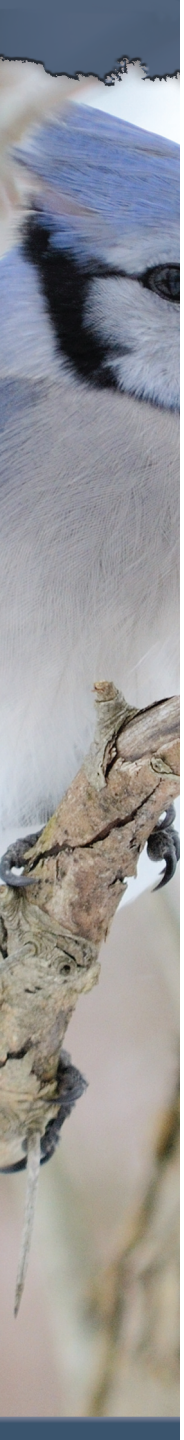
```
boolean boo(String s, Predicate<String> p) {
    return p.test(s);
}

boo("foo", s -> s.trim().length() > 2);
```



Functional interfaces

- Functional types can be assigned to variables, eg,:

```
BinaryOperator<String> akaOp =
    (name, alias) -> name + " (AKA " + alias + ")";

String aka(String s, String t,
        BinaryOperator<String> bo) {
   return bo.apply(s, t);
}

aka("foo", "bar", akaOp);
```

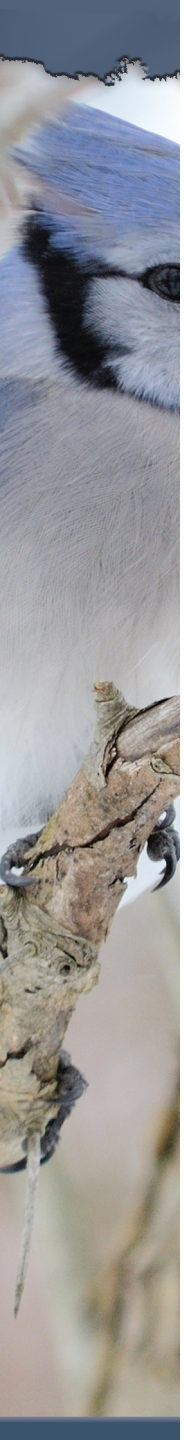


Functional interfaces

- Functional types can be assigned to variables, eg,:

```
BinaryOperator<String> akaOp =
    (name, alias) -> name + " (AKA " + alias + ")";

String aka(String s, String t,
        BinaryOperator<String> bo) {
   return bo.apply(s, t);
}

aka("foo", "bar", akaOp);
```

```
foo (AKA bar)
```



Functional interfaces

- Functional types can be assigned to variables, eg,:

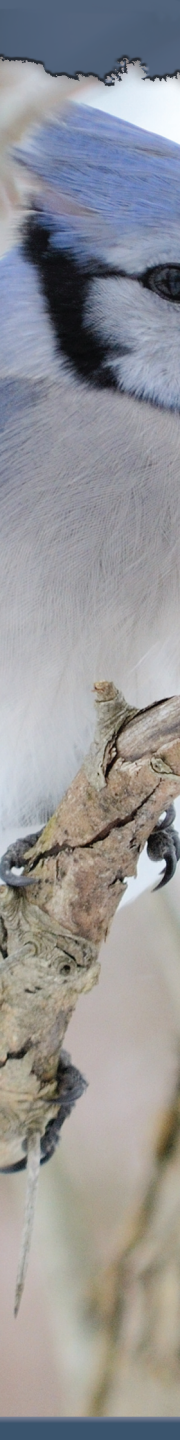
```
BinaryOperator<String> akaOp =
    (name, alias) -> name + " (AKA " + alias + ")";

String aka(String s, String t,
        BinaryOperator<String> bo) {
   return bo.apply(s, t);
}

aka("foo", "bar", akaOp);
```

Functional interfaces

- Functional types can be assigned to variables, eg,:

```
BinaryOperator<String> akaOp =
    (name, alias) -> name + " (AKA " + alias + ")";

String aka(String s, String t,
        BinaryOperator<String> bo) {
   return bo.apply(s, t);
}

aka("foo", "bar", akaOp);
```

```
foo (AKA bar)
```



Functional interfaces

- Functional types can be used in collections, eg,:

```
private static Map<String, Supplier<String>> FUNCS
        = new HashMap<>();
FUNCS.put("A", this::methodA);
FUNCS.put("B", this::methodB);
FUNCS.put("C", this::methodC);

private String methodA() {return "methodA";}
private String methodB() {return "methodB";}
private String methodC() {return "methodC";}

System.out.println(FUNCS.get("B").get());
```

Functional interfaces

- Functional types can be used in collections, eg,:

```
private static Map<String, Supplier<String>> FUNCS
        = new HashMap<>();
FUNCS.put("A", this::methodA);
FUNCS.put("B", this::methodB);
FUNCS.put("C", this::methodC);

private String methodA() {return "methodA";}
private String methodB() {return "methodB";}
private String methodC() {return "methodC";}

System.out.println(FUNCS.get("B").get());
```

```
instance methodB
```