3 Lambda expressions

This chapter covers

- Lambdas in a nutshell
- Where and how to use lambdas
- The execute-around pattern
- Functional interfaces, type inference
- Method references
- Composing

3.1 Lambdas in a nutshell

- A *lambda expression*
  - a concise representation of an anonymous function that can be passed around.
- Using a lambda expression you can create a custom Comparator object in a more concise way.

Before:

```
Comparator<Apple> byWeight = new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
};
```

After (with lambda expressions):

```
Comparator<Apple> byWeight =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
```

The lambda we just showed you has three parts, as shown in figure 3.1:

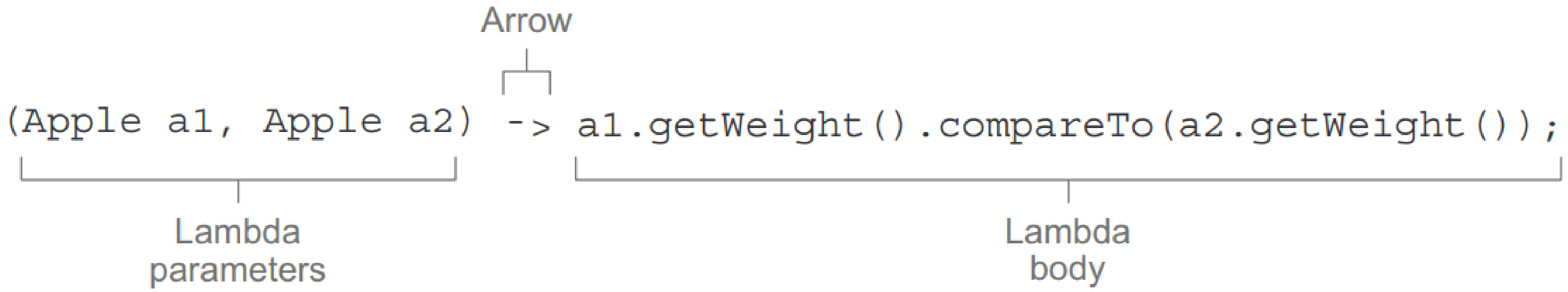


Figure 3.1 A lambda expression is composed of parameters, an arrow, and a body.

- *A list of parameters*
- *An arrow*
- *The body of the lambda*

Listing 3.1 Valid lambda expressions in Java 8

```
(String s) -> s.length()
(Apple a) -> a.getWeight() > 150
(int x, int y) -> {
    System.out.println("Result:");
    System.out.println(x + y);
}

() -> 42
(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())
```

Takes one parameter of type String and returns an int. It has no return statement as return is implied.

Takes one parameter of type Apple and returns a boolean (whether the apple is heavier than 150 g).

Takes two parameters of type int and returns no value (void return). Its body contains two statements.

Takes no parameter and returns the int 42

Takes two parameters of type Apple and returns an int representing the comparison of their weights

Table 3.1 Examples of lambdas

Use case	Examples of lambdas
A boolean expression	`(List<String> list) -> list.isEmpty()`
Creating objects	`() -> new Apple(10)`
Consuming from an object	`(Apple a) -> { System.out.println(a.getWeight()); }`
Select/extract from an object	`(String s) -> s.length()`
Combine two values	`(int a, int b) -> a * b`
Compare two objects	`(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`

3.2 Where and how to use lambdas

- List<Apple> greenApples =
  filter(inventory, (Apple a) -> GREEN.equals(a.getColor()));
  Predicate<T>

- Where exactly can you use lambdas?
  Ans: You can use a lambda expression in the context of
  a functional interface.

Functional Interface

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

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

- a functional interface
  - an interface that specifies exactly one abstract method.
  - Now a functional interface can have many default methods.
- What can you do with functional interfaces?
  - Lambda expression let you provide the implementation of the abstract method of a functional interface directly in line.
  - Treat the whole expression as an instance of a functional interface.

```
public interface Comparator<T> {                          java.util.Comparator
    int compare(T o1, T o2);
}
public interface Runnable {                               java.lang.Runnable
    void run();
}
public interface ActionListener extends EventListener {   java.awt.event.ActionListener
    void actionPerformed(ActionEvent e);
}
public interface Callable<V> {                            java.util.concurrent.Callable
    V call() throws Exception;
}
public interface PrivilegedAction<T> {                    java.security.PrivilegedAction
    T run();
}
```

Runnable is a functional interface defining only one abstract method, run:

```
Runnable r1 = () -> System.out.println("Hello World 1");
Runnable r2 = new Runnable() {
    public void run() {
        System.out.println("Hello World 2");
    }
};
public static void process(Runnable r) {
    r.run();
}
process(r1);
process(r2);
process(() -> System.out.println("Hello World 3"));
```

Uses a lambda

Uses an anonymous class

Prints "Hello World 1"

Prints "Hello World 2"

Prints "Hello World 3" with a lambda passed directly

Function descriptor

- The signature of the abstract method of the functional interface describes the signature of the lambda expression. We call this abstract method a *functional descriptor*.
- () -> void
  (Apple, Apple) -> int
- A lambda expression can be
  - assigned to a variable
  - passed to a method expecting a functional interface as an argument

```
public void process(Runnable r) {
    r.run();
}
process(() -> System.out.println("This is awesome!!"));
```

Lambdas and void method invocation

Although this may feel weird, the following lambda expression is valid:

```
process(() -> System.out.println("This is awesome"));
```

After all, `System.out.println` returns `void` so this is clearly not an expression! Why don't we have to enclose the body with curly braces like this?

```
process(() -> { System.out.println("This is awesome"); });
```

It turns out that there's a special rule for a void method invocation defined in the Java Language Specification. You don't have to enclose a single void method invocation in braces.

3.3 Putting Lambdas into practice: the execute-around pattern

- the *execute-around* pattern
  - Setup phase -> Code doing the processing -> Cleanup phase
- Read the next slide.
  - This code is limited to read only the first line of the file.
  - You need to parameterize the behavior of processFile.

```java
public String processFile() throws IOException {
try (BufferedReader br =
             new BufferedReader(new FileReader("data.txt"))) {
        return br.readLine();
    }
}
```

This is the line that does useful work.

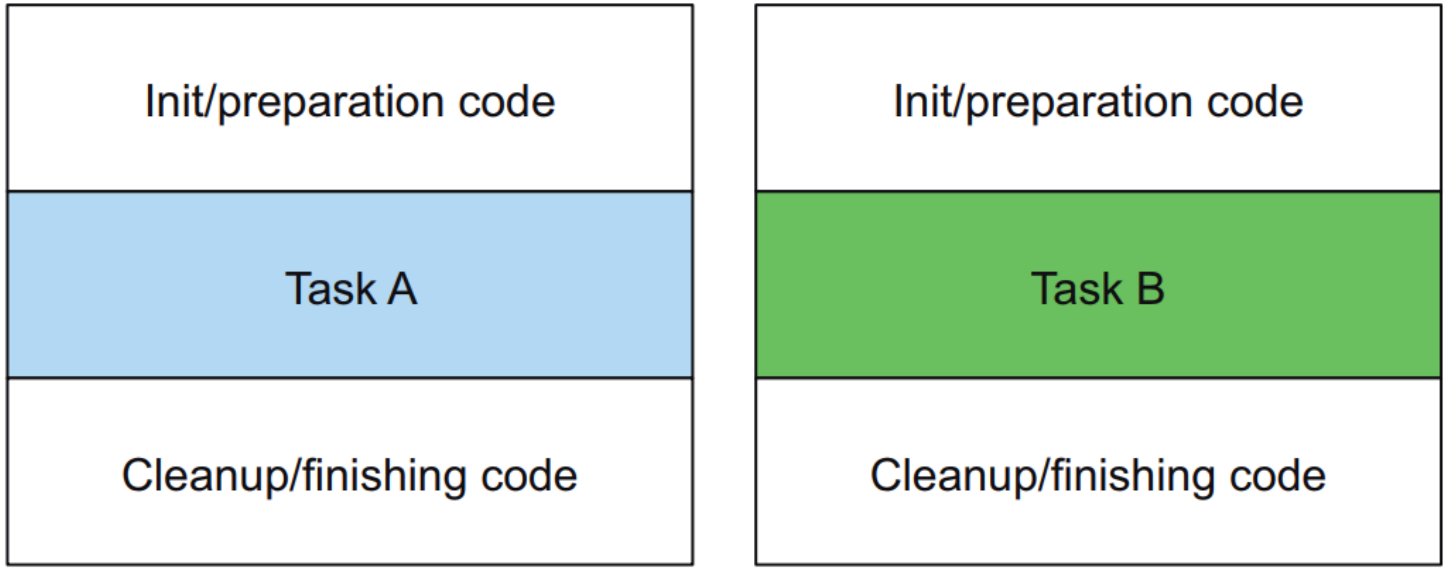


Figure 3.2 Tasks A and B are surrounded by boilerplate code responsible for preparation/cleanup.

- Step 1: Remember behavior parameterization
  - String result
    = processFile((BufferedReader br) -> br.readLine() + br.readLine());
- Step 2: Use functional interfaces to pass behaviors.
  - ```
    @FunctionalInterface
    public interface BufferedReaderProcessor {
        String process(BufferedReader b) throws IOException;
    }
    public String processFile(BufferedReaderProcessor p)  throws IOException {
        …
    }
    ```
    ```

- Step 3: Execute a behavior!

```
public String processFile(BufferedReaderProcessor p) throws IOException {
    try (BufferedReader br =
                    new BufferedReader(new FileReader("data.txt"))) {
        return p.process(br);
    }
}
```

Processes the BufferedReader object

- Step 4: Pass lambdas

The following shows processing one line:

```
String oneLine =
    processFile((BufferedReader br) -> br.readLine());
```

The following shows processing two lines:

```
String twoLines =
    processFile((BufferedReader br) -> br.readLine() + br.readLine());
```

```java
public String processFile() throws IOException {
    try (BufferedReader br =
            new BufferedReader(new FileReader("data.txt"))){
        return br.readLine();
    }
}
```

1

```java
public interface BufferedReaderProcessor {
  String process(BufferedReader b) throws IOException;
}

public String processFile(BufferedReaderProcessor p) throws
IOException {
   …
}
```

2

```java
public String processFile(BufferedReaderProcessor p)
throws IOException {
   try (BufferedReader br =
           new BufferedReader(new FileReader("data.txt"))){
       return p.process(br);
   }
}
```

3

```java
String oneLine = processFile((BufferedReader br) ->
                                 br.readLine());

String twoLines = processFile((BufferedReader br) ->
                                 br.readLine + br.readLine());
```

4

Figure 3.3 Four-step process to apply the execute-around pattern

3.4 Using functional interfaces

- Comparable, Runnable, and Callable functional interfaces.
- java.util.function package
  - Predicate
    - java.util.function.Predicate<T>
  - Consumer
    - java.util.function.Consumer<T>
  - Function
    - java.util.function.Function<T, R>
  - Supplier
    - java.util.function.Supplier<T>

Predicate

Listing 3.2 Working with a `Predicate`

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
}
public <T> List<T> filter(List<T> list, Predicate<T> p) {
    List<T> results = new ArrayList<>();
    for(T t: list) {
        if(p.test(t)) {
            results.add(t);
        }
    }
    return results;
}
Predicate<String> nonEmptyStringPredicate = (String s) -> !s.isEmpty();
List<String> nonEmpty = filter(listOfStrings, nonEmptyStringPredicate);
```

Consumer

Listing 3.3 Working with a `Consumer`

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
public <T> void forEach(List<T> list, Consumer<T> c) {
    for(T t: list) {
        c.accept(t);
    }

}
forEach(
         Arrays.asList(1,2,3,4,5),
        (Integer i) -> System.out.println(i)
       );
```

The lambda is the implementation of the accept method from Consumer.

Function

Listing 3.4 Working with a `Function`

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
public <T, R> List<R> map(List<T> list, Function<T, R> f) {
    List<R> result = new ArrayList<>();
    for(T t: list) {
        result.add(f.apply(t));
    }
    return result;
}
// [7, 2, 6]
List<Integer> l = map(
                       Arrays.asList("lambdas", "in", "action"),
                       (String s) -> s.length()
               );
```

Implements the apply method of Function

- Every Java type is either a reference type (for example, Byte, Integer, Object, List) or a primitive type (for example, int, double, byte, char).
- But generic parameters (for example, the T in Consumer<T>) can be bound only to reference type.
- Boxing vs. Unboxing
- Java 8 added a specialized version of the functional interface.
  - IntPredicate vs Predicate<Integer>
  - DoublePredicate, IntConsumer, LongBinaryOperator, IntFunction
  - ToIntFunction<T>, IntToDoubleFunction, and so on.

```java
public interface IntPredicate {
    boolean test(int t);
}
IntPredicate evenNumbers = (int i) -> i % 2 == 0;
evenNumbers.test(1000);
Predicate<Integer> oddNumbers = (Integer i) -> i % 2 != 0;
oddNumbers.test(1000);
```

True (no boxing)

False (boxing)

Table 3.2 Common functional interfaces added in Java 8

Functional interface	Predicate<T>	Consumer<T>
Predicate<T>	T -> boolean	IntPredicate, LongPredicate, DoublePredicate
Consumer<T>	T -> void	IntConsumer, LongConsumer, DoubleConsumer
Function<T, R>	T -> R	IntFunction<R>, IntToDoubleFunction, IntToLongFunction, LongFunction<R>, LongToDoubleFunction, LongToIntFunction, DoubleFunction<R>, DoubleToIntFunction, DoubleToLongFunction, ToIntFunction<T>, ToDoubleFunction<T>, ToLongFunction<T>

Supplier<T>	() -> T	BooleanSupplier, IntSupplier, LongSupplier, DoubleSupplier
UnaryOperator<T>	T -> T	IntUnaryOperator, LongUnaryOperator, DoubleUnaryOperator
BinaryOperator<T>	(T, T) -> T	IntBinaryOperator, LongBinaryOperator, DoubleBinaryOperator
BiPredicate<T, U>	(T, U) -> boolean	
BiConsumer<T, U>	(T, U) -> void	ObjIntConsumer<T>, ObjLongConsumer<T>, ObjDoubleConsumer<T>
BiFunction<T, U, R>	(T, U) -> R	ToIntBiFunction<T, U>, ToLongBiFunction<T, U>, ToDoubleBiFunction<T, U>

Table 3.3 Examples of lambdas with functional interfaces

Use case	Example of lambda	Matching functional interface
A boolean expression	`(List<String> list) -> list.isEmpty()`	`Predicate<List<String>>`
Creating objects	`() -> new Apple(10)`	`Supplier<Apple>`
Consuming from an object	`(Apple a) -> System.out.println(a.getWeight())`	`Consumer<Apple>`
Select/extract from an object	`(String s) -> s.length()`	`Function<String, Integer>` or `ToIntFunction<String>`
Combine two values	`(int a, int b) -> a * b`	`IntBinaryOperator`
Compare two objects	`(Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight())`	`Comparator<Apple>` or `BiFunction<Apple, Apple, Integer>` or `ToIntBiFunction<Apple, Apple>`

3.5 Type checking, type inference, and restrictions

- The type of a lambda is deduced from the context in which the lambda is used.
  - a method parameter that it is passed to
  - a local variable that it is assigned to
- *target type*

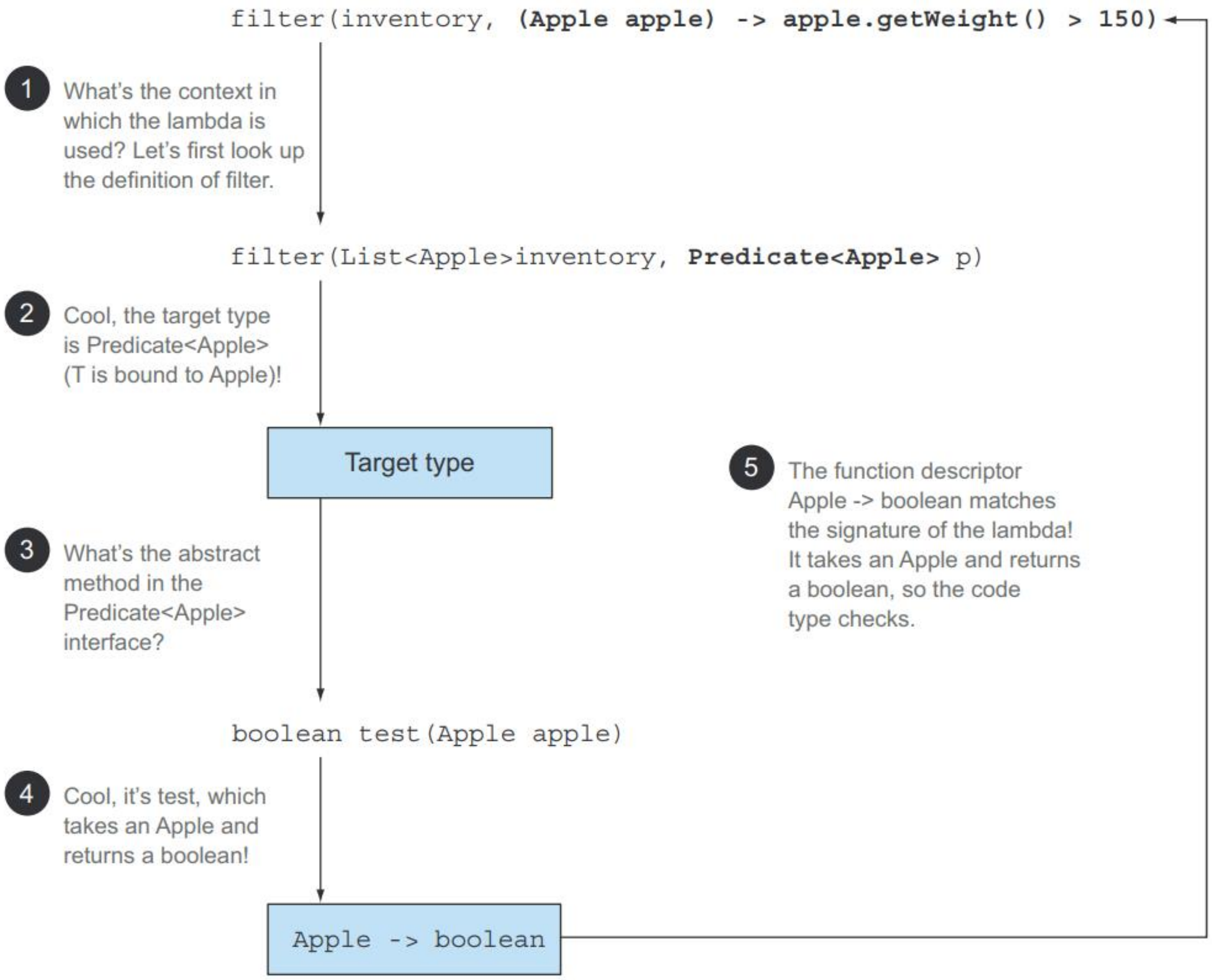


Figure 3.4 Deconstructing the type-checking process of a lambda expression

Same lambda, different functional interface

- Callable<Integer> c = () -> 42;
  PrivilegedAction<Integer> p = () -> 42;

- ```
  Comparator<Apple> c1 =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
  ToIntBiFunction<Apple, Apple> c2 =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
  BiFunction<Apple, Apple, Integer> c3 =
    (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
  ```
  ```

Type Inference

- The Java compiler deduces what functional interface to associate with a lambda expression from its surrounding context (the target type), meaning it can also deduce an appropriate signature for the lambda because the functional descriptor is available through the target type.

No explicit type on the parameter apple

```
List<Apple> greenApples =
        filter(inventory, apple -> GREEN.equals(apple.getColor()));
```

The benefits of code readability are more noticeable with lambda expressions that have several parameters. For example, here's how to create a `Comparator` object:

Without type inference

```
Comparator<Apple> c =
  (Apple a1, Apple a2) -> a1.getWeight().compareTo(a2.getWeight());
Comparator<Apple> c =
  (a1, a2) -> a1.getWeight().compareTo(a2.getWeight());
```

With type inference

Using Local Variables

All the lambda expressions we've shown so far used only their arguments inside their body. But lambda expressions are also allowed to use *free variables* (variables that aren't the parameters and are defined in an outer scope) like anonymous classes can. They're called *capturing lambdas*. For example, the following lambda captures the variable `portNumber`:

```
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
```

Nonetheless, there's a small twist. There are some restrictions on what you can do with these variables. Lambdas are allowed to capture (to reference in their bodies) instance variables and static variables without restrictions. But when local variables are captured, they have to be explicitly declared `final` or be effectively `final`. Lambda expressions can capture local variables that are assigned to only once. (Note: capturing an instance variable can be seen as capturing the final local variable `this`.) For

```java
int portNumber = 1337;
Runnable r = () -> System.out.println(portNumber);
portNumber = 31337;
```

Error: local variable portNumber is not final or effectively final.

3.6 Method references

- Method references can be seen as shorthand for lambdas calling only a specific method.
- Indeed, a method reference lets you create a lambda expression from an exiting method implementation.
- You can think of method reference as syntactic sugar for lambdas that refer only to a single method.

Before:

```
inventory.sort((Apple a1, Apple a2)
a1.getWeight().compareTo(a2.getWeight()));
```

After (using a method reference and `java.util.Comparator.comparing`):

```
inventory.sort(comparing(Apple::getWeight));
```

Your first method reference

Table 3.4 Examples of lambdas and method reference equivalents

Lambda	Method reference equivalent
`(Apple apple) -> apple.getWeight()`	`Apple::getWeight`
`() -> Thread.currentThread().dumpStack()`	`Thread.currentThread()::dumpStack`
`(str, i) -> str.substring(i)`	`String::substring`
`(String s) -> System.out.println(s)` `(String s) -> this.isValidName(s)`	`System.out::println` `this::isValidName`

Recipe for Constructing Method Referencesa

There are three main kinds of method references:

1. A method reference to a *static method* (for example, the method `parseInt` of `Integer`, written `Integer::parseInt`)
2. A method reference to an instance method of an arbitrary type (for example, the method `length` of a `String`, written `String::length`)
3. A method reference to an *instance method of an existing object or expression* (for example, suppose you have a local variable `expensiveTransaction` that holds an object of type `Transaction`, which supports an instance method `getValue`; you can write `expensiveTransaction::getValue`)

- For example, say you defined a helper method isValideName:

```
private boolean isValidName(String string) {
    return Character.isUpperCase(string.charAt(0));
}
```

You can now pass this method around in the context of a `Predicate<String>` using a method reference:

```
filter(words, this::isValidName)
```

-
```
List<String> str = Arrays.asList("a","b","A","B");
str.sort((s1, s2) -> s1.compareToIgnoreCase(s2));

List<String> str = Arrays.asList("a","b","A","B");
str.sort(String::compareToIgnoreCase);
```

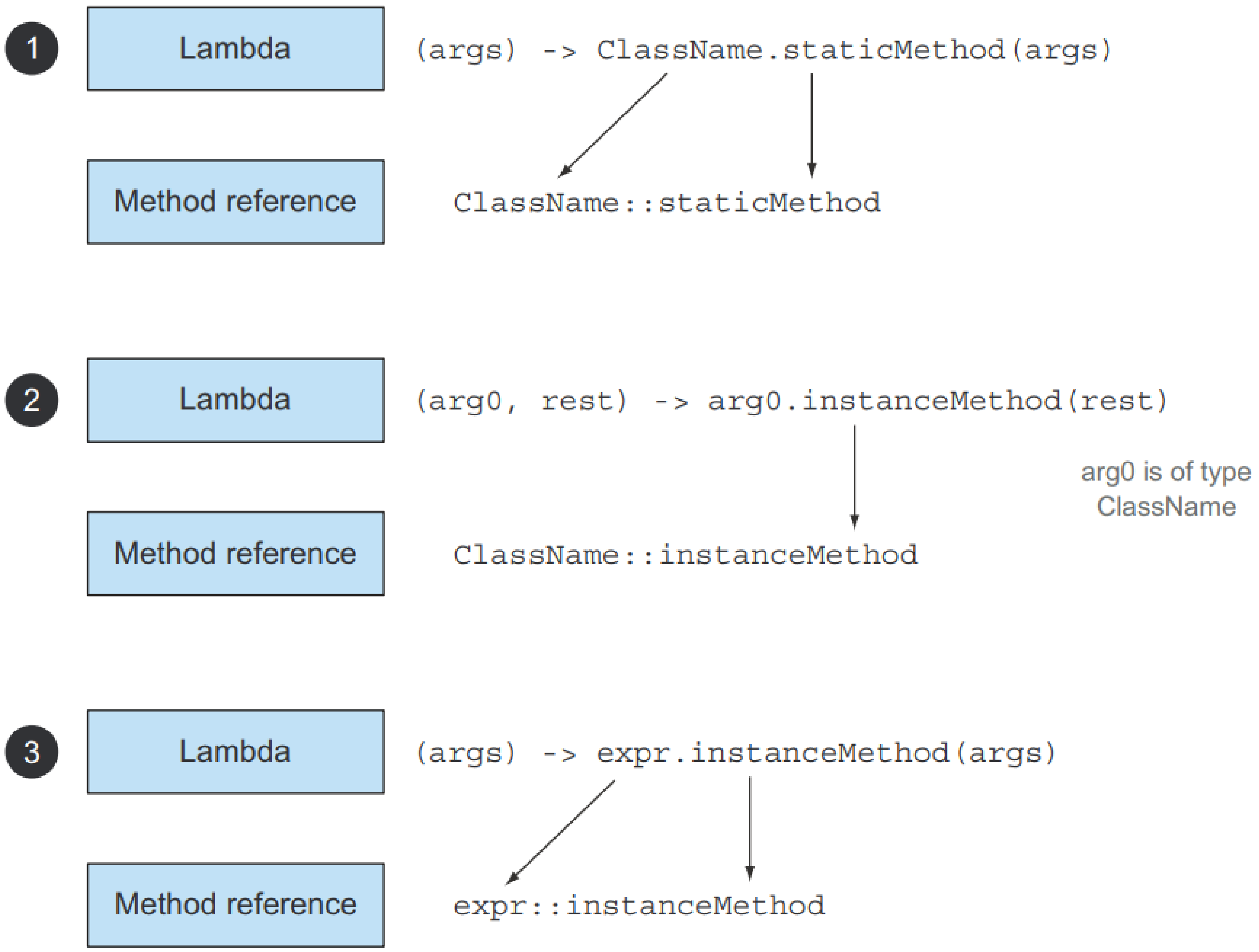


Figure 3.5 Recipes for constructing method references for three different types of lambda expressions

Constructor References

- You can create a reference to an existing constructor using its name and the keyword new as follows: ClassName::new.

For example, suppose there's a zero-argument constructor. This fits the signature `() -> Apple` of `Supplier`; you can do the following:

```
Supplier<Apple> c1 = Apple::new;
Apple a1 = c1.get();
```

Constructor reference to the default Apple() constructor

Calling Supplier's get method produces a new Apple.

which is equivalent to

```
Supplier<Apple> c1 = () -> new Apple();
Apple a1 = c1.get();
```

Lambda expression to create an Apple using the default constructor

Calling Supplier's get method produces a new Apple.

- If you have a constructor with signature Apple(Integer weight), it fits the signature of the Function interface, so you can do this

```java
Function<Integer, Apple> c2 = Apple::new;
Apple a2 = c2.apply(110);
```

Constructor reference to Apple (Integer weight)

Calling Function's apply method with a given weight produces an Apple.

which is equivalent to

Lambda expression to create an Apple with a given weight

```java
Function<Integer, Apple> c2 = (weight) -> new Apple(weight);
Apple a2 = c2.apply(110);
```

Calling Function's apply method with a given weight produces a new Apple object.

In the following code, each element of a `List` of `Integers` is passed to the constructor of `Apple` using a similar `map` method we defined earlier, resulting in a `List` of apples with various weights:

```
List<Integer> weights = Arrays.asList(7, 3, 4, 10);
List<Apple> apples = map(weights, Apple::new);
public List<Apple> map(List<Integer> list, Function<Integer, Apple> f) {
    List<Apple> result = new ArrayList<>();
    for(Integer i: list) {
        result.add(f.apply(i));
    }
    return result;
}
```

Passing a constructor reference to the map method

- If you have a two-argument constructor, Apple (Color color, Integer weight), it fits the signature of the BiFunction interface, so you can do this:

```
BiFunction<Color, Integer, Apple> c3 = Apple::new;
Apple a3 = c3.apply(GREEN, 110);
```

Constructor reference to Apple (Color color, Integer weight)

BiFunction's apply method with a given color and weight produces a new Apple object.

which is equivalent to

```
BiFunction<String, Integer, Apple> c3 =
    (color, weight) -> new Apple(color, weight);
Apple a3 = c3.apply(GREEN, 110);
```

Lambda expression to create an Apple with a given color and weight

BiFunction's apply method with a given color and weight produces a new Apple object.

- Use a Map to associate constructors with a string value. Create a method giveMeFruit that, given a String and an Integer, can create different types of fruits with different weights, as follows:

```
static Map<String, Function<Integer, Fruit>> map = new HashMap<>();
static {
    map.put("apple", Apple::new);
    map.put("orange", Orange::new);
    // etc...
}
public static Fruit giveMeFruit(String fruit, Integer weight){
    return map.get(fruit.toLowerCase())
               .apply(weight);
}
```

Get a Function<Integer, Fruit> from the map

Function's apply method with an Integer weight parameter creates the requested Fruit.

3.7 Putting lambdas and method references into practice

- Behavior Parameterization, anonymous classes, lambda expressions, and method references.
- Step 1: Pass code

```
void sort(Comparator<? super E> c);
```

```
public class AppleComparator implements Comparator<Apple> {
        public int compare(Apple a1, Apple a2){
                return a1.getWeight().compareTo(a2.getWeight());
        }
}
inventory.sort(new AppleComparator());
```

- Step 2: Use an anonymous class

```
inventory.sort(new Comparator<Apple>() {
    public int compare(Apple a1, Apple a2){
        return a1.getWeight().compareTo(a2.getWeight());
    }
});
```

- Step 3: Use lambda expressions

```
inventory.sort((Apple a1, Apple a2)
                -> a1.getWeight().compareTo(a2.getWeight())
);
```

```
inventory.sort((a1, a2) -> a1.getWeight().compareTo(a2.getWeight()));
```

- Comparator includes a static helper method called comparing that takes a Function extracting a Comparable key and produces a Comparator object.

```
Comparator<Apple> c = Comparator.comparing((Apple a) -> a.getWeight());
```

You can now rewrite your solution in a slightly more compact form:

```
import static java.util.Comparator.comparing;
inventory.sort(comparing(apple -> apple.getWeight()));
```

- Step 4: Use method references

```
inventory.sort(comparing(Apple::getWeight));
```

3.8 Useful methods to compose lambda expressions

- Many functional interfaces such as Comparator, Function, and Predicate that are used to pass lambda expressions provide methods that allow composition.
- *Composing Comparators*
  - Reversed Order
    - inventory.sort(comparing(Apple::getWeight).reversed());
  - Chaining Comparators

```
inventory.sort(comparing(Apple::getWeight)
        .reversed()
        .thenComparing(Apple::getCountry));
```

Sorts by decreasing weight

Sorts further by country when two apples have same weight

- *Composing Predicate*
  - negate, and, and or.

```
Predicate<Apple> notRedApple = redApple.negate();
```

Produces the negation of the existing Predicate object redApple

You may want to combine two lambdas to say that an apple is both red and heavy with the and method:

```
Predicate<Apple> redAndHeavyApple =
    redApple.and(apple -> apple.getWeight() > 150);
```

Chains two predicates to produce another Predicate object

```
Predicate<Apple> redAndHeavyAppleOrGreen =
    redApple.and(apple -> apple.getWeight() > 150)
            .or(apple -> GREEN.equals(a.getColor()));
```

Chains three predicates to construct a more complex Predicate object

- *Composing Functions*
  - andThen and compose

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.andThen(g);
int result = h.apply(1);
```

In mathematics you'd write g(f(x)) or (g o f)(x).

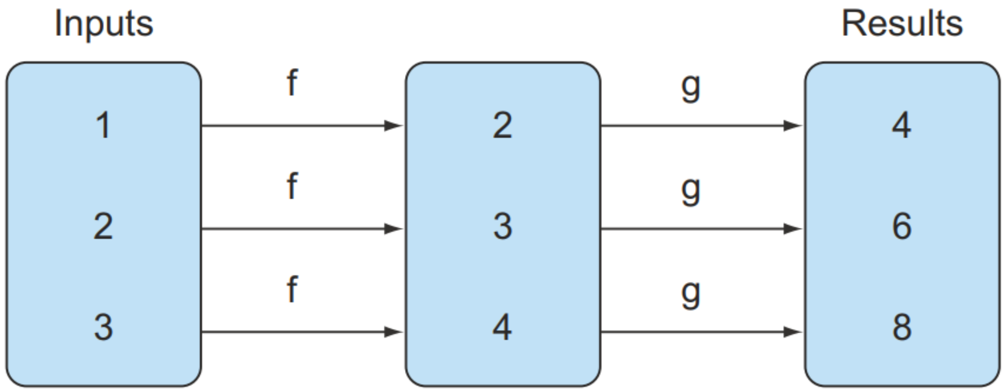
This returns 4.

```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
Function<Integer, Integer> h = f.compose(g);
int result = h.apply(1);
```

In mathematics you'd write f(g(x)) or (f o g)(x).

This returns 3.

`f.andThen(g)`




```
Function<Integer, Integer> f = x -> x + 1;
Function<Integer, Integer> g = x -> x * 2;
```

`f.compose(g)`

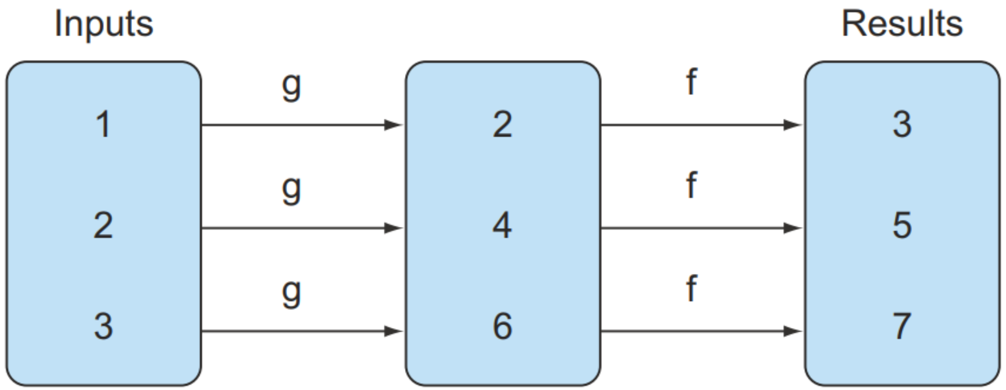


Figure 3.6 Using `andThen` versus `compose`

```
Function<String, String> addHeader = Letter::addHeader;
Function<String, String> transformationPipeline
  = addHeader.andThen(Letter::checkSpelling)
              .andThen(Letter::addFooter);
```

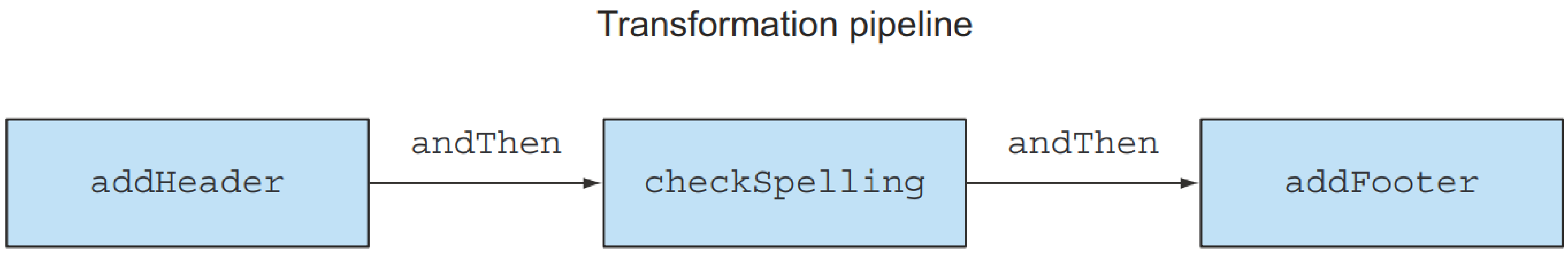


Figure 3.7 A transformation pipeline using `andThen`

3.9 Similar ideas from mathematics

- $f(x) = x + 10$
- $\int_3^7 f(x)dx$ or $\int_3^7 (x + 10)dx$

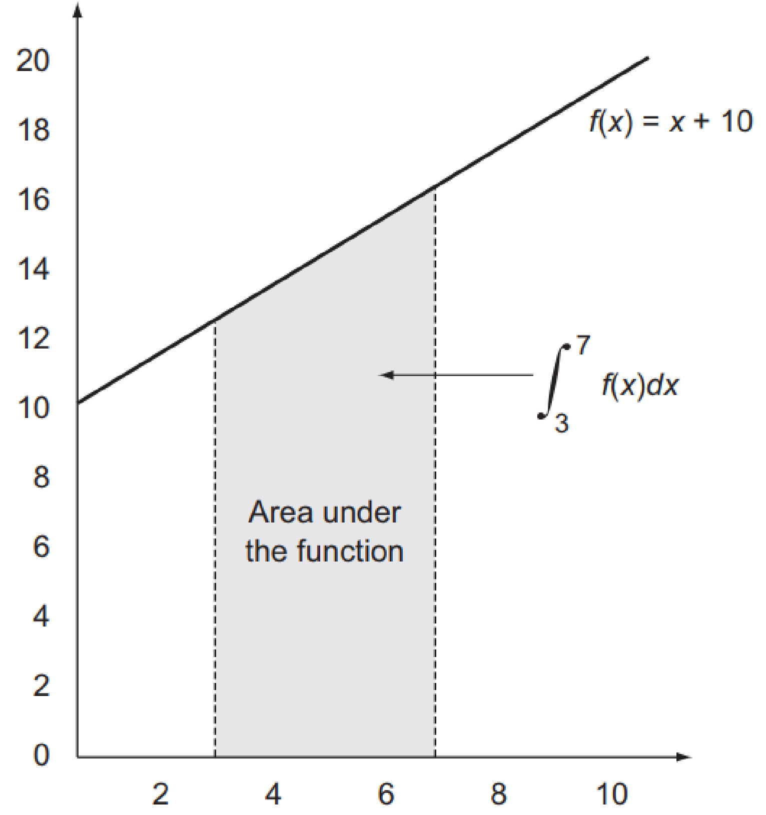


Figure 3.8 Area under the function `f(x) = x + 10` for x from 3 to 7

```
integrate(f, 3, 7)
integrate(x + 10, 3, 7)

integrate((double x) -> x + 10, 3, 7)
Integrate(C::f, 3, 7)
```

```
public double integrate((double -> double) f, double a, double b) {
      return (f(a) + f(b)) * (b - a) / 2.0
}
```

Incorrect Java code! (You can't write functions as you do in mathematics.)

But because lambda expressions can be used only in a context expecting a functional interface (in this case, `DoubleFunction`[4]), you have to write it the following way:

```
public double integrate(DoubleFunction<Double> f, double a, double b) {
      return (f.apply(a) + f.apply(b)) * (b - a) / 2.0;
}
```

or using `DoubleUnaryOperator`, which also avoids boxing the result:

```
public double integrate(DoubleUnaryOperator f, double a, double b) {
      return (f.applyAsDouble(a) + f.applyAsDouble(b)) * (b - a) / 2.0;
}
```